Earth System Modeling Framework

ESMF Reference Manual for Fortran

Version 3.1


25th July 2008
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, which informed the design of our regridding functionality
- The Model Coupling Toolkit (MCT) from Argonne National Laboratory, on which we based our sparse matrix multiply approach to general regridding
- 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 the overall ESMF architecture
- The Common Component Architecture (CCA) effort within the Department of Energy, from which we drew many ideas about how to design components
- The Vector Signal Image Processing Library (VSIPL) and its predecessors, which informed many aspects of our design, and the radar system software design group at Lincoln Laboratory
- The Portable, Extensible Toolkit for Scientific Computation (PETSc) package from Argonne National Laboratories, on which we based our initial makefile system
- The Community Climate System Model (CCSM) and Weather Research and Forecasting (WRF) modeling groups at NCAR, who have provided valuable feedback on the design and implementation of the framework
Contents

1 ESMF Overview 17

1 What is the Earth System Modeling Framework? 18

2 The ESMF Reference Manual for Fortran 18

3 How to Contact User Support and Find Additional Information 19

4 How to Submit Comments, Bug Reports, and Feature Requests 19

5 Conventions 19

5.1 Typeface and Diagram Conventions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.2 Method Name and Argument Conventions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.3 Locating Methods in this Manual . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

6 The ESMF Application Programming Interface 21

6.1 Standard Methods and Interface Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
6.2 Deep and Shallow Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
6.3 Special Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
6.4 The ESMF Data Hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
6.5 ESMF Spatial Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
6.6 ESMF Maps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
6.7 ESMF Specification Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
6.8 ESMF Utility Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

7 Overall Rules and Behavior 24

7.1 Allocation Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
7.2 Equality and Copying Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

8 Integrating ESMF into Applications 24

8.1 Using the ESMF Superstructure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

9 Global Options and Parameters 25

9.1 Flags . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

9.1.1 ESMF AllocFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
9.1.2 ESMF BlockingFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
9.1.3 ESMF ContextFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
9.1.4 ESMF CopyFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
9.1.5 ESMF DefaultFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
9.1.6 ESMF DecompFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
9.1.7 ESMF IndexFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
9.1.8 ESMF NeededFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
9.1.9 ESMF ReadyFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
9.1.10 ESMF ReduceFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
9.1.11 ESMF RegionFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
9.1.12 ESMF ReqForRestartFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
9.1.13 ESMF Status . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
9.1.14 ESMF ValidFlag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

9.2 Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
13.6.2 ESMF_GridCompSetEntryPoint .................................................. 60
13.6.3 ESMF_GridCompSetInternalState .................................................. 61
13.6.4 ESMF_GridCompSetServices ......................................................... 62

14 CplComp Class ................................................................................... 62
14.1 Description ...................................................................................... 62
14.2 Use and Examples ........................................................................... 63
  14.2.1 Specifying a User-Code SetServices Routine ............................. 63
  14.2.2 Specifying a User-Code Initialize Routine ............................... 64
  14.2.3 Specifying a User-Code Run Routine ...................................... 65
  14.2.4 Specifying a User-Code Finalize Routine ............................... 65
14.3 Restrictions and Future Work ......................................................... 67
14.4 Class API: Basic CplComp Methods ................................................. 67
  14.4.1 ESMF_CplCompCreate ............................................................... 67
  14.4.2 ESMF_CplCompDestroy ............................................................. 68
  14.4.3 ESMF_CplCompFinalize .............................................................. 69
  14.4.4 ESMF_CplCompGet ................................................................. 70
  14.4.5 ESMF_CplCompInitialize ......................................................... 70
  14.4.6 ESMF_CplCompPrint ............................................................... 71
  14.4.7 ESMF_CplCompRun ................................................................. 72
  14.4.8 ESMF_CplCompSet ................................................................. 72
  14.4.9 ESMF_CplCompValidate ......................................................... 73
  14.4.10 ESMF_CplCompWait .............................................................. 74
  14.4.11 ESMF_CplCompIsPetLocal ..................................................... 74
14.5 Class API: SetServices and Related Methods ................................... 74
  14.5.1 ESMF_CplCompGetInternalState ............................................. 74
  14.5.2 ESMF_CplCompSetEntryPoint .................................................. 75
  14.5.3 ESMF_CplCompSetInternalState ............................................. 76
  14.5.4 ESMF_CplCompSetServices .................................................... 77

15 State Class ......................................................................................... 77
15.1 Description ...................................................................................... 77
15.2 State Options ................................................................................   78
  15.2.1 ESMF_StateItemType ............................................................... 78
  15.2.2 ESMF_StateType ................................................................. 78
15.3 Use and Examples ........................................................................... 78
  15.3.1 Empty State Create ................................................................. 79
  15.3.2 Adding Items to a State ........................................................... 80
  15.3.3 Adding Placeholders to a State .............................................. 80
  15.3.4 Marking an Item Needed ......................................................... 80
  15.3.5 Creating a Needed Item .......................................................... 81
  15.3.6 Initialization and SetServices Routines ................................... 81
  15.3.7 Creating Components on subsets of the current PET list ........... 82
  15.3.8 Invoking Components on a subset of the Parent PETs ............. 83
  15.3.9 Using State Reconcile ............................................................ 83
15.4 Restrictions and Future Work ......................................................... 83
15.5 Design and Implementation Notes .................................................. 84
15.6 Object Model ................................................................................. 87
15.7 Class API: Basic State Methods ..................................................... 87
  15.7.1 ESMF_StateAdd ................................................................. 87
  15.7.2 ESMF_StateAdd ................................................................. 88
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2.5</td>
<td>Create Field with Grid and Array</td>
</tr>
<tr>
<td>18.2.6</td>
<td>Create an empty Field and finish it with FieldSetCommit</td>
</tr>
<tr>
<td>18.2.7</td>
<td>Create 7D Field with 5D Grid and 2D ungridded bounds from Fortran data array</td>
</tr>
<tr>
<td>18.2.8</td>
<td>Create 2D Field with 2D Grid and Fortran data array</td>
</tr>
<tr>
<td>18.2.9</td>
<td>Create 3D Field with 2D Grid and 3D Fortran data array</td>
</tr>
<tr>
<td>18.2.10</td>
<td>Create 2D Field with 2D Grid and Fortran data array with gridToFieldMap</td>
</tr>
<tr>
<td>18.2.11</td>
<td>Create 3D Field with 2D Grid and 3D Fortran data array with halos</td>
</tr>
<tr>
<td>18.3</td>
<td>Restrictions and Future Work</td>
</tr>
<tr>
<td>18.4</td>
<td>Design and Implementation Notes</td>
</tr>
<tr>
<td>18.5</td>
<td>Class API</td>
</tr>
<tr>
<td>18.5.1</td>
<td>ESMF FieldCreate</td>
</tr>
<tr>
<td>18.5.2</td>
<td>ESMF FieldCreate</td>
</tr>
<tr>
<td>18.5.3</td>
<td>ESMF FieldCreate</td>
</tr>
<tr>
<td>18.5.4</td>
<td>ESMF FieldCreateEmpty</td>
</tr>
<tr>
<td>18.5.5</td>
<td>ESMF FieldDestroy</td>
</tr>
<tr>
<td>18.5.6</td>
<td>ESMF FieldGet</td>
</tr>
<tr>
<td>18.5.7</td>
<td>ESMF FieldGet</td>
</tr>
<tr>
<td>18.5.8</td>
<td>ESMF FieldGet</td>
</tr>
<tr>
<td>18.5.9</td>
<td>ESMF FieldPrint</td>
</tr>
<tr>
<td>18.5.10</td>
<td>ESMF FieldSetCommit</td>
</tr>
<tr>
<td>18.5.11</td>
<td>ESMF FieldValidate</td>
</tr>
<tr>
<td>19.1</td>
<td>ArrayBundle Class</td>
</tr>
<tr>
<td>19.1.1</td>
<td>Description</td>
</tr>
<tr>
<td>19.1.2</td>
<td>Use and Examples</td>
</tr>
<tr>
<td>19.1.2.1</td>
<td>ArrayBundle creation from a list of Arrays</td>
</tr>
<tr>
<td>19.1.2.2</td>
<td>Access Arrays inside the ArrayBundle</td>
</tr>
<tr>
<td>19.1.2.3</td>
<td>Destroy an ArrayBundle and its constituents</td>
</tr>
<tr>
<td>19.3</td>
<td>Restrictions and Future Work</td>
</tr>
<tr>
<td>19.4</td>
<td>Design and Implementation Notes</td>
</tr>
<tr>
<td>19.5</td>
<td>Class API</td>
</tr>
<tr>
<td>19.5.1</td>
<td>ESMF ArrayBundleCreate</td>
</tr>
<tr>
<td>19.5.2</td>
<td>ESMF ArrayBundleDestroy</td>
</tr>
<tr>
<td>19.5.3</td>
<td>ESMF ArrayBundleGet</td>
</tr>
<tr>
<td>19.5.4</td>
<td>ESMF ArrayBundlePrint</td>
</tr>
<tr>
<td>19.5.5</td>
<td>ESMF ArrayBundleRedist</td>
</tr>
<tr>
<td>19.5.6</td>
<td>ESMF ArrayBundleRedistRelease</td>
</tr>
<tr>
<td>19.5.7</td>
<td>ESMF ArrayBundleRedistStore</td>
</tr>
<tr>
<td>19.5.8</td>
<td>ESMF ArrayBundleRedistStore</td>
</tr>
<tr>
<td>19.5.9</td>
<td>ESMF ArrayBundleSMM</td>
</tr>
<tr>
<td>19.5.10</td>
<td>ESMF ArrayBundleSMMRelease</td>
</tr>
<tr>
<td>19.5.11</td>
<td>ESMF ArrayBundleSMMStore</td>
</tr>
<tr>
<td>19.5.12</td>
<td>ESMF ArrayBundleSMMStore</td>
</tr>
<tr>
<td>20.1</td>
<td>Array Class</td>
</tr>
<tr>
<td>20.1.1</td>
<td>Description</td>
</tr>
<tr>
<td>20.2</td>
<td>Use and Examples</td>
</tr>
<tr>
<td>20.2.1</td>
<td>Array from native Fortran90 array with 1 DE per PET</td>
</tr>
<tr>
<td>20.2.2</td>
<td>Array from native Fortran90 array with elements for halo</td>
</tr>
<tr>
<td>20.2.3</td>
<td>Array from ESMF_LocalArray</td>
</tr>
<tr>
<td>20.2.4</td>
<td>Array creation with automatic memory allocation</td>
</tr>
</tbody>
</table>
20.2.5 Native language memory access – the most general way .......................... 153
20.2.6 Regions and default bounds .................................................................. 154
20.2.7 Array bounds ....................................................................................... 156
20.2.8 Computational region and extra elements for halo or padding .............. 157
20.2.9 1D and 3D Arrays ............................................................................... 159
20.2.10 Working with Arrays of different rank ................................................ 160
20.2.11 Array and DistGrid rank – 2D+1 Arrays .............................................. 161
20.2.12 Arrays with replicated dimensions ...................................................... 163
20.2.13 Communication – Scatter and Gather ................................................. 166
20.2.14 Communication – Redist .................................................................... 169
20.2.15 Communication – SparseMatMul ......................................................... 173

20.3 Restrictions and Future Work ................................................................... 179
20.4 Design and Implementation Notes ............................................................ 179
20.5 Class API ................................................................................................. 180
20.5.1 ESMF_ArrayCreate .............................................................................. 180
20.5.2 ESMF_ArrayCreate .............................................................................. 182
20.5.3 ESMF_ArrayCreate .............................................................................. 184
20.5.4 ESMF_ArrayCreate .............................................................................. 186
20.5.5 ESMF_ArrayDestroy ............................................................................ 186
20.5.6 ESMF_ArrayGather ............................................................................. 186
20.5.7 ESMF_ArrayGet .................................................................................... 187
20.5.8 ESMF_ArrayGet .................................................................................... 190
20.5.9 ESMF_ArrayGet .................................................................................... 190
20.5.10 ESMF_ArrayGet .................................................................................. 191
20.5.11 ESMF_ArrayPrint .............................................................................. 191
20.5.12 ESMF_ArrayRedist ............................................................................ 192
20.5.13 ESMF_ArrayRedistRelease ................................................................. 193
20.5.14 ESMF_ArrayRedistStore .................................................................... 193
20.5.15 ESMF_ArrayRedistStore .................................................................... 194
20.5.16 ESMF_ArrayScatter ........................................................................... 195
20.5.17 ESMF_ArraySet ................................................................................. 195
20.5.18 ESMF_ArraySet ................................................................................. 196
20.5.19 ESMF_ArraySMM .............................................................................. 197
20.5.20 ESMF_ArraySMMRelease ................................................................. 198
20.5.21 ESMF_ArraySMMStore .................................................................... 198
20.5.22 ESMF_ArraySMMStore .................................................................... 199
20.5.23 ESMF_ArrayValidate ....................................................................... 200

21 LocalArray Class ......................................................................................... 200
21.1 Description ............................................................................................... 200
21.2 Restrictions and Future Work .................................................................. 201
21.3 Class API ................................................................................................. 201
21.3.1 ESMF_LocalArrayCreate ................................................................. 201
21.3.2 ESMF_LocalArrayCreate ................................................................. 201
21.3.3 ESMF_LocalArrayCreate ................................................................. 201
21.3.4 ESMF_LocalArrayCreate ................................................................. 203
21.3.5 ESMF_LocalArrayDestroy ............................................................... 203
21.3.6 ESMF_LocalArrayGet ....................................................................... 204
21.3.7 ESMF_LocalArrayGet ....................................................................... 205
### 22 ArraySpec Class

- **22.1 Description**
- **22.2 Use and Examples**
  - 22.2.1 Setting ArraySpec Values
  - 22.2.2 Getting ArraySpec Values
- **22.3 Restrictions and Future Work**
- **22.4 Design and Implementation Notes**
- **22.5 Class API**
  - 22.5.1 ESMF_ArraySpecGet
  - 22.5.2 ESMF_ArraySpecSet
  - 22.5.3 ESMF_ArraySpecValidate
  - 22.5.4 ESMF_ArraySpecPrint

### 23 Grid Class

- **23.1 Description**
  - 23.1.1 Grid Representation in ESMF
  - 23.1.2 Supported Grids
  - 23.1.3 Grid Topologies and Periodicity
  - 23.1.4 Grid Distribution
  - 23.1.5 Grid Coordinates
  - 23.1.6 Coordinate Specification and Generation
  - 23.1.7 Staggering
  - 23.1.8 Options for Building Grids
- **23.2 Use and Examples**
  - 23.2.1 Shortcut Creation Method for Single-Tile Grids
  - 23.2.2 Creating a 2D Regularly Distributed Rectilinear Grid With Uniformly Spaced Coordinates
  - 23.2.3 Creating a 2D Irregularly Distributed Rectilinear Grid With Uniformly Spaced Coordinates
  - 23.2.4 Creating a 2D Irregularly Distributed Grid With Curvilinear Coordinates
  - 23.2.5 Creating an Irregularly Distributed Rectilinear Grid with a Non-Distributed Vertical Dimension
  - 23.2.6 Creating an Empty Grid in a Parent Component for Completion in a Child Component
  - 23.2.7 Associating Coordinates with Stagger Locations
  - 23.2.8 Specifying the Relationship of Coordinate Arrays to Index Space Dimensions
  - 23.2.9 Accessing Grid Coordinates
  - 23.2.10 Grid Regions and Bounds
  - 23.2.11 Getting Grid Coordinate Bounds
  - 23.2.12 Getting Grid Stagger Location Bounds
  - 23.2.13 Getting Grid Stagger Location Information
  - 23.2.14 Creating an Array at a Stagger Location
  - 23.2.15 Creating More Complex Grids Using DistGrid
  - 23.2.16 Specifying Custom Stagger Locations
  - 23.2.17 Specifying Custom Stagger Padding
- **23.3 Restrictions and Future Work**
- **23.4 Design and Implementation Notes**
  - 23.4.1 Grid Topology
  - 23.4.2 Storage and Distribution of Stagger Locations in Grid
- **23.5 Grid Options**
  - 23.5.1 ESMF_GridConn
  - 23.5.2 ESMF_GridStatus
  - 23.5.3 ESMF_StaggerLoc
- **23.6 Class API: General Grid Methods**
  - 23.6.1 ESMF_GridAddCoord
### 23.6.2 ESMF_GridCreate

23.6.3 ESMF_GridCreateEmpty

23.6.4 ESMF_GridCreateShapeTile

23.6.5 ESMF_GridCreateShapeTile

23.6.6 ESMF_GridDestroy

23.6.7 ESMF_GridGet

23.6.8 ESMF_GridGet

23.6.9 ESMF_GridGet

23.6.10 ESMF_GridGetCoord

23.6.11 ESMF_GridGetCoord

23.6.12 ESMF_GridGetCoord

23.6.13 ESMF_GridGetStatus

23.6.14 ESMF_GridMatch

23.6.15 ESMF_GridSetCoord

23.6.16 ESMF_GridSetCommitShapeTile

23.6.17 ESMF_GridSetCommitShapeTile

23.6.18 ESMF_GridValidate

### 23.7 Class API: StaggerLoc Methods

23.7.1 ESMF_StaggerLocSet

23.7.2 ESMF_StaggerLocSet

23.7.3 ESMF_StaggerLocString

23.7.4 ESMF_StaggerLocPrint

### 24 DistGrid Class

4 DistGrid Class

24.1 Description

24.2 Use and Examples

24.2.1 Single patch DistGrid with regular decomposition

24.2.2 DistGrid and DELayout

24.2.3 Single patch DistGrid with decomposition by DE blocks

24.2.4 Single patch DistGrid with periodic boundaries

24.2.5 2D patchwork DistGrid with regular decomposition

24.3 Restrictions and Future Work

24.4 Design and Implementation Notes

24.5 Class API

24.5.1 ESMF_DistGridCreate

24.5.2 ESMF_DistGridCreate

24.5.3 ESMF_DistGridCreate

24.5.4 ESMF_DistGridCreate

24.5.5 ESMF_DistGridDestroy

24.5.6 ESMF_DistGridGet

24.5.7 ESMF_DistGridGet

24.5.8 ESMF_DistGridGet

24.5.9 ESMF_DistGridPrint

24.5.10 ESMF_DistGridMatch

24.5.11 ESMF_DistGridValidate

24.5.12 ESMF_DistGridConnection
30.5.1 ESMF_CalendarOperator(==) .................................................. 310
30.5.2 ESMF_CalendarOperator(==) .................................................. 311
30.5.3 ESMF_CalendarOperator(==) .................................................. 311
30.5.4 ESMF_CalendarOperator(==) .................................................. 312
30.5.5 ESMF_CalendarOperator(/=) ................................................... 312
30.5.6 ESMF_CalendarOperator(/=) ................................................... 313
30.5.7 ESMF_CalendarOperator(/=) ................................................... 313
30.5.8 ESMF_CalendarOperator(/=) ................................................... 314
30.5.9 ESMF_CalendarCreate ............................................................ 314
30.5.10 ESMF_CalendarCreate ......................................................... 315
30.5.11 ESMF_CalendarCreate ......................................................... 315
30.5.12 ESMF_CalendarDestroy ....................................................... 316
30.5.13 ESMF_CalendarGet ............................................................... 316
30.5.14 ESMF_CalendarIsLeapYear ................................................... 317
30.5.15 ESMF_CalendarIsLeapYear ................................................... 318
30.5.16 ESMF_CalendarPrint ............................................................ 319
30.5.17 ESMF_CalendarSet ............................................................... 319
30.5.18 ESMF_CalendarSet ............................................................... 320
30.5.19 ESMF_CalendarSetDefault .................................................. 321
30.5.20 ESMF_CalendarSetDefault .................................................. 321
30.5.21 ESMF_CalendarValidate ..................................................... 322

31 Time Class .............................................................................. 323
31.1 Description ........................................................................... 323
31.2 Use and Examples ................................................................. 323
31.2.1 Time Initialization ............................................................... 323
31.2.2 Time Increment ................................................................. 324
31.2.3 Time Comparison ............................................................... 324
31.3 Restrictions and Future Work .................................................. 324
31.4 Class API .............................................................................. 325
31.4.1 ESMF_TimeOperator(+) ......................................................... 325
31.4.2 ESMF_TimeOperator(-) ......................................................... 325
31.4.3 ESMF_TimeOperator(-) ......................................................... 326
31.4.4 ESMF_TimeOperator(==) ....................................................... 326
31.4.5 ESMF_TimeOperator(/=) ....................................................... 327
31.4.6 ESMF_TimeOperator(<) ......................................................... 327
31.4.7 ESMF_TimeOperator(<=) ...................................................... 328
31.4.8 ESMF_TimeOperator(>) ......................................................... 328
31.4.9 ESMF_TimeOperator(>=) ...................................................... 329
31.4.10 ESMF_TimeGet ................................................................. 329
31.4.11 ESMF_TimeIsLeapYear ....................................................... 332
31.4.12 ESMF_TimeIsSameCalendar ............................................... 332
31.4.13 ESMF_TimePrint ............................................................... 333
31.4.14 ESMF_TimeSet ................................................................. 334
31.4.15 ESMF_TimeSyncToRealTime .............................................. 336
31.4.16 ESMF_TimeValidate .......................................................... 336
32. TimeInterval Class

32.1 Description .................................................................................................................. 338
32.2 Use and Examples.......................................................................................................... 338
  32.2.1 Time Interval Initialization ...................................................................................... 339
  32.2.2 Time Interval Conversion ....................................................................................... 339
  32.2.3 Time Interval Difference ......................................................................................... 339
  32.2.4 Time Interval Multiplication .................................................................................. 339
  32.2.5 Time Interval Comparison ...................................................................................... 340
32.3 Restrictions and Future Work ...................................................................................... 340
32.4 Class API ...................................................................................................................... 340
  32.4.1 ESMF_TimeIntervalOperator(+) ........................................................................... 340
  32.4.2 ESMF_TimeIntervalOperator(-) ........................................................................... 341
  32.4.3 ESMF_TimeIntervalOperator(-) ........................................................................... 341
  32.4.4 ESMF_TimeIntervalOperator(/) ............................................................................. 341
  32.4.5 ESMF_TimeIntervalOperator(/) ............................................................................. 342
  32.4.6 ESMF_TimeIntervalFunction(MOD) ..................................................................... 342
  32.4.7 ESMF_TimeIntervalOperator(x) .......................................................................... 343
  32.4.8 ESMF_TimeIntervalOperator(x) .......................................................................... 343
  32.4.9 ESMF_TimeIntervalOperator(==) ........................................................................ 344
  32.4.10 ESMF_TimeIntervalOperator(/=) ......................................................................... 344
  32.4.11 ESMF_TimeIntervalOperator(<) ......................................................................... 345
  32.4.12 ESMF_TimeIntervalOperator(<=) ....................................................................... 345
  32.4.13 ESMF_TimeIntervalOperator(>) ...................................................................... 346
  32.4.14 ESMF_TimeIntervalOperator(>=) ...................................................................... 346
  32.4.15 ESMF_TimeIntervalAbsValue ............................................................................. 347
  32.4.16 ESMF_TimeIntervalGet ...................................................................................... 347
  32.4.17 ESMF_TimeIntervalGet ...................................................................................... 350
  32.4.18 ESMF_TimeIntervalGet ...................................................................................... 352
  32.4.19 ESMF_TimeIntervalGet ...................................................................................... 354
  32.4.20 ESMF_TimeIntervalNegAbsValue ...................................................................... 357
  32.4.21 ESMF_TimeIntervalPrint .................................................................................... 357
  32.4.22 ESMF_TimeIntervalSet ....................................................................................... 358
  32.4.23 ESMF_TimeIntervalSet ....................................................................................... 360
  32.4.24 ESMF_TimeIntervalSet ....................................................................................... 361
  32.4.25 ESMF_TimeIntervalSet ....................................................................................... 363
  32.4.26 ESMF_TimeIntervalValidate ............................................................................. 365

33. Clock Class ...................................................................................................................... 366

33.1 Description .................................................................................................................... 366
33.2 Clock Options ................................................................................................................ 366
  33.2.1 ESMF Direction ..................................................................................................... 366
33.3 Use and Examples ......................................................................................................... 366
  33.3.1 Clock Creation ...................................................................................................... 367
  33.3.2 Clock Advance ..................................................................................................... 368
  33.3.3 Clock Examination ............................................................................................... 368
  33.3.4 Clock Reversal ..................................................................................................... 368
  33.3.5 Clock Destruction ............................................................................................... 369
33.4 Restrictions and Future Work ...................................................................................... 369
33.5 Class API ...................................................................................................................... 369
  33.5.1 ESMF_ClockOperator(==) .................................................................................. 369
  33.5.2 ESMF_ClockOperator(/=) ................................................................................... 370
33.5.3 ESMF_ClockAdvance
33.5.4 ESMF_ClockCreate
33.5.5 ESMF_ClockCreate
33.5.6 ESMF_ClockDestroy
33.5.7 ESMF_ClockGet
33.5.8 ESMF_ClockGetAlarm
33.5.9 ESMF_ClockGetAlarmList
33.5.10 ESMF_ClockGetNextTime
33.5.11 ESMF_ClockIsDone
33.5.12 ESMF_ClockIsReverse
33.5.13 ESMF_ClockIsStopTime
33.5.14 ESMF_ClockIsStopTimeEnabled
33.5.15 ESMF_ClockPrint
33.5.16 ESMF_ClockSet
33.5.17 ESMF_ClockStopTimeDisable
33.5.18 ESMF_ClockStopTimeEnable
33.5.19 ESMF_ClockSyncToRealTime
33.5.20 ESMF_ClockValidate

34 Alarm Class

34.1 Description
34.2 Alarm Options
34.3 Use and Examples
34.4 Restrictions and Future Work
34.5 Design and Implementation Notes
34.6 Class API
35  Config Class

35.1 Description

35.1.1 Package History

35.2 Use and Examples

35.2.1 Resource File

35.2.2 Package History

35.2.3 A Quick Overview

35.2.4 Common Code Arguments

35.2.5 Creation of a Config

35.2.6 Retrieval of constants

35.2.7 Retrieval of file names

35.2.8 Retrieval of tables

35.2.9 Destruction of a Config

35.3 Class API

35.3.1 ESMF_ConfigCreate

35.3.2 ESMF_ConfigDestroy

35.3.3 ESMF_ConfigFindLabel

35.3.4 ESMF_ConfigGetAttribute

35.3.5 ESMF_ConfigGetAttr

35.3.6 ESMF_ConfigGetChar

35.3.7 ESMF_ConfigGetDim

35.3.8 ESMF_ConfigGetLen

35.3.9 ESMF_ConfigLoadFile

35.3.10 ESMF_ConfigNextLine

35.3.11 ESMF_ConfigSetAttribute

35.3.12 ESMF_ConfigValidate

36  LogErr Class

36.1 Description

36.2 LogErr Options

36.2.1 ESMF_HaltType

36.2.2 ESMF_MsgType

36.2.3 ESMF_LogType

36.3 Use and Examples

36.3.1 Default Log

36.3.2 User Created Log

36.3.3 Get and Set

36.4 Restrictions and Future Work

36.5 Design and Implementation Notes

36.6 Object Model

36.7 Class API

36.7.1 ESMF_LogClose

36.7.2 ESMF_LogFlush

36.7.3 ESMF_LogFoundAllocError

36.7.4 ESMF_LogFoundError

36.7.5 ESMF_LogMsgFoundAllocError

36.7.6 ESMF_LogMsgFoundError

36.7.7 ESMF_LogMsgSetError

36.7.8 ESMF_LogOpen

36.7.9 ESMF_LogSet

36.7.10 ESMF_LogWrite
37 DELayout Class 418

37.1 Description .......................... 418
37.2 Use and Examples .......................... 418
  37.2.1 Default DELayout .......................... 418
  37.2.2 DELayout with specified number of DEs .......................... 419
  37.2.3 DELayout with computational and communication weights .......................... 419
  37.2.4 DELayout from petMap .......................... 420
  37.2.5 DELayout from petMap with multiple DEs per PET .......................... 420
  37.2.6 Working with a DELayout - simple 1-to-1 DE to PET mapping .......................... 420
  37.2.7 Working with a DELayout - general DE to PET mapping .......................... 421
  37.2.8 Work queue dynamic load balancing .......................... 421
37.3 Restrictions and Future Work .......................... 422
37.4 Design and Implementation Notes .......................... 422
37.5 Class API .......................... 422
  37.5.1 ESMF_DELayoutCreate .......................... 422
  37.5.2 ESMF_DELayoutCreate .......................... 423
  37.5.3 ESMF_DELayoutCreate .......................... 423
  37.5.4 ESMF_DELayoutDestroy .......................... 425
  37.5.5 ESMF_DELayoutGet .......................... 425
  37.5.6 ESMF_DELayoutPrint .......................... 426
  37.5.7 ESMF_DELayoutServiceComplete .......................... 427
  37.5.8 ESMF_DELayoutServiceOffer .......................... 427
  37.5.9 ESMF_DELayoutValidate .......................... 428

38 VM Class 428

38.1 Description .......................... 428
38.2 Use and Examples .......................... 428
  38.2.1 VM Default Basics Example .......................... 429
  38.2.2 VMGet MPI Communicator Example .......................... 430
  38.2.3 ESMF inside user MPI application .......................... 430
  38.2.4 ESMF inside user defined MPI communicator .......................... 431
  38.2.5 VMSend/VMRecv Example .......................... 431
  38.2.6 VMScatter/VMGather Example .......................... 431
  38.2.7 VMAllFullReduce Example .......................... 431
  38.2.8 VM Component Example .......................... 432
38.3 Restrictions and Future Work .......................... 433
38.4 Design and Implementation Notes .......................... 434
38.5 Class API .......................... 436
  38.5.1 ESMF_VMAllFullReduce .......................... 436
  38.5.2 ESMF_VMAllGather .......................... 437
  38.5.3 ESMF_VMAllGatherV .......................... 438
  38.5.4 ESMF_VMAllReduce .......................... 439
  38.5.5 ESMF_VMAllToAllV .......................... 440
  38.5.6 ESMF_VMBroadcast .......................... 441
  38.5.7 ESMF_VMBroadcastV .......................... 442
  38.5.8 ESMF_VMGather .......................... 443
  38.5.9 ESMF_VMGatherV .......................... 444
  38.5.10 ESMF_VMGet .......................... 444
  38.5.11 ESMF_VMGetGlobal .......................... 445
  38.5.12 ESMF_VMGetCurrent .......................... 446
  38.5.13 ESMF_VMGetPETLocalInfo .......................... 446
Part I

ESMF Overview
1 What is the Earth System Modeling Framework?

The Earth System Modeling Framework (ESMF) is a suite of software tools for developing high-performance, multi-component Earth science modeling applications. Such applications may include a few or dozens of components representing atmospheric, oceanic, terrestrial, or other physical domains, and their constituent processes (dynamical, chemical, biological, etc.). Often these components are developed by different groups independently, and must be “coupled” together using software that transfers and transforms data among the components in order to form functional simulations.

ESMF supports the development of these complex applications in a number of ways. It introduces a set of simple, consistent component interfaces that apply to all types of components, including couplers themselves. These interfaces expose in an obvious way the inputs and outputs of each component. It offers a variety of data structures for transferring data between components, and libraries for regridding, time advancement, and other common modeling functions. Finally, it provides a growing set of tools for using metadata to describe components and their input and output fields. This capability is important because components that are self-describing can be integrated more easily into automated workflows, model and dataset distribution and analysis portals, and other emerging “semantically enabled” computational environments.

ESMF is not a single Earth system model into which all components must fit, and its distribution doesn’t contain any scientific code. Rather it provides a way of structuring components so that they can be used in many different user-written applications and contexts with minimal code modification, and so they can be coupled together in new configurations with relative ease. The idea is to create many components across a broad community, and so to encourage new collaborations and combinations.

ESMF offers the flexibility needed by this diverse user base. It is tested nightly on more than two dozen platform/compiler combinations; can be run on one processor or thousands; supports shared and distributed memory programming models and a hybrid model; can run components sequentially (on all the same processors) or concurrently (on mutually exclusive processors); and supports single executable or multiple executable modes.

ESMF’s generality and breadth of function can make it daunting for the novice user. To help users navigate the software, we try to apply consistent names and behavior throughout and to provide many examples. The large-scale structure of the software is straightforward. The utilities and data structures for building modeling components are called the ESMF *infrastructure*. The coupling interfaces and drivers are called the *superstructure*. User code sits between these two layers, making calls to the infrastructure libraries underneath and being scheduled and synchronized by the superstructure above. The configuration resembles a sandwich, as shown in Figure 1.

ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap their components in the ESMF superstructure in order to utilize framework coupling services. Either way, we encourage users to contact our [support team](mailto:support@earthsystemmodeling.org) if questions arise about how to best use the software, or how to structure their application. ESMF is more than software; it’s a group of people dedicated to realizing the vision of a collaborative model development community that spans institutional and national bounds.

2 The ESMF Reference Manual for Fortran

ESMF has a complete set of Fortran interfaces and some C interfaces. This *ESMF Reference Manual* is a listing of ESMF interfaces for Fortran.

Interfaces are grouped by class. A class is comprised of the data and methods for 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 usually represent large pieces of functionality such as atmosphere and ocean models, and States, which are the data structures used to transfer data between Components. There are both data structures and utilities in the ESMF infrastructure. Data structures include multi-dimensional Arrays, Fields that are comprised of an Array and a Grid, and collections of Arrays and Fields called ArrayBundles and FieldBundles, respectively. There are utility libraries for data decomposition and communications, time management, logging and error handling, and application configuration.

---

1 Since the customer base for it is small, we have not yet prepared a comprehensive reference manual for C.
Figure 1: Schematic of the ESMF “sandwich” architecture. 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.

### 3 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.

More information on the ESMF project as a whole is available on the ESMF website, [http://www.esmf.ucar.edu](http://www.esmf.ucar.edu). The website includes release notes and known bugs for each version of the framework, supported platforms, project history, values, and metrics, related projects, the ESMF management structure, and more. The *ESMF User’s Guide* contains build and installation instructions, an overview of the ESMF system and a description of how its classes interrelate (this version of the document corresponds to the last public version of the framework). Also available on the ESMF website is the *ESMF Developer’s Guide* that details ESMF procedures and conventions.

### 4 How to Submit Comments, Bug Reports, and Feature Requests

We welcome input on any aspect of the ESMF project. Send questions and comments to esmf_support@ucar.edu

### 5 Conventions

#### 5.1 Typeface and Diagram Conventions

The following conventions for fonts and capitalization are used in this and other ESMF documents.
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.

### 5.2 Method Name and Argument Conventions

Method names begin with `ESMF_`, followed by the class name, followed by the name of the operation being performed. Each new word is capitalized. Although Fortran interfaces are not case-sensitive, we use case to help parse multi-word names.

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 looks like this:

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

where

* `<ClassName>` is the class name,
* `<Operation>` 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.

### 5.3 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, method listings are split into a number of different sections. For example, there may be separate listings for basic Field methods and overloaded Field methods. 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. Eventually we hope to eliminate this issue and have all methods in a class listed together.
6 The ESMF Application Programming Interface

The ESMF Application Programming Interface (API) is based on the object-oriented programming concept 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 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.

6.1 Standard Methods and Interface Rules

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

- \texttt{ESMF\_<Class>\_Create()} and \texttt{ESMF\_<Class>\_Destroy()}, for creating and destroying classes. The \texttt{ESMF\_<Class>\_Create()} method allocates memory for the class structure itself and for internal variables, and initializes variables where appropriate. It is always written as a Fortran function that returns a derived type instance of the class.

- \texttt{ESMF\_<Class>\_Set()} and \texttt{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 \texttt{ESMF\_<Class>\_Set<Something>()} and \texttt{ESMF\_<Class>\_Get<Something>()} interfaces.

- \texttt{ESMF\_<Class>\_Add()} and \texttt{ESMF\_<Class>\_Remove()} for manipulating items that can be appended or inserted into a list of like items within a class. For example, the \texttt{ESMF\_StateAdd()} method adds another Field to the list of Fields contained in the State class.

- \texttt{ESMF\_<Class>\_Print()}, for printing the contents of a class to standard out. This method is mainly intended for debugging.

- \texttt{ESMF\_<Class>\_ReadRestart()} and \texttt{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.

- \texttt{ESMF\_<Class>\_Validate()}, for determining whether a class is internally consistent. For example, \texttt{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’)
```
6.2 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 ESMF, including Fields, FieldBundles, Arrays, ArrayBundles, 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. Examples of shallow classes are Times, TimeIntervals, and ArraySpecs. 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 a Time, is stored in a deep object such as a Clock. The Clock then carries a copy of the Time in persistent memory. The Time is deallocated with the ESMF_ClockDestroy() call.

See Section 10, 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, see the ESMF Implementation Report.

6.3 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.

6.4 The ESMF Data Hierarchy

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

- **ArrayBundle**: An ArrayBundle is a collection of Arrays, not necessarily distributed in the same manner. It is useful for performing collective data operations and communications.

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

- **FieldBundle**: A FieldBundle is a collection of Fields discretized on the same grid. The staggering of data points may be different for different Fields within a FieldBundle. Like the ArrayBundle, it is useful for performing collective data operations and communications.

- **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 Arrays, ArrayBundles, Fields, FieldBundles, or other States.
**Component** A Component is a 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 are contained within more complex ones.

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

### 6.5 ESMF Spatial Classes

Like the hierarchy of model data classes, ranging from the simple to the complex, ESMF is organized around an hierarchy of classes that represent different spaces associated with a computation. Each of these spaces can be manipulated, in order to give the user control over how a computation is executed. For Earth system models, this hierarchy starts with the address space associated with the computer and extends to the physical region described by the application.

The main spatial classes in ESMF, from 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, single and multi-core systems. Its primary purpose is resource allocation and management. Each Component runs in its own VM, using the resources it defines. The elements of a VM are **Persistent Execution Threads**, or PETS, that are executing in **Virtual Address Spaces**, or VASs. A simple case is one in which every PET is associated with a single MPI process. In this case every PET is executing in its own private VAS. 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 resources associated with the PETS they’ve received.

- **DELayout** A DELayout represents a data decomposition (we also refer to this as a distribution). Its basic elements are **Decomposition Elements**, or DEs. A DELayout associates a set of DEs with the PETS in a VM. 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 also be defined if an application requires it.

- **DistGrid** A DistGrid represents the index space associated with a grid. It is a useful abstraction because often a full specification of grid coordinates is not necessary to define data communication patterns. The DistGrid contains information about the sequence and connectivity of data points, which is sufficient information for many operations. Arrays are defined on DistGrids.

- **Array** An Array defines how the index space described in the DistGrid is associated with the VAS of each PET. This association considers the type, kind and rank of the indexed data. Fields are defined on Arrays.

- **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. Grids in ESMF are comprised of DistGrids plus additional coordinate information.

- **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 data at multiple times. Fields also keep track of the stagger location of a Field data point within its associated Grid cell.

### 6.6 ESMF Maps

In order to define how the index spaces of the spatial classes relate to each other, we require either implicit rules (in which case the relationship between spaces is defined by default), or special Map arrays that allow the user to
specify the desired association. The form of the specification is usually that the position of the array element carries information about the first object, and the value of the array element carries information about the second object. ESMF includes a distGridToArrayMap, a gridToFieldMap, a distGridToGridMap, and others.

6.7 ESMF Specification Classes

It can be useful to make small packets of descriptive parameters. ESMF has one of these:

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

6.8 ESMF Utility Classes

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

- **Attributes**, for storing metadata about Fields, FieldBundles, States, and other classes.
- **TimeMgr**, for calendar, time, 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.

7 Overall Rules and Behavior

7.1 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 deallocate the buffer when all use of it is complete.

Classes such as Fields, FieldBundles, and States may have Arrays, Fields, Grids and FieldBundles 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.

7.2 Equality and Copying Objects

The equal sign operator in ESMF does not generate any special behavior on the part of the framework. If the user decides to set one object equal to another, the internal contents will simply be copied. That means that if there is a pointer within the object being copied, the pointer will be replicated and the data pointed to will be referenced by the object copy. As a matter of style and safety, users should try to avoid exploiting such implicit behavior. A preferable approach is to use a class creation or duplication method. Unfortunately, not all classes have duplication methods yet.

8 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., intertwined functions, very large, highly performance-tuned, resource limitations) there is little motivation to fully restructure. The superstructure can usually 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 maximizing code reuse and minimizing maintenance costs is a goal. There may be a specific need for functionality or the component writer may be starting from scratch. The calendar management utility is a popular place to start.

8.1 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. 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.

1. Decide how to organize the application as discrete Gridded and Coupler Components. This might involve reorganizing 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. This is done by first wrapping model data in either ESMF Arrays or Fields. Arrays are simpler to create and use than Fields, but carry less information and have a more limited range of operations. These Arrays and Fields are then added to Import and Export States. They may be packed into ArrayBundles or FieldBundles first, for more efficient communications. Metadata describing the model data can also be added. At the end of this step, the data to be transferred between components will be in a compact and largely self-describing form.

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 Global Options and Parameters

9.1 Flags

9.1.1 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.
9.1.2  ESMF_BlockingFlag

DESCRIPTION:
Indicates method blocking behavior and PET synchronization for communication calls as well as Component initialize, run and finalize methods.

For communication calls the *ESMF_BLOCKING* and *ESMF_NONBLOCKING* modes provide behavior that is practically identical to the blocking and non-blocking communication calls familiar to MPI users.

The details of how the blocking mode setting affects Component methods are more complex as a consequence of the fact that ESMF Components can be run in threaded or non-threaded mode. However, in the default, non-threaded case, where an ESMF application runs as a pure MPI program, most of the complexity is removed.

See the VM item in [6.5] for an explanation of the PET and VAS concepts used in the following descriptions.

Valid values are:

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

**Component calls:** The called method will block until all PETs of the VM have completed the operation.

For a non-threaded, pure MPI component the behavior is identical to calling a barrier before returning from the method. Generally this kind of rigid synchronization is not the desirable mode of operation for an MPI application, but may be useful for application debugging. In the opposite case, where all PETs of the component are running as threads in shared memory, i.e. in a single VAS, strict synchronization of all PETs is required to prevent race conditions.

**ESMF_VASBLOCKING**  
**Communication calls:** Not available for communication calls.

**Component calls:** The called method will block each PET until all operations in the PET-local VAS have completed.

This mode is a combination of *ESMF_BLOCKING* and *ESMF_NONBLOCKING* modes. It provides a default setting that leads to the typically desirable behavior for pure MPI components as well as those that share address spaces between PETs.

For a non-threaded, pure MPI component each PET returns independent of the other PETs. This is generally the expected behavior in the pure MPI case where calling into a component method is practically identical to a subroutine call without extra synchronization between the processes.

In the case where some PETs of the component are running as threads in shared memory *ESMF_VASBLOCKING* becomes identical to *ESMF_BLOCKING* within thread groups, to prevent race conditions, while there is no synchronization between the thread groups.

**ESMF_NONBLOCKING**  
**Communication calls:** The called method will not block but returns immediately after initiating the requested operation. It is unsafe to modify or use participating local data before all local operations have completed. Use the `ESMF_VMCommWait()` or `ESMF_VMCommQueueWait()` method to block the local PET until local data access is safe again.

**Component calls:** The behavior of this mode is fundamentally different for threaded and non-threaded components, independent on whether the components use shared memory or not. The *ESMF_NONBLOCKING* mode is the most complex mode for calling component methods and should only be used if the extra control, described below, is absolutely necessary.

For non-threaded components (the ESMF default) calling a component method with *ESMF_NONBLOCKING* is identical to calling it with **ESMF_VASBLOCKING**. However, different than for **ESMF_VASBLOCKING**, a call to `ESMF_GridCompWait()` or `ESMF_CplCompWait()` is required in order to deallocate memory internally allocated for the **ESMF_NONBLOCKING** mode.

For threaded components the calling PETs of the parent component will not be blocked and return immediately after initiating the requested child component method. In this scenario parent and child components will run
concurrently in identical VASs. This is the most complex mode of operation. It is unsafe to modify or use VAS local data that may be accessed by concurrently running components until the child component method has completed. Use the appropriate ESMF_GridCompWait() or ESMF_CplCompWait() method to block the local parent PET until the child component method has completed in the local VAS.

9.1.3 ESMF_ContextFlag

DESCRIPTION:
Indicates the type of VM context in which a component is executing.
Valid values are:

ESMF_CHILD_IN_NEW_VM The component is running in its own, separate VM context. Resources are inherited from the parent but can be arranged to fit the component’s requirements.

ESMF_CHILD_IN_PARENT_VM The component uses the parent’s VM for resource management. Compared to components that use their own VM context components that run in the parent’s VM context are more lightweight with respect to the overhead of calling into their initialize, run and finalize methods. Furthermore, VM-specific properties remain unchanged when going from the parent component to the child component. These properties include the MPI communicator, the number of PETs, the PET labeling, communication attributes, threading-level.

9.1.4 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.

9.1.5 ESMF_DefaultFlag

DESCRIPTION:
Indicates whether to use defaults or not.
Valid values are:

ESMF_USE_DEFAULTS Use default values where possible.
ESMF_NO_DEFAULTS Don’t use any default values.

9.1.6 ESMF_DecompFlag

DESCRIPTION:
Indicates how DistGrid elements are decomposed over DEs.
Valid values are:

ESMF_DECOMP_CYCLIC Decompose elements cyclically across DEs.
ESMF_DECOMP_DEFAULT Use default decomposition behavior. Currently equal to ESMF_DECOMP_HOMOGEN.
ESMF_DECOMP_HOMOGEN Decompose elements as homogenously as possible across DEs. The maximum difference in number of elements per DE is 1, with the extra elements on the lower DEs.
ESMF_DECOMP_RESTFIRST Divide elements over DEs. Assign the rest of this division to the first DE.
ESMF_DECOMP_RESTLAST Divide elements over DEs. Assign the rest of this division to the last DE.
9.1.7 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.

9.1.8 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.

9.1.9 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.

9.1.10 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.

9.1.11 ESMF_RegionFlag

**DESCRIPTION:**
Specifies various regions in the data layout of an Array or Field object.

Valid values are:

**ESMF_REGION_TOTAL** Total allocated memory.

**ESMF_REGION_SELECT** Region of operation-specific elements.

**ESMF_REGION_EMPTY** The empty region contains no elements.
9.1.12 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.

9.1.13 ESMF_Status
DESCRIPTION:
This is a general object status flag used throughout the framework.
Valid values are:
ESMF_STATUS_UNINIT Object is uninitialized.
ESMF_STATUS_READY Object is ready for use.
ESMF_STATUS_UNALLOCATED Object has not yet been allocated.
ESMF_STATUS_ALLOCATED Object has been allocated.
ESMF_STATUS_BUSY Object is not able to respond.
ESMF_STATUS_INVALID Object is invalid.

9.1.14 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.

9.2 Parameters
9.2.1 ESMF_TypeKind
DESCRIPTION:
Supported ESMF type and kind combinations. This is an ESMF derived type used for arguments to subroutines and functions that specify or query a data precision and type. These values cannot be used when declaring variables; see the next section on Fortran Kinds for that.
Valid values are:
ESMF_TYPEKIND_I1 1 byte integer.
ESMF_TYPEKIND_I2 2 byte integer.
ESMF_TYPEKIND_I4 4 byte integer.
ESMF_TYPEKIND_I8 8 byte integer.
ESMF_TYPEKIND_R4 4 byte real.
ESMF_TYPEKIND_R8 8 byte real.


9.2.2 Fortran Kinds

DESCRIPTION:
These are integer parameters of the proper type to be used when declaring variables with a specific precision in Fortran syntax. For example:

    integer(ESMF_KIND_I4) :: myintegervariable
    real(ESMF_KIND_R4) :: myrealvariable

The Fortran 90 standard does not mandate what numeric values correspond to actual number of bytes allocated for the various kinds, so these are defined by ESMF to be correct across the different supported Fortran 90 compilers. Note that not all compilers support every kind listed below; in particular 1 and 2 byte integers can be problematic.

Valid values are:

- **ESMF_KIND_I1**: 1 byte integer.
- **ESMF_KIND_I2**: 2 byte integer.
- **ESMF_KIND_I4**: 4 byte integer.
- **ESMF_KIND_I8**: 8 byte integer.
- **ESMF_KIND_R4**: 4 byte real.
- **ESMF_KIND_R8**: 8 byte real.
- **ESMF_KIND_C8**: 8 byte character.
- **ESMF_KIND_C16**: 16 byte character.

10 Overall Design and Implementation Notes

1. **Deep and shallow classes.** The deep and shallow classes described in Section 6.2 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, which holds a small set of system-wide capabilities.
Part II
Superstructure
11 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 hierarchical 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.

<table>
<thead>
<tr>
<th>Key Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular, component-based architecture.</td>
</tr>
<tr>
<td>Hierarchical assembly of components into applications.</td>
</tr>
<tr>
<td>Use of components in multiple contexts without modification.</td>
</tr>
<tr>
<td>Sequential or concurrent component execution.</td>
</tr>
<tr>
<td>Single program, multiple datastream (SPMD) applications for maximum portability and reconfigurability.</td>
</tr>
<tr>
<td>Multiple program, multiple datastream (MPMD) option for flexibility.</td>
</tr>
</tbody>
</table>

11.1 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 myOceanInit might be associated with the standard initialize routine of an ESMF Gridded Component named “myOcean” 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, FieldBundles, Arrays, ArrayBundles, and other States. A Component is associated with two States, an Import State and an Export State. Its Import State holds the data that it receives from other Components. Its Export State contains data that it can make available to other 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.

The parent Gridded Component is responsible for creating the child Gridded Components that are exchanging data, for 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.

11.2 Hierarchical Creation of Components

Components are allocated computational resources in the form of Persistent Execution Threads, or PETs. 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. Components are created hierarchically, with parent Components creating child Components and allocating some or all of their PETs to each one. By default ESMF creates a new VM for each child Component, which allows Components to tailor their VM resources to match their needs. In some cases a child may want to share its parent’s VM - ESMF supports this too.
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.

11.3 Sequential and Concurrent Execution of Components

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**.

Figure 4 illustrates a typical configuration for a simple coupled sequential application, and Figure 5 shows a possible configuration for the same application running in a concurrent mode.

Parent Components can select if and when to wait for concurrently executing child Components, synchronizing only when required.

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.

11.4 Intra-Component Communication

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 redistribute data 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.

11.5 Data Distribution and Scoping in Components

The scope of distributed objects is the VM of the currently executing Component. For this reason, all PETs in the current VM must make the same distributed object creation calls. When a Coupler Component running on a super-set of a Gridded Component’s PETs needs to make communication calls involving objects created by the Gridded Component, an ESMF-supplied function called `ESMF_StateReconcile()` creates proxy objects for those PETs that had no previous information about the distributed objects. Proxy objects contain no local data but can be used in communication calls (such as regrid or redistribute) to describe the remote source for data being moved to the current PET, or to describe the remote destination for data being moved from the local PET. Figure 6 is a simple schematic that shows the sequence of events in a reconcile call.

11.6 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.
Figure 4: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running sequentially with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The AppDriver, Coupler, and all Gridded Components are distributed over nine PETs.
Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The AppDriver, Coupler, and top-level “Hurricane Model” Gridded Component are distributed over nine PETs. The “Atmosphere” Gridded Component is distributed over three PETs and the “Ocean” Gridded Component is distributed over six PETs.
Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.
11.7 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.

![UML diagram showing relationships among ESMF superstructure classes](image)

12 Application Driver and Required ESMF Methods

12.1 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 sequential execution model every Component executes on all PETs, with each Component completing execution before the next Component begins. 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 grid 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 multiple 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.

**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 currently has Fortran interfaces for all public functions. Some functions also have C interfaces, and the number of these is expected to increase over time.

Number of Executables  The simplest way to run an application is to run the same executable program on all PETs. Different Components can still be run on mutually exclusive PETs by using branching (e.g., if this is PET 1, 2, or 3, run Component A, if it is PET 4, 5, or 6 run Component B). This is a SPMD model, Single Program Multiple Data.

The alternative is to start a different executable program on different PETs. 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 communications established. ESMF currently has some support for MPMD: different Components can run as separate executables, but the Coupler that transfers data between the Components must still run on the union of their PETs.

12.2 Application Driver and Required ESMF Methods Options

12.2.1 ESMF_TerminationFlag

DESCRIPTION: The ESMF_TerminationFlag determines how an ESMF application is shut down. Valid values are:

ESMF_ABORT  Global abort of the ESMF application. There is no guarantee that all PETs will shut down cleanly during an abort. However, all attempts are made to prevent the application from hanging and the LogErr of at least one PET will be completely flushed during the abort. This option should only be used if a condition is detected that prevents normal continuation or termination of the application. Typical conditions that warrant the use of ESMF_ABORT are those that occur on a per PET basis where other PETs may be blocked in communication calls, unable to reach the normal termination point.

ESMF_FINALIZE  Normal termination of the ESMF application. Wait for all PETs of the global VM to reach ESMF_Finalize() before termination. This is the clean way of terminating an application. MPI_Finalize() will be called in case of MPI applications.

ESMF_KEEPMPI  Same as ESMF_FINALIZE but MPI_Finalize() will not be called. It is the user code’s responsibility to shut down MPI cleanly.

12.3 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-Component of another, larger application if desired. When a simple 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|concur>_<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. It 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.
Examples of concurrent component execution can be found in the system tests that are bundled with the ESMF distribution.

---
EXAMPLE: This is an AppDriver.F90 file for a sequential ESMF application.
---

The ChangeMe.F90 file that’s included below 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. This file is in the same directory as the AppDriver.F90 file.

```fortran
#include "ChangeMe.F90"

program ESMF_AppDriver
#define ESMF_METHOD "program ESMF_AppDriver"
#include "ESMF.h"

! 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

Define local variables

! Components and States
type(ESMF_GridComp) :: compGridded
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_TimeInterval) :: timeStep
type(ESMF_Time) :: startTime

! Variables related to the Grid
integer :: i_max, j_max

! Return codes for error checks
integer :: rc, localrc
```
Initialize ESMF. Note that an output Log is created by default.

```fortran
call ESMF_Initialize(defaultCalendar=ESMF_CAL_GREGORIAN, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_LogWrite("ESMF AppDriver start", ESMF_LOG_INFO)
```

Create and load a configuration file.  
The USER_CONFIG_FILE is set to sample.rc in the ChangeMe.F90 file.  
The sample.rc file is also included in the directory with the AppDriver.F90 file.

```fortran
config = ESMF_ConfigCreate(rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_ConfigLoadFile(config, USER_CONFIG_FILE, rc = localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)
```

Get configuration information.

A configuration file like sample.rc might include:
- size and coordinate information needed to create the default Grid.
- the default start time, stop time, and running intervals
  for the main time loop.

```fortran
call ESMF_ConfigGetAttribute(config, i_max, 'I Counts:', default=10, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_ConfigGetAttribute(config, j_max, 'J Counts:', default=40, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)
```

Create the top Gridded Component.

```fortran
compGridded = ESMF_GridCompCreate(name="ESMF Gridded Component", rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)
```
call ESMF_LogWrite("Component Create finished", ESMF_LOG_INFO)

Register the set services method for the top Gridded Component.

call ESMF_GridCompSetServices(compGridded, SetServices, rc)
if (ESMF_LogMsgFoundError(rc, "Registration failed", rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

Create and initialize a Clock.

call ESMF_TimeIntervalSet(timeStep, s=2, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_TimeSet(startTime, yy=2004, mm=9, dd=25, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_TimeSet(stopTime, yy=2004, mm=9, dd=26, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

clock = ESMF_ClockCreate("Application Clock", timeStep, startTime, &
stopTime, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

Create and initialize a Grid.

The default lower indices for the Grid are (/1,1/).
The upper indices for the Grid are read in from the sample.rc file,
where they are set to (/10,40/). This means a Grid will be
created with 10 grid cells in the x direction and 40 grid cells in the
y direction. The Grid section in the Reference Manual shows how to set
coordinates.

grid = ESMF_GridCreateShapeTile(maxIndex=(/i_max, j_max/), &
name="source grid", rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &
ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

! Attach the grid to the Component
call ESMF_GridCompSet(compGridded, grid=grid, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

Create and initialize a State to use for both import and export. In a real code, separate import and export States would normally be created.

defaultstate = ESMF_StateCreate("Default State", rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

Call the initialize, run, and finalize methods of the top component. When the initialize method of the top component is called, it will in turn call the initialize methods of all its child components, they will initialize their children, and so on. The same is true of the run and finalize methods.

call ESMF_GridCompInitialize(compGridded, defaultstate, defaultstate, &clock, rc=localrc)
if (ESMF_LogMsgFoundError(rc, "Initialize failed", rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_GridCompRun(compGridded, defaultstate, defaultstate, & clock, rc=localrc)
if (ESMF_LogMsgFoundError(rc, "Run failed", rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_GridCompFinalize(compGridded, defaultstate, defaultstate, &clock, rc=localrc)
if (ESMF_LogMsgFoundError(rc, "Finalize failed", rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

Destroy objects.

call ESMF_ClockDestroy(clock, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_StateDestroy(defaultstate, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

call ESMF_GridCompDestroy(compGridded, rc=localrc)
if (ESMF_LogMsgFoundError(localrc, ESMF_ERR_PASSTHRU, &ESMF_CONTEXT, rcToReturn=rc)) &
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)
call ESMF_Finalize(rc=localrc, terminationflag=ESMF_ABORT)

Finalize and clean up.

call ESMF_Finalize()

end program ESMF_AppDriver

12.4 Required ESMF Methods

12.4.1 ESMF_Initialize - Initialize the ESMF

INTERFACE:

subroutine ESMF_Initialize(defaultConfigFileName, defaultCalendar, &
defaultLogFileName, defaultLogType, mpiCommunicator, vm, rc)

ARGUMENTS:

character(len=*), intent(in), optional :: defaultConfigFileName
type(ESMF_CalendarType), intent(in), optional :: defaultCalendar
character(len=*), intent(in), optional :: defaultLogFileName
type(ESMF_LogType), intent(in), optional :: defaultLogType
integer, intent(in), optional :: mpiCommunicator
integer, 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. The method contains a barrier before returning, ensuring that all processes made it successfully through initialization.

Typically ESMF_Initialize() will call MPI_Init() internally unless MPI has been initialized by the user code before initializing the framework. If the MPI initialization is left to ESMF_Initialize() it inherits all of the MPI implementation dependent limitations of what may or may not be done before MPI_Init(). For instance, it is unsafe for some MPI implementations, such as MPICH, to do IO before the MPI environment is initialized. Please consult the documentation of your MPI implementation for details.

Note that when using MPICH as the MPI library, ESMF needs to use the application command line arguments for MPI_Init(). However, ESMF acquires these arguments internally and the user does not need to worry about providing them. Also, note that ESMF does not alter the command line arguments, so that if the user obtains them they will be as specified on the command line (including those which MPICH would normally strip out).

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. See section 30.2.1 for a list of valid options. 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.

[defaultLogType] Sets the default Log Type to be used by ESMF Log Manager. See section 36.2.3 for a list of valid options. If not specified, defaults to ESMF_LOG_MULTI.

[mpiCommunicator] MPI communicator defining the group of processes on which the ESMF application is running. If not specified, defaults to MPI_COMM_WORLD.

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

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

12.4.2 ESMF_Finalize - Clean up and close the ESMF

INTERFACE:

    subroutine ESMF_Finalize(terminationflag, rc)

ARGUMENTS:

    type(ESMF_TerminationFlag), intent(in), optional :: terminationflag
    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 optional argument terminationflag may be used to indicate the mode of termination.

The arguments are:

[terminationflag] Specify mode of termination. The default is ESMF_FINAL which waits for all PETs of the global VM to reach ESMF_Finalize() before termination. See section 12.2.1 for a complete list and description of valid flags.

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

12.4.3 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. Both arguments are required and must not be declared as optional. If an intent is specified in the interface it must be intent(inout) for the first and intent(out) for the second argument. 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>CompSetInternalSta 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 must call the component-specific set services routine to associate ESMF-standard methods to user-code methods, and to create the VM in which this component will run. After set services has been called, the framework now will be able to call the component’s initialize, run, and finalize routines as required.

12.4.4 User-Code Initialize, Run, and Finalize Methods

User-code initialize, run, and finalize routines must be provided for each component. See Sections 13.6 and 14.5 for the prescribed interfaces and examples of how to set these up.

13 GridComp Class

13.1 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. Gridded Components may also represent individual processes, such as radiation or chemistry. It’s up to the application writer to decide how deeply to “componentize.”

Earth system software components tend to share a number of basic features. Most ingest and produce 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 tailored for Earth system modeling and yet is still flexible enough to represent a variety of domains.

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. 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. Data is passed between Gridded Components using an intermediary Coupler Component, described in Section 14.1.

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 that represents a physical domain or performs 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.

13.2 GridComp Options

13.2.1 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.
- **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.

### 13.3 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.

This section shows how to create Gridded Components. For demonstrations of the use of Gridded Components, see the system tests that are bundled with the ESMF software distribution. These can be found in the directory esmf/src/system_tests.

#### 13.3.1 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. Both arguments are required and must not be declared as optional. If an intent is specified in the interface it must be intent(inout) for the first and intent(out) for the second argument.

The set services routine 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.

```fortran
module ESMF_GriddedCompEx

! ESMF Framework module
use ESMF_Mod
implicit none
public GComp_SetServices
contains

subroutine GComp_SetServices(comp, rc)
  type(ESMF_GridComp) :: comp
  integer, intent(out) :: 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

end subroutine GComp_SetServices
end module ESMF_GriddedCompEx
```
! to be passed in to the routines above:
! call ESMF_GridCompSetData(comp, mydatablock, rc)

rc = ESMF_SUCCESS

end subroutine

13.3.2 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, intent(out) :: rc

print *, "Gridded Comp Init starting"
!
print *, "Gridded Comp Init returning"

rc = ESMF_SUCCESS
end subroutine

13.3.3 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, intent(out) :: rc

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

! This is where the model specific computation goes.
! Fill export state here using ESMF_StateAdd(), etc

print *, "Gridded Comp Run returning"
rc = ESMF_SUCCESS
end subroutine GComp_Run

13.3.4 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, intent(out) :: 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
data space can then be referenced for each ensemble member’s calculations.

See the code below for a simple example of using this capability.

! ESMF Framework module
use ESMF_Mod
implicit none

type(ESMF_GridComp) :: compl
integer :: rc, finalrc

! Internal State Variables
type testData
sequence
  integer :: testValue
  real :: testScaling
end type
type dataWrapper
sequence
  type(testData), pointer :: p
end type
type (dataWrapper) :: wrap1, wrap2
type(testData), target :: data1, data2

finalrc = ESMF_SUCCESS
!-------------------------------------------------- -----------------------
call ESMF_Initialize(rc=rc)
if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
!-------------------------------------------------- -----------------------
! ! Creation of a Component
compl = ESMF_GridCompCreate(name="test", rc=rc)
if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
!-------------------------------------------------- -----------------------
! This could be called, for example, during a routine’s initialize phase.
! ! Set Internal State
data1%testValue = 4567
data1%testScaling = 0.5
wrap1%p => data1
call ESMF_GridCompSetInternalState(compl, wrap1, rc)
if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
!-------------------------------------------------- -----------------------
! And this could be called, for example, during a routine’s run phase.
! ! Get Internal State
! ! note that we do not assign the pointer inside wrap2 - this call
! ! does that.
call ESMF_GridCompGetInternalState(comp1, wrap2, rc)
if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

data2 = wrap2%p
if ((data2%testValue .ne. 4567) .or. (data2%testScaling .ne. 0.5)) then
  print *, "did not get same values back"
  finalrc = ESMF_FAILURE
else
  print *, "got same values back from GetInternalState as original"
endif

13.4 Restrictions and Future Work

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

2. **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.

13.5 Class API: Basic GridComp Methods

13.5.1 ESMF_GridCompCreate - Create a Gridded Component

INTERFACE:

recursive function ESMF_GridCompCreate(name, gridcomptype, grid, &
  config, configFile, clock, petList, contextflag, parentVm, rc)

RETURN VALUE:

type(ESMF_GridComp) :: ESMF_GridCompCreate

ARGUMENTS:

!external :: services
character(len=*), intent(in), optional :: name

  type(ESMF_GridCompType), intent(in), optional :: gridcomptype
type(ESMF_Grid), intent(inout), optional :: grid

  type(ESMF_Config), intent(inout), optional :: config
character(len=*), intent(in), optional :: configFile

type(ESMF_Clock), intent(inout), optional :: clock

  integer, intent(in), optional :: petList(:)
type(ESMF_ContextFlag), intent(in), optional :: contextflag

type(ESMF_VM), intent(inout), optional :: parentVm

  integer, intent(out), optional :: rc

DESCRIPTION:

Create an ESMF_GridComp object.
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 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 parent PETs given to the created child component by the parent component. If petList is not specified all of the parent PETs will be given to the child component. The order of PETs in petList determines how the child local PETs refer back to the parent PETs.

**[contextflag]** Specify the component’s VM context. The default context is ESMF_CHILD_IN_NEW_VM. See section 9.1.3 for a complete list of valid flags.

**[parentVm]** ESMF_VM object for the current component. This will become the parent ESMF_VM for the newly created ESMF_GridComp object. By default the current VM is determined automatically.

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

### 13.5.2 ESMF_GridCompDestroy - Release resources for a GridComp

**INTERFACE:**

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

**ARGUMENTS:**

```fortran
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.
13.5.3 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(inout), 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 code for an ESMF_GridComp. The arguments are:

gridcomp The ESMF_GridComp to call finalize routine for.

[importState] ESMF_State containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] ESMF_State containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[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. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[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] Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

13.5.4 ESMF_GridCompGet - Query a GridComp for information

INTERFACE:

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

ARGUMENTS:
type (ESMF_GridComp), intent(inout) :: gridcomp
character(len=*), intent(out), optional :: name
type (ESMF_GridCompType), intent(out), optional :: gridcomptype
type (ESMF_Grid), intent(out), optional :: grid
character(len=*), intent(out), optional :: config
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
type (ESMF_ContextFlag), intent(out), optional :: contextflag
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.
[contextflag] Return the ESMF_ContextFlag for this ESMF_GridComp. See section 9.1.3 for a complete list of valid flags.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

13.5.5 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(inout), 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. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

**[exportState]** ESMF_State containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

**[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. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

**[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, accommodating 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]** Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

---

### 13.5.6 ESMF_GridCompPrint - Print the contents of a GridComp

**INTERFACE:**

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

**ARGUMENTS:**

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

**DESCRIPTION:**

Prints information about an ESMF_GridComp to stdout.

Note: Many ESMF_<classname>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call `flush(6)` as a workaround until this issue is fixed in a future release.

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.
13.5.7 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(inout), 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. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] ESMF_State containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[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. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[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, accommodating 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] Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

13.5.8 ESMF_GridCompSet - Set or reset information about the GridComp

INTERFACE:

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Intent</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>type(ESMF_GridComp),</td>
<td>intent(inout)</td>
<td></td>
</tr>
<tr>
<td>character(len=*),</td>
<td>intent(in),</td>
<td>optional</td>
</tr>
<tr>
<td>type(ESMF_GridCompType),</td>
<td>intent(in),</td>
<td>optional</td>
</tr>
<tr>
<td>type(ESMF_Grid),</td>
<td>intent(inout), optional</td>
<td></td>
</tr>
<tr>
<td>type(ESMF_Config),</td>
<td>intent(inout), optional</td>
<td></td>
</tr>
<tr>
<td>character(len=*),</td>
<td>intent(in),</td>
<td>optional</td>
</tr>
<tr>
<td>type(ESMF_Clock),</td>
<td>intent(inout), optional</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>intent(out),</td>
<td>optional</td>
</tr>
</tbody>
</table>

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. Superceded 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.

13.5.9   ESMF_GridCompValidate - Check validity of a GridComp

INTERFACE:

subroutine ESMF_GridCompValidate(gridcomp, options, rc)

ARGUMENTS:

<table>
<thead>
<tr>
<th>Type</th>
<th>Intent</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>type(ESMF_GridComp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>character (len = *),</td>
<td>intent(in),</td>
<td>optional</td>
</tr>
<tr>
<td>integer</td>
<td>intent(out),</td>
<td>optional</td>
</tr>
</tbody>
</table>

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.
13.5.10 ESMF_GridCompWait - Wait for a GridComp to return

INTERFACE:

    subroutine ESMF_GridCompWait(gridcomp, blockingFlag, rc)

ARGUMENTS:

    type(ESMF_GridComp), intent(inout) :: gridcomp
    type (ESMF_BlockingFlag), intent(in), optional :: blockingFlag
    integer, intent(out), optional :: rc

DESCRIPTION:

When executing asynchronously, wait for an ESMF_GridComp to return.

The arguments are:

gridcomp  ESMF_GridComp to wait for.

[blockingFlag]  Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

13.5.11 ESMF_GridCompIsPetLocal - Inquire if this component is to execute on the calling PET.

INTERFACE:

    recursive function ESMF_GridCompIsPetLocal(gridcomp, rc)

RETURN VALUE:

    logical :: ESMF_GridCompIsPetLocal

ARGUMENTS:

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

DESCRIPTION:

Inquire if this ESMF_GridComp object is to execute on the calling PET.

The return value is .true. if the component is to execute on the calling PET, .false. otherwise.

The arguments are:

gridcomp  ESMF_GridComp queried.

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

13.6 Class API: SetServices and Related Methods

13.6.1 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) :: 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. Note that the dataPointer argument needs to be a derived type which contains only a pointer of the type of the data block defined by the user. When making this call the pointer needs to be unassociated. When the call returns the pointer will now reference the original data block which was set during the previous call to ESMF_GridCompSetInternalState.

The arguments are:

gridcomp  An ESMF_GridComp object.

dataPointer  A derived type, containing only an unassociated pointer to the private data block. The framework will fill in the pointer. 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. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

13.6.2 ESMF_GridCompSetEntryPoint - Set name of GridComp subroutines

INTERFACE:

    subroutine ESMF_GridCompSetEntryPoint(comp, subroutineType, &
                                          subroutineName, phase, rc)

ARGUMENTS:

    type(ESMF_GridComp), intent(inout) :: comp
classic(len=*), intent(in) :: subroutineType
subroutine, intent(in) :: subroutineName
integer, intent(in) :: phase
integer, intent(out) :: rc

DESCRIPTION:

Intended to be called by an ESMF_GridComp during the registration process. An ESMF_GridComp calls ESMF_GridCompSetEntryPoint for each of the predefined init, run, and finalize routines, to associate 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:

comp  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 user-supplied 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 multiple phases, the phase number which corresponds to this subroutine name. For single phase subroutines 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.

crc  Return code; equals ESMF_SUCCESS if there are no errors. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

The user-supplied routine must conform to the following interface:

INTERFACE:

    interface subroutine subroutineName (comp, importState, exportState, clock, rc)
    type(ESMF_GridComp) :: comp
    type(ESMF_State) :: importState, exportState
    type(ESMF_Clock) :: clock
    integer, intent(out) :: rc
    end subroutine subroutineName
    end interface

13.6.3  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) :: 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.

crc  Return code; equals ESMF_SUCCESS if there are no errors. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.
13.6.4 ESMF_GridCompSetServices - Register GridComp interface routines

**INTERFACE:**

    subroutine ESMF_GridCompSetServices(comp, subroutineName, rc)

**ARGUMENTS:**

    type(ESMF_GridComp) :: comp
    subroutine :: subroutineName
    integer, intent(out) :: 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 user-supplied, 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. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

The user-supplied registration routine must conform to the following interface:

**INTERFACE:**

    interface
        subroutine subroutineName (comp, rc)
            type(ESMF_GridComp) :: comp
            integer, intent(out) :: rc
        end subroutine
    end interface

**DESCRIPTION:**

The subroutine, when called by the framework, must make successive calls to ESMF_GridCompSetEntryPoint to preset callback routines for initialization, run, and finalization for a coupler component.

14 CplComp Class

14.1 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 13.1). 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 transformations or time averaging operations, so users must manage those operations themselves.

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.

The user-written portion 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.

### 14.2 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.

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

#### 14.2.1 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. Both arguments are required and must not be declared as optional. If an intent is specified in the interface it must be intent(inout) for the first and intent(out) for the second argument.

The set services routine 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.

```fortran
! Example Coupler Component
module ESMF_CouplerEx
```

---

2 It is not necessary to create a Coupler Component for each individual data transfer.
14.2.2 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 \texttt{rc} return code should be set if an error occurs, otherwise the value \texttt{ESMF\_SUCCESS} should be returned.

```fortran
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, intent(out) :: rc

  print *, "Coupler Init starting"

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

  rc = ESMF\_SUCCESS

  print *, "Coupler Init returning"

end subroutine CPL_Init
```
14.2.3 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.

```fortran
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, intent(out) :: rc

  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
```

14.2.4 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.

```fortran
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, intent(out) :: rc

  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
```
These routines save the address of an internal, private data block during the execution of a Component’s initialize, run, or finalize code, and retrieve the address back during a different invocation of these routines. One situation where this is useful is in the creation of ensembles of the same component. In this case it can be tricky to distinguish which data belongs to which ensemble member – especially if the ensemble members are executing on the same PETs. Internal states enable the user to create an array of internal data spaces, one for each ensemble member. The correct data space can then be referenced for each ensemble member’s calculations.

See the code below for a simple example of using this capability.

! ESMF Framework module
use ESMF_Mod
implicit none
type(ESMF_GridComp) :: comp1
integer :: rc, finalrc

! Internal State Variables
type testData
sequence
  integer :: testValue
  real :: testScaling
end type
type dataWrapper
sequence
  type(testData), pointer :: p
end type
type (dataWrapper) :: wrap1, wrap2
type(testData), target :: data1, data2

finalrc = ESMF_SUCCESS
!-----------------------------------------------------
call ESMF_Initialize(rc=rc)
if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
!-----------------------------------------------------

! Creation of a Component
comp1 = ESMF_GridCompCreate(name="test", rc=rc)
if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

!-------------------------------------------------- -----------------------
! This could be called, for example, during a routine’s initialize phase.
!
! ! Set Internal State
! data1%testValue = 4567
! data1%testScaling = 0.5
! wrap1%p => data1

  call ESMF_GridCompSetInternalState(comp1, wrap1, rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

!-------------------------------------------------- -----------------------
! And this could be called, for example, during a routine’s run phase.
!
! ! Get Internal State
! ! note that we do not assign the pointer inside wrap2 - this call
! ! does that.
  call ESMF_GridCompGetInternalState(comp1, wrap2, rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

  data2 = wrap2%p
  if ((data2%testValue .ne. 4567) .or. (data2%testScaling .ne. 0.5)) then
    print *, "did not get same values back"
    finalrc = ESMF_FAILURE
  else
    print *, "got same values back from GetInternalState as original"
  endif

14.3 Restrictions and Future Work

1. **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.

2. **No automatic unit conversions.** The ESMF framework does not currently contain tools for performing unit conversions, operations that are fairly standard within Coupler Components.

3. **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.

14.4 Class API: Basic CplComp Methods

14.4.1 ESMF_CplCompCreate - Create a Coupler Component

**INTERFACE:**

```c
recursive function ESMF_CplCompCreate(name, config, configFile, &
  clock, petList, contextflag, parentVm, rc)
```

**RETURN VALUE:**
type(ESMF_CplComp) :: ESMF_CplCompCreate

ARGUMENTS:

character(len=*), intent(in), optional :: name

integer, intent(in), optional :: petList(:)

integer, intent(out), optional :: rc

DESCRIPTION:

Create an ESMF_CplComp object.

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 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 parent PETs given to the created child component by the parent component. If petList is not specified all of the parent PETs will be given to the child component. The order of PETs in petList determines how the child local PETs refer back to the parent PETs.

[contextflag] Specify the component’s VM context. The default context is ESMF_CHILD_IN_NEW_VM. See section 9.1.3 for a complete list of valid flags.

[parentVm] ESMF_VM object for the current component. This will become the parent ESMF_VM for the newly created ESMF_CplComp object. By default the current VM is determined automatically.

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

14.4.2 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.

---

### 14.4.3 ESMF_CplCompFinalize - Call the CplComp’s finalize routine

**INTERFACE:**

```fortran
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(inout), 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. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.
- **[exportState]** ESMF_State containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.
- **[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. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.
- **[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, accommodating 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]** Blocking behavior of this method call. See section §9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.
14.4.4 ESMF_CplCompGet - Query a CplComp for information

INTERFACE:

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

ARGUMENTS:

    type(ESMF_CplComp), intent(inout) :: 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
    type(ESMF_ContextFlag) , intent(out), optional :: contextflag
    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.
- **[contextflag]** Return the ESMF_ContextFlag for this ESMF_CplComp. See section 9.1.3 for a complete list of valid flags.
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.

14.4.5 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(inout), 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. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState]  ESMF_State containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

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. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[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, accommodating 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]  Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

14.4.6  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. Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.

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.
14.4.7 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(inout), 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. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

**[exportState]** ESMF_State containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

**[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. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

**[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]** Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

14.4.8 ESMF_CplCompSet - Set or reset information about the CplComp

INTERFACE:

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

ARGUMENTS:
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. Superceded 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.

14.4.9 ESMF_CplCompValidate – Ensure the CplComp is internally consistent

INTERFACE:

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

ARGUMENTS:

```fortran
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.
14.4.10  ESMF_CplCompWait - Wait for a CplComp to return

INTERFACE:

subroutine ESMF_CplCompWait(cplcomp, blockingFlag, rc)

ARGUMENTS:

  type(ESMF_CplComp), intent(inout) :: cplcomp
  type (ESMF_BlockingFlag), intent(in), optional :: blockingFlag
  integer, intent(out), optional :: rc

DESCRIPTION:

When executing asynchronously, wait for an ESMF_CplComp to return.

The arguments are:

cplcomp  ESMF_CplComp to wait for.

[blockingFlag]  Blocking behavior of this method call. See section 9.1.2 for a list of valid blocking options. Default option is ESMF_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

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

14.4.11  ESMF_CplCompIsPetLocal - Inquire if this component is to execute on the calling PET.

INTERFACE:

recursive function ESMF_CplCompIsPetLocal(cplcomp, rc)

RETURN VALUE:

  logical :: ESMF_CplCompIsPetLocal

ARGUMENTS:

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

DESCRIPTION:

Inquire if this ESMF_CplComp object is to execute on the calling PET.

The return value is .true. if the component is to execute on the calling PET, .false. otherwise.

The arguments are:

cplcomp  ESMF_CplComp queried.

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

14.5  Class API: SetServices and Related Methods

14.5.1  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) :: 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. Note that the dataPointer argument needs to be a derived type which contains only a pointer of the type of the data block defined by the user. When making this call the pointer needs to be unassociated. When the call returns the pointer will now reference the original data block which was set during the previous call to ESMF_CplCompSetInternalState.

The arguments are:

    cplcomp  An ESMF_CplComp object.
    dataPointer  A derived type, containing only an unassociated pointer to the private data block. The framework will fill in the pointer. 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. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

14.5.2 ESMF_CplCompSetEntryPoint - Set name of CplComp subroutines

INTERFACE:

    subroutine ESMF_CplCompSetEntryPoint(comp, subroutineType, subroutineName, phase, rc)

ARGUMENTS:

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

DESCRIPTION:

Intended to be called by an ESMF_CplComp during the registration process. An ESMF_CplComp calls ESMF_CplCompSetEntryPoint for each of the predefined init, run, and finalize routines, to associate 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:

    comp  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 user-supplied 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 multiple 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. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

The user-supplied routine must conform to the following interface:

INTERFACE:

```fortran
interface subroutine subroutineName (comp, importState, exportState, clock, rc)
  type(ESMF_CplComp) :: comp
  type(ESMF_State) :: importState, exportState
  type(ESMF_Clock) :: clock
  integer, intent(out) :: rc
end subroutine
end interface
```

14.5.3  ESMF_CplCompSetInternalState - Set private data block pointer

INTERFACE:

```fortran
subroutine ESMF_CplCompSetInternalState(cplcomp, dataPointer, rc)
ARGUMENTS:
  type(ESMF_CplComp), intent(inout) :: cplcomp
  type(any), pointer, intent(in) :: dataPointer
  integer, intent(out) :: 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. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.
```
14.5.4 ESMF_CplCompSetServices - Register CplComp interface routines

INTERFACE:

    subroutine ESMF_CplCompSetServices(comp, subroutineName, rc)

ARGUMENTS:

    type(ESMF_CplComp), intent(inout) :: comp
    subroutine, intent(in) :: subroutineName
    integer, intent(out) :: 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 user-supplied, 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 user-supplied cplcomp ESMF_CplCompSetServices routine. 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. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

The user-supplied registration routine must conform to the following interface:

INTERFACE:

    interface
        subroutine subroutineName (comp, rc)
        type(ESMF_CplComp) :: comp
        integer, intent(out) :: rc
        end subroutine
    end interface

DESCRIPTION:

The subroutine, when called by the framework, must make successive calls to ESMF_CplCompSetEntryPoint to preset callback routines for initialization, run, and finalization for a coupler component.

15 State Class

15.1 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 data transfers between 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 Arrays, ArrayBundles, Fields, FieldBundles, and other States. They cannot directly contain Fortran arrays. Objects in a State must span the VM on which they are running. For sequentially executing components which run on the same set of PETs this happens by calling the object create methods on each PET creating the object in unison. For concurrently executing components which are running on subsets of PETs, an additional reconcile method is provided by the ESMF to broadcast information about objects which were created in sub-components.

State methods include creation and deletion, adding and retrieving data items, adding and retrieving attributes, and performing queries.

15.2 State Options

15.2.1 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_FieldBundle 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_NOTFOUND Only valid as a return object type from a query routine. Indicates that no object with this name exists in the ESMF_State.
ESMF_STATEITEM_UNKNOWN Object type within an ESMF_State is unknown.

15.2.2 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.

15.3 Use and Examples

A Gridded Component generally has one associated import State and one export State. Generally the States associated with a Gridded Component will be created by the Gridded Component’s parent component. In many cases, the States will be created containing no data. Both the empty States and 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, FieldBundles, 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, FieldBundle, 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, FieldBundles, 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 FieldBundles 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.

Objects inside States are normally created in unison where each PET executing a component makes the same object create call. If the object contains data, like a Field, each PET may have a different local chunk of the entire dataset but each Field has the same name and is logically one part of a single distributed object. As States are passed between components if any object in a State was not created in unison on all the current PETs then some PETs have no object to pass into a communication method (e.g. regrid or data redistribution). A State method called reconcile must be called to broadcast information about these objects to all PETs in a component; after which all PETs have a single uniform view of all objects.

If components are running in sequential mode on all available PETs and States are being passed between them there is no need to make the reconcile call since all PETs have a uniform view of the objects. However, if components are running on a subset of the PETs, as is usually the case when running in concurrent mode, then when States are passed into components which contain a superset of those PETs, for example, a Coupler Component, all PETs must call reconcile on the States before using them in any ESMF communication methods. The reconcile process broadcasts metadata information about objects which exist only on a subset of the PETs. On PETs missing those objects it creates a proxy object which contains any attributes of the original object plus enough information for it to be a data source or destination for a regrid or data redistribution operation.

15.3.1 Empty State Create

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

    statename = "Atmosphere"
15.3.2 Adding Items to a State

Creation of an empty `ESMF_State`, and adding an `ESMF_FieldBundle` to it. Note that the `ESMF_FieldBundle` 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.

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

bundlename = "Temperature"
bundle1 = ESMF_FieldBundleCreate(name=bundlename, rc=rc)
print *, "FieldBundle Create returned", rc

call ESMF_StateAdd(state2, bundle1, rc)
print *, "StateAdd returned", rc

call ESMF_StateDestroy(state2, rc)
call ESMF_FieldBundleDestroy(bundle1, rc)
```

15.3.3 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.

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

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

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

15.3.4 Marking an Item Needed

How to set the `NEEDED` state of an item.

```fortran
dataname = "Downward wind"
call ESMF_StateSetNeeded(state3, dataname, ESMF_NEEDED, rc)
```
15.3.5 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.

```fortran
    dataname = "Downward wind"
    if (ESMF_StateIsNeeded(state3, dataname, rc)) then
      bundlename = dataname
      bundle2 = ESMF_FieldBundleCreate(name=bundlename, rc=rc)
      call ESMF.StateAdd(state3, bundle2, rc)
    else
      print *, "Data not marked as needed", trim(dataname)
    endif
```

15.3.6 Initialization and SetServices Routines

These are the separate subroutines called by the code above.

```fortran
  ! Initialize routine which creates "field1" on PETs 0 and 1
  subroutine comp1_init(gcomp, istate, ostate, clock, rc)
    type(ESMF_GridComp), intent(inout) :: gcomp
    type(ESMF_State), intent(inout) :: istate, ostate
    type(ESMF_Clock), intent(in) :: clock
    integer, intent(out) :: rc

    type(ESMF_Field) :: field1
    integer :: localrc

    print *, "i am comp1_init"
    field1 = ESMF_FieldCreateNoData(name="Comp1 Field", rc=localrc)
    call ESMF_StateAdd(istate, field1, rc=localrc)
    rc = localrc
  end subroutine comp1_init

  ! Initialize routine which creates "field2" on PETs 2 and 3
  subroutine comp2_init(gcomp, istate, ostate, clock, rc)
    type(ESMF_GridComp), intent(inout) :: gcomp
    type(ESMF_State), intent(inout) :: istate, ostate
    type(ESMF_Clock), intent(in) :: clock
    integer, intent(out) :: rc

    type(ESMF_Field) :: field2
    integer :: localrc

    print *, "i am comp2_init"
    field2 = ESMF_FieldCreateNoData(name="Comp2 Field", rc=localrc)
    call ESMF.StateAdd(istate, field2, rc=localrc)
    rc = localrc
  end subroutine comp2_init
```
print *, "i am comp2_init"

field2 = ESMF_FieldCreateNoData(name="Comp2 Field", rc=localrc)
call ESMF_StateAdd(istate, field2, rc=localrc)
rc = localrc
end subroutine comp2_init

subroutine comp_dummy(gcomp, rc)
type(ESMF_GridComp), intent(inout) :: gcomp
integer, intent(out) :: rc

rc = ESMF_SUCCESS
end subroutine comp_dummy

! !PROGRAM: ESMF_StateReconcileEx - State reconciliation !
! !DESCRIPTION:
! ! This program shows examples of using the State Reconcile function !---------------------------------------------------------------
!
! ESMF Framework module
use ESMF_Mod
use ESMF_StateReconcileEx_Mod
implicit none
!
! Local variables
integer :: rc, petCount
type(ESMF_State) :: state1
type(ESMF_GridComp) :: comp1, comp2
type(ESMF_VM) :: vm
character(len=ESMF_MAXSTR) :: comp1name, comp2name, statename

15.3.7 Creating Components on subsets of the current PET list

A Component can be created which will run only on a subset of the current PET list.

! Get the global VM for this job.
call ESMF_VMGetGlobal(vm=vm, rc=rc)
comp1name = "Atmosphere"
comp1 = ESMF_GridCompCreate(name=comp1name, petList=(/ 0, 1 /), rc=rc)
print *, "GridComp Create returned, name = ", trim(comp1name)
comp2name = "Ocean"
comp2 = ESMF_GridCompCreate(name=comp2name, petList=(/ 2, 3 /), rc=rc)
print *, "GridComp Create returned, name = ", trim(comp2name)
statename = "Ocn2Atm"
state1 = ESMF_StateCreate(statename, rc=rc)
15.3.8 Invoking Components on a subset of the Parent PETs

Here we register the subroutines which should be called for initialization. Then we call ESMF_GridCompInitialize() on all PETs, but the code runs only on the PETs given in the petList when the Component was created. Because this example is so short, we call the entry point code directly instead of the normal procedure of nesting it in a separate SetServices() subroutine.

! This is where the VM for each component is initialized.
! Normally you would call SetEntryPoint inside set services,
! but to make this example very short, they are called inline below.
! This is o.k. because the SetServices routine must execute from within
! the parent component VM.
call ESMF_GridCompSetServices(comp1, comp_dummy, rc)
call ESMF_GridCompSetServices(comp2, comp_dummy, rc)

print *, "ready to set entry point 1"
call ESMF_GridCompSetEntryPoint(comp1, ESMF_SETINIT, &
    comp1_init, ESMF_SINGLEPHASE, rc)

print *, "ready to set entry point 2"
call ESMF_GridCompSetEntryPoint(comp2, ESMF_SETINIT, &
    comp2_init, ESMF_SINGLEPHASE, rc)

print *, "ready to call init for comp 1"
call ESMF_GridCompInitialize(comp1, state1, rc=rc)
print *, "ready to call init for comp 2"
call ESMF_GridCompInitialize(comp2, state1, rc=rc)

15.3.9 Using State Reconcile

Now we have state1 containing field1 on PETs 0 and 1, and state1 containing field2 on PETs 2 and 3. For the code to have a rational view of the data, we call ESMF_StateReconcile which determines which objects are missing from any PET, and communicates information about the object. After the call to reconcile, all ESMF_State objects now have a consistent view of the data.

print *, "State before calling StateReconcile()"
call ESMF_StatePrint(state1, rc=rc)
call ESMF_StateReconcile(state1, vm, rc=rc)
print *, "State after calling StateReconcile()"
call ESMF_StatePrint(state1, rc=rc)

end program ESMF_StateReconcileEx

15.4 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. (The needed flag is fully supported.)
2. **No synchronization at object create time.** Object IDs are using during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time. User input at design time requested there be no communication overhead during the create of an object, so there is no opportunity to synchronize IDs if one or more PETs create objects which are not in unison (not all PETs in the VM make the same calls).

Even if the user follows the unison rules, if components are running on a subset of the PETs, when they return to the parent (calling) component the next available ID will potentially not be the same across all PETs in the VM. Part of the reconcile process or part of the return to the parent will need to have a broadcast which sends the current ID number, and all PETs can reset the next available number to the highest number broadcast. This could be an async call to avoid as much as possible serialization and barrier issues.

Default object names are based on the object id (e.g. "Field1", "Field2") to create unique object names, so basing the detection of unique objects on the name instead of on the object id is no better solution.

### 15.5 Design and Implementation Notes

1. 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 FieldBundles, 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.

2. Data contained in States must be created in unison on each PET of the current VM. This allows the creation process to avoid doing communications since each PET can compute any information it needs to know about any remote PET (for example, the grid distribute method can compute the decomposition of the grid on not only the local PET but also the remote PETs since it knows each PET is making the identical call). For all PETs to have a consistent view of the data this means objects must be given unique names when created, or all objects must be created in the same order on all PETs so ESMF can generate consistent default names for the objects.

When running components on subsets of the original VM all the PETs can create consistent objects but then when they are put into a State and passed to a component with a different VM and a different set of PETs, a communication call (reconcile) must be made to communicate the missing information to the PETs which were not involved in the original object creation. The reconcile call broadcasts object lists; those PETs which are missing any objects in the total list can receive enough information to reconstruct a proxy object which contains all necessary information about that object, with no local data, on that PET. These proxy objects can be queried by ESMF routines to determine the amount of data and what PETs contain data which is destined to be moved to the local PET (for receiving data) and conversely, can determine which other PETs are going to receive data and how much (for sending data).

For example, the FieldExcl system test creates 2 Gridded Components on separate subsets of PETs. They use the option of mapping particular, non-monotonic PETs to DEs. The following figures illustrate how the DEs are mapped in each of the Gridded Components in that test:

In the coupler code, all PETs must make the reconcile call before accessing data in the State. On PETs which already contain data, the objects are unchanged. On PETs which were not involved during the creation of the FieldBundles or Fields, the reconcile call adds an object to the State which contains all the same metadata associated with the object, but creates a slightly different Grid object, called a Proxy Grid. These PETs contain no local data, so the Array object is empty, and the DELayout for the Grid is like this:
Figure 7: The mapping of PETs (processors) to DEs (data) in the source grid created by *user_model1.F90* in the FieldExcl system test.

Figure 8: The mapping of PETs (processors) to DEs (data) in the destination grid created by *user_model2.F90* in the FieldExcl system test.
Figure 9: The mapping of PETs (processors) to DEs (data) in the source grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

Figure 10: The mapping of PETs (processors) to DEs (data) in the destination grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.
15.6 Object Model

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

15.7 Class API: Basic State Methods

15.7.1 ESMF_StateAdd - Add a single item to a State

**INTERFACE:**

```fortran
subroutine ESMF_StateAdd(state, <item>, rc)
```

**ARGUMENTS:**

- `type(ESMF_State), intent(inout) :: state`<br>
- `<item>`, see below for supported values<br>
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Add a reference to a single `<item>` to an existing `state`. Any of the supported `<item>`s can be marked needed for a particular run using the `ESMF_StateSetNeeded()` call. The name of the `<item>` must be unique within the `state`.

One of the supported options below is to add only the name of the item to the `state` during a first pass. The name can be replaced with the actual `<item>` in a later call. When doing this, the name of the `<item>` provided to the `state` during the first pass must match the name stored in the `<item>` itself.

Supported values for `<item>` are:

- `type(ESMF_Array), intent(in) :: array`
- `type(ESMF_ArrayBundle), intent(in) :: arraybundle`
- `type(ESMF_Field), intent(in) :: field`
- `type(ESMF_FieldBundle), intent(in) :: fieldbundle`
- `character (len=*)`, intent(in) :: name
- `type(ESMF_RouteHandle), intent(in) :: routehandle`
- `type(ESMF_State), intent(in) :: nestedState`
The arguments are:

**state**  The **ESMF_State** to which <item>s will be added.

**<item>**  The <item> to be added. This is a reference only; when the **state** is destroyed the <item>s contained in it will not be destroyed. Also, the <item> cannot be safely destroyed before the **state** is destroyed. Since <item>s can be added to multiple containers, it remains the user's responsibility to manage their destruction when they are no longer in use.

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

---

15.7.2 **ESMF_StateAdd** - Add a list of items to a State

**INTERFACE:**

```fortran
subroutine ESMF_StateAdd(state, <itemList>, count, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_State), intent(inout) :: state
<itemList>, see below for supported values
integer, intent(in), optional :: count
integer, intent(out), optional :: rc
```

**DESCRIPTION:**

Add a list of items to an **ESMF_State**.

Supported values for **<itemList>** are:

```fortran
type(ESMF_Array), intent(in) :: arrayList(:)
type(ESMF_ArrayBundle), intent(in) :: arraybundleList(:)
type(ESMF_Field), intent(in) :: fieldList(:)
type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)
character (len=*) intent(in) :: nameList(:)
type(ESMF_RouteHandle), intent(in) :: routehandleList(:)
type(ESMF_State), intent(in) :: stateList(:)
```

The arguments are:

**state**  An **ESMF_State** to which the **<itemList>** will be added.

**<itemList>**  The list of items to be added. This is a reference only; when the **ESMF_State** is destroyed the **<itemList>** contained in it will not be destroyed. Also, the **<itemList>** cannot be safely destroyed before the **ESMF_State** is destroyed. Since **<itemList>s** can be added to multiple containers, it remains the user's responsibility to manage their destruction when they are no longer in use.

**[count]**  The number of items to be added. By default equal to the size of the **<itemList>** argument.

**[rc]**  Return code; equals **ESMF_SUCCESS** if there are no errors.
15.7.3  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_FieldBundle), dimension(:), intent(inout), optional :: bundleList
    type(ESMF_Field), dimension(:), intent(inout), 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_FieldBundles.

[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 – FieldBundles, 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 9.1.8. 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 9.1.9. 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 9.1.14. 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 9.1.12. If not specified, the default value is set to ESMF_REQUIRED_FOR_RESTART.

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

15.7.4 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. Actual objects added to ESMF_States will not be destroyed, it remains the user’s responsibility to destroy these objects in the correct context. However, proxy objects automatically created during ESMF_StateReconcile() are destroyed when the State is destroyed.

The arguments are:

state Destroy contents of this ESMF_State.

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

15.7.5 ESMF_StateGet - Get information about a State

INTERFACE:

    ! Private name; call using ESMF_StateGet()
    subroutine ESMF_StateGetInfo(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:
An ESMF_State object to be queried.

Name of this ESMF_State.

Import or Export ESMF_State. Possible values are listed in Section 15.2.2.

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

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

Array of possible item object types in state, including placeholder names. Must be at least itemCount long. Options are listed in Section 15.2.1.

Return code; equals ESMF_SUCCESS if there are no errors.

[15.7.6 ESMF_StateGet - Retrieve an item from a State]

INTERFACE:

    subroutine ESMF_StateGet(state, itemName, <item>, nestedStateName, rc)

ARGUMENTS:

    type(ESMF_State), intent(in) :: state
    character (len=*) intent(in) :: itemName
    <item>, see below for supported values
    character (len=*) intent(in), optional :: nestedStateName
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns an <item> from an ESMF_State by name. If the ESMF_State contains the <item> directly, only itemName is required. If the state contains multiple nested ESMF_States and the <item> is one level down, this routine can return it in a single call by specifying the proper nestedStateName. ESMF_States can be nested to any depth, but this routine only searches immediate descendents. It is an error to specify a nestedStateName if the state contains no nested ESMF_States.

Supported values for <item> are:

    type(ESMF_Array), intent(out) :: array
    type(ESMF_ArrayBundle), intent(out) :: arraybundle
    type(ESMF_Field), intent(out) :: field
    type(ESMF_FieldBundle), intent(out) :: fieldbundle
    type(ESMF_RouteHandle), intent(out) :: routehandle
    type(ESMF_State), intent(out) :: nestedState

The arguments are:

    state  State to query for an <item> named itemName.
    itemName Name of <item> to be returned.
    <item> Returned reference to the <item>.
[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 <item> being requested is 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.

15.7.7 ESMF_StateGet - Get information about an item in a State

INTERFACE:

! Private name; call using ESMF_StateGet()
subroutine ESMF_StateGetItemInfo(state, name, stateitemtype, rc)

ARGUMENTS:

type(ESMF_State), intent(in) :: state
classer (len=*) char (len=*), intent(in) :: name
type(ESMF_StateItemType), intent(out) :: stateitemtype
integer, intent(out), optional :: rc

DESCRIPTION:

Returns the type for the item named name in this ESMF_State. If no item with this name exists, the value ESMF_STATEITEM_NOTFOUND will be returned and the error code will not be set to an error. Thus this routine can be used to safely query for the existence of items by name whether or not they are expected to be there. The error code will be set in case of other errors, for example if the ESMF_State itself is invalid.

The arguments are:

state ESMF_State to be queried.

name Name of the item to return information about.

stateitemtype Returned item types for the item with the given name, including placeholder names. Options are listed in Section [15.2.1] If no item with the given name is found, ESMF_STATEITEM_NOTFOUND will be returned and rc will not be set to an error.

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

15.7.8 ESMF_StateGetNeeded - Query whether a data item is needed

INTERFACE:

subroutine ESMF_StateGetNeeded(state, itemName, neededflag, rc)

ARGUMENTS:

type(ESMF_State), intent(in) :: state
classer (len=*) char (len=*), intent(in) :: itemName
type(ESMF_NeededFlag), intent(out) :: neededflag
integer, intent(out), optional :: rc
DESCRIPTION:

Returns 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 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 9.1.8.

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

15.7.9 ESMF_StateIsNeeded – Return logical true if data item needed

INTERFACE:

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

**RETURN VALUE:**

- logical :: ESMF_StateIsNeeded

**ARGUMENTS:**

```fortran
type(ESMF_State), intent(in) :: state
classer (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.

15.7.10 ESMF_StatePrint - Print the internal data for a State

INTERFACE:

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

**ARGUMENTS:**

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

Prints information about the state to stdout.
Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.
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.

15.7.11 ESMF_StateSetNeeded - Set if a data item is needed

INTERFACE:

subroutine ESMF_StateSetNeeded(state, itemName, neededflag, rc)

ARGUMENTS:

type(ESMF_State), intent(inout) :: state
type(charcome (len=*), intent(in) :: itemName

type(ESMF_NeededFlag), intent(in) :: neededflag

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 9.1.8 for possible values.

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

15.7.12 ESMF_StateValidate - Check validity of a State

INTERFACE:

subroutine ESMF_StateValidate(state, options, rc)

ARGUMENTS:

type(ESMF_State) :: state
type(charcome (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.

15.8  Class API: State Communications

15.8.1  ESMF_StateReconcile – Reconcile State data across all PETs in a VM

INTERFACE:

subroutine ESMF_StateReconcile(state, vm, options, rc)

ARGUMENTS:

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

DESCRIPTION:

Must be called for any ESMF_State which contains ESMF objects that have not been created on all the PETs of the currently running ESMF_Component. For example, if a coupler is operating on data which was created by another component that ran on only a subset of the coupler’s PETs, the coupler must make this call first before operating on any data inside that ESMF_State. After calling ESMF_StateReconcile all PETs will have a common view of all objects contained in this ESMF_State.

The arguments are:

state  ESMF_State to reconcile.

vm  ESMF_VM for this ESMF_Component.

[options]  Currently unused. Here for possible future expansion in the options for the reconciliation process.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
Part III
Infrastructure: Fields and Grids
16 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 DEs, 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.

16.1 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.

- **ArrayBundle** ArrayBundles are collections of Arrays that are stored in a single object. Unlike FieldBundles, they don’t need to be distributed the same way across PETs. The motivation for ArrayBundles is both convenience and performance.
• **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.

• **FieldBundle** Groups of Fields on the same underlying physical grid can be collected into a single object called a FieldBundle. A FieldBundle 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 FieldBundle 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.

  FieldBundle objects contain methods for setting and retrieving constituent fields, regridding, data I/O, and re-ordering of data in memory.

16.2 Design and Implementation Notes

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

17.1 Description

A FieldBundle functions mainly as a convenient container for storing similar Fields. It represents “bundles” of Fields that are discretized on the same Grid and distributed in the same manner. It is an important data structure because this is often the form that data being transferred between Components takes.

Fields within a FieldBundle may be located at different locations relative to the vertices of their common Grid. The Fields in a FieldBundle 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 FieldBundle.

FieldBundles can be created and destroyed, can have Attributes added or retrieved, and can have Fields added or retrieved. Methods include queries that return information about the FieldBundle itself and about the Fields that it contains. The Fortran data pointer of a Field within a FieldBundle can be obtained by first retrieving the the Field with a call to ESMF_FieldBundleGet, and then using the ESMF_FieldGet() method to get the data.

FieldBundles can be added to States, which are used for sending to or receiving data from Components. In the future FieldBundles will serve as a mechanism for performance optimization. ESMF will take advantage of the similarities of the Fields within a FieldBundle in order to implement collective communication, IO, and regridding. See Section 17.4 for a description of features that are being planned.

17.2 FieldBundle Options

17.2.1 ESMF_PackFlag

DESCRIPTION:
Specifies whether a FieldBundle is packed or not. A packed FieldBundle contains an array in which all the data in its constituent Fields is packed contiguously. FieldBundles 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.

17.3 Use and Examples

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

17.3.1 FieldBundle Creation

After creating multiple Fields, a FieldBundle can be created by passing a list of the Fields into the method ESMF_FieldBundleCreate(). The FieldBundle will contain references to the Fields. An empty FieldBundle can also be created and Fields added one at a time or in groups.

17.3.2 Accessing FieldBundle Data

To access data in a FieldBundle 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.

17.3.3 FieldBundle Deletion

The user must call ESMF_FieldBundleDestroy() before deleting any of the Fields it contains. Because Fields can be shared by multiple FieldBundles and States, they are not deleted by this call. See the following code fragments for examples of how to create new FieldBundles.
Example program showing various ways to create a FieldBundle object.

program ESMF_FieldBundleCreateEx

! ESMF Framework module
use ESMF_Mod

implicit none

! Local variables
integer :: i, rc, fieldcount

!-------------------------------------------------- -----------------------
! ! Create several Fields and add them to a new FieldBundle. !-------------------------------------------------- -----------------------
! Create an empty FieldBundle and then add a single field to it.
bundle2 = ESMF_FieldBundleCreate(name="time step 1", rc=rc)

call ESMF_FieldBundleAdd(bundle2, simplefield, rc)

call ESMF_FieldBundleGet(bundle2, fieldCount=fieldcount, rc=rc)
print *, "FieldBundle example 2 returned, fieldcount =", fieldcount

!-------------------------------------------------- -----------------------
! ! Create an empty FieldBundle and then add multiple fields to it.

bundle3 = ESMF_FieldBundleCreate(name="southern hemisphere", rc=rc)

call ESMF_FieldBundleAdd(bundle3, 3, field, rc)

call ESMF_FieldBundleGet(bundle3, fieldCount=fieldcount, rc=rc)
print *, "FieldBundle example 3 returned, fieldcount =", fieldcount

!-------------------------------------------------- -----------------------
! ! Get a Field back from a FieldBundle, first by name and then by index.
! ! Also get the FieldBundle name.

call ESMF_FieldBundleGet(bundle1, "pressure", returnedfield1, rc)

call ESMF_FieldGet(returnedfield1, name=fname1, rc=rc)

call ESMF_FieldBundleGet(bundle1, 2, returnedfield2, rc)

call ESMF_FieldGet(returnedfield2, name=fname2, rc=rc)

call ESMF_FieldBundleGet(bundle1, name=bname1, rc=rc)
print *, "FieldBundle example 4 returned, field names = ", &
       trim(fname1), ", ", trim(fname2)
print *, "FieldBundle name = ", trim(bname1)

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

call ESMF_FieldBundleDestroy(bundle1, rc=rc)

call ESMF_FieldBundleDestroy(bundle2, rc=rc)

call ESMF_FieldBundleDestroy(bundle3, rc=rc)
do i=1, 3
    call ESMF_FieldDestroy(field(i), rc=rc)
enddo

call ESMF_FieldDestroy(simplefield, rc=rc)

end program ESMF_FieldBundleCreateEx

17.4 Restrictions and Future Work

1. No mathematical operators. The FieldBundle class does not support differential or other mathematical operators. We do not anticipate providing this functionality in the near future.

2. Limited validation and print options. We are planning to increase the number of validity checks available for FieldBundles as soon as possible. We also will be working on print options.

3. Limited communication support. Only a subset of the communication routines are currently supported for FieldBundles, and the Fields contained in the FieldBundles must currently have the same structure (e.g. same halo width, same dimensionality). Support for more variable data will be added in a later release. For those routines not implemented yet, or for those FieldBundles which contain Fields with differing data, the user can loop over the Fields in the FieldBundle and call the Field level communication routines instead.

4. Packed data not supported. One of the options that we are currently working on for FieldBundles is packing. Packing means that the data from all the Fields that comprise the FieldBundle are 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. This will reduce the latency overhead of the communication.

5. Interleaving Fields within a FieldBundle. Data locality is important for performance on some computing platforms. An interleave option will allow the user to create a packed FieldBundle in which Fields are either concatenated in memory or in which Field elements are interleaved.

17.5 Design and Implementation Notes

1. Fields in a FieldBundle reference the same Grid. In order to reduce memory requirements and ensure consistency, the Fields within a FieldBundle all reference the same Grid object. This restriction may be relaxed in the future.

17.6 Class API: Basic FieldBundle Methods

17.6.1 ESMF_FieldBundleAdd - Add a Field to a FieldBundle

INTERFACE:

! Private name; call using ESMF_FieldBundleAdd()
subroutine ESMF_FieldBundleAddOneField(bundle, field, rc)

ARGUMENTS:

type(ESMF_FieldBundle), intent(inout) :: bundle
type(ESMF_Field), intent(inout) :: 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_FieldBundle to add the ESMF_Field to.
field  The ESMF_Field to add.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

17.6.2 ESMF_FieldBundleAdd - Add a list of Fields to a FieldBundle

INTERFACE:

! Private name; call using ESMF_FieldBundleAdd()
subroutine ESMF_FieldBundleAddFieldList(bundle, fieldCount, fieldList, rc)

ARGUMENTS:

type(ESMF_FieldBundle), intent(inout) :: bundle
integer, intent(in) :: fieldCount
type(ESMF_Field), dimension(:), intent(inout) :: fieldList
integer, intent(out), optional :: rc

DESCRIPTION:

Adds a fieldList to an existing ESMF_FieldBundle. 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_FieldBundle to add ESMF_Fields to.
fieldCount  Number of ESMF_Fields to be added to the ESMF_FieldBundle; 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_FieldBundle.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

17.6.3 ESMF_FieldBundleCreate - Create a FieldBundle from existing Fields

INTERFACE:

! Private name; call using ESMF_FieldBundleCreate()
function ESMF_FieldBundleCreateNew(fieldCount, fieldList, &
packflag, name, iospec, rc)

RETURN VALUE:

type(ESMF_FieldBundle) :: ESMF_FieldBundleCreateNew

ARGUMENTS:
integer, intent(in) :: fieldCount
type(ESMF_Field), dimension (:) :: fieldList
type(ESMF_PackFlag), intent(in), optional :: packflag
class (len = *), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc

DESCRIPTION:

Creates an ESMF_FieldBundle 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_FieldBundle. The arguments are:

fieldCount  Number of fields to be added to the new ESMF_FieldBundle. 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_FieldBundle.

[packflag]  The packing option is not yet implemented. See Section 17.2 for a description of packing, and Section
17.2.1 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.

[name]  ESMF_FieldBundle name. A default name is generated if one is not specified.

[iospec]  The ESMF_IOSpec is not yet used by ESMF_FieldBundles. Any values passed in will be ignored.

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

17.6.4  ESMF_FieldBundleCreate - Create a FieldBundle with no Fields

INTERFACE:

! Private name; call using ESMF_FieldBundleCreate()
function ESMF_FieldBundleCreateNoFields(grid, name, iospec, rc)

RETURN VALUE:

  type(ESMF_FieldBundle) :: ESMF_FieldBundleCreateNoFields

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_FieldBundle with no associated ESMF_Fields. The arguments are:

[grid]  The ESMF_Grid which all ESMF_Fields added to this ESMF_FieldBundle 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_FieldBundle.
ESMF_FieldBundle name. A default name is generated if one is not specified.

The ESMF_IOSpec is not yet used by ESMF_FieldBundles. Any values passed in will be ignored.

Return code; equals ESMF_SUCCESS if there are no errors.

### 17.6.5 ESMF_FieldBundleDestroy - Free all resources associated with a FieldBundle

**INTERFACE:**

```fortran
subroutine ESMF_FieldBundleDestroy(bundle, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_FieldBundle) :: 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_FieldBundle object.
- `rc` Return code; equals ESMF_SUCCESS if there are no errors.

### 17.6.6 ESMF_FieldBundleGet - Return information about a FieldBundle

**INTERFACE:**

```fortran
subroutine ESMF_FieldBundleGetInfo(bundle, grid, fieldCount, name, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_FieldBundle), intent(inout) :: 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_FieldBundle was originally created without specifying a name, a unique name will have been generated by the framework.

The arguments are:

- `bundle` The ESMF_FieldBundle object to query.
- `grid` The ESMF_Grid associated with the bundle.
- `fieldCount` Number of ESMF_Fields in the bundle.
- `name` A character string where the bundle name is returned.
- `rc` Return code; equals ESMF_SUCCESS if there are no errors.
17.6.7 ESMF_FieldBundleGet - Retrieve a Field by name

INTERFACE:

! Private name; call using ESMF_FieldBundleGet()
subroutine ESMF_FieldBundleGetFieldByName(bundle, name, field, rc)

ARGUMENTS:

type(ESMF_FieldBundle), intent(inout) :: 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_FieldBundle to query for ESMF_Field.
name ESMF_Field name.
field Returned ESMF_Field.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

17.6.8 ESMF_FieldBundleGet - Retrieve a Field by index number

INTERFACE:

! Private name; call using ESMF_FieldBundleGet()
subroutine ESMF_FieldBundleGetFieldByNum(bundle, fieldIndex, field, rc)

ARGUMENTS:

type(ESMF_FieldBundle), intent(inout) :: 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_FieldBundle 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.
17.6.9  ESMF_FieldBundleGet - Return all Field names in a FieldBundle

INTERFACE:

    ! Private name; call using ESMF_FieldBundleGet()
    subroutine ESMF_FieldBundleGetFieldNames(bundle, nameList, nameCount, rc)

ARGUMENTS:

    type(ESMF_FieldBundle), intent(inout) :: bundle
    character (len = *), intent(out) :: nameList(:)
    integer, intent(out), optional :: nameCount
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns an array of ESMF_Field names in an ESMF_FieldBundle. The arguments are:

bundle  An ESMF_FieldBundle object.

nameList   An array of character strings where each ESMF_Field name is returned. Must be at least as long as nameCount.

[nameCount] A count of how many ESMF_Field names were returned. Same as the number of ESMF_Fields in the ESMF_FieldBundle.

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

17.6.10  ESMF_FieldBundlePrint - Print information about a FieldBundle

INTERFACE:

    subroutine ESMF_FieldBundlePrint(bundle, options, rc)

ARGUMENTS:

    type(ESMF_FieldBundle), intent(inout) :: bundle
    character (len=*), intent(in), optional :: options
    integer, intent(out), optional :: rc

DESCRIPTION:

Prints diagnostic information about the bundle to stdout. Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release. The arguments are:

bundle  An ESMF_FieldBundle object.

[options] Print options are not yet supported.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
17.6.11 ESMF_FieldBundleSetGrid - Associate a Grid with an empty FieldBundle

INTERFACE:

    subroutine ESMF_FieldBundleSetGrid(bundle, grid, rc)

ARGUMENTS:

    type(ESMF_FieldBundle), intent(inout) :: bundle
    type(ESMF_Grid), intent(in) :: grid
    integer, intent(out), optional :: rc

DESCRIPTION:

Sets the grid for a bundle that contains no ESMF_Fields. All ESMF_Fields 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_FieldBundle object.

grid  The ESMF_Grid which all ESMF_Fields added to this ESMF_FieldBundle must have.

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

17.6.12 ESMF_FieldBundleValidate - Check validity of a FieldBundle

INTERFACE:

    subroutine ESMF_FieldBundleValidate(bundle, options, rc)

ARGUMENTS:

    type(ESMF_FieldBundle), 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_FieldBundle to validate.

[options]  Validation options are not yet supported.

[rc]  Return code; equals ESMF_SUCCESS if the bundle is valid.

18 Field Class

18.1 Description

An ESMF Field represents a physical field, such as temperature. The motivation for including Fields in ESMF is that bundles of Fields are the entities that are normally exchanged when coupling Components.
The ESMF Field class contains distributed, discretized field data, a reference to its associated grid, and metadata. The Field class maintains the relationship of how a data array maps onto a 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, ...). This means that different Fields which are on the same underlying ESMF Grid but have different staggerings can share the same Grid object without needing to replicate it multiple times.

Fields can be added to States for use in inter-Component data communications. Fields can also be added to FieldBundles, which are currently defined as groups of Fields on the same underlying grid. One motivation for FieldBundles is convenience; another is the ability to perform optimized collective data transfers. Field communications are not enabled for this release and will be restored in subsequent releases. These operations include data redistribution and regridding, and scatter and gather.

ESMF does not currently support vector fields, so the components of a vector field must be stored as separate Field objects.

### 18.2 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 can be queried and manipulated.

The sections below go into more detail about Field usage.

#### 18.2.1 Field Creation and Destruction

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 typekind 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.

When finished with a ESMF_Field, the ESMF_FieldDestroy method removes it. However, the objects inside the ESMF_Field created externally should be destroyed separately, since objects can be added to more than one ESMF_Field. For example, the same ESMF_Grid can be referenced by multiple ESMF_Fields. In this case the internal Grid is not deleted by the ESMF_FieldDestroy call.

#### 18.2.2 Get Fortran data pointer, bounds, and counts information from a Field

A user can get bounds and counts information from an ESMF_Field through the `ESMF_FieldGet()` interface. Also available through this interface is the intrinsic Fortran data pointer contained in the internal ESMF_Array object of an ESMF_Field. The bounds and counts information are DE specific for the associated Fortran data pointer.

In this example, we first create a 3D Field based on a 3D Grid and Array. Then we use the `ESMF_FieldGet()` interface to retrieve the data pointer, potentially updating or verifying its values. We also retrieve the bounds and counts information of the 3D Field to assist in data element iteration.
! create a 3D data Field from a Grid and Array.
! first create a Grid
grid3d = ESMF_GridCreateShapeTile(minIndex=(/1,1,1/), maxIndex=(/xdim,ydim,zdim/), &
regDecomp=(/2,2,1/), name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

allocate(farray(fa_shape(1), fa_shape(2), fa_shape(3)) )

! create an Array
array3d = ESMF_ArrayCreate(farray, distgrid=distgrid3d, &
    staggerloc=0, computationalEdgeLWidth=(/0,0,0/), &
    computationalEdgeUWidth=(/-1,-1,-1/), rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create a Field
field = ESMF_FieldCreate(grid3d, array3d, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

do k = totalLBnd(3), totalUBnd(3)
do j = totalLBnd(2), totalUBnd(2)
do i = totalLBnd(1), totalUBnd(1)
    farray1(i, j, k) = sin(2*i/total_count(1)*PI) + &
                      sin(4*j/total_count(2)*PI) + &
                      sin(8*k/total_count(2)*PI)
endo
endo
endo
18.2.3  Get Grid and Array and other information from a Field

A user can get the internal ESMF_Grid and ESMF_Array from a ESMF_Field. Note that the user should not issue any destroy command on the retrieved grid or array object since they are referenced from within the ESMF_Field. The retrieved objects should be used in a read-only fashion to query additional information not directly available through the ESMF_FieldGet() interface.

```fortran
call ESMF_FieldGet(field, grid=grid, array=array, &
    typekind=typekind, dimCount=dimCount, staggerloc=staggerloc, &
    gridToFieldMap=gridToFieldMap, &
    ungriddedLBound=ungriddedLBound, ungriddedUBound=ungriddedUBound, &
    maxHaloLWidth=maxHaloLWidth, maxHaloUWidth=maxHaloUWidth, &
    name=name, &
    rc=rc)
```

18.2.4  Create Field with Grid and Arrayspec

A user can create an ESMF_Field from an ESMF_Grid and a ESMF_Arrayspec with corresponding rank and type. This create method associates the two objects.

We first create a Grid with a regular distribution that is 10x20 in 2x2 DEs. This version of Field create simply associates the data with the Grid. The data is referenced explicitly on a regular 2x2 uniform grid. Then we create an ArraySpec. Finally we create a Field from the Grid, ArraySpec, and a user specified StaggerLoc.

This example also illustrates a typical use of this Field creation method. By creating a Field from a Grid and an ArraySpec, the user allows the ESMF library to create an internal Array in the Field. Then the user can use ESMF_FieldGet() to retrieve the Fortran data array and necessary bounds information to assign initial values to it.

```fortran
! create a grid
grid = ESMF_GridCreateShapeTile(minIndex=(/1,1/), maxIndex=(/10,20/), &
    regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! retrieve distgrid from the Grid
!call ESMF_GridGet(grid, distgrid=distgrid, rc=rc)
!if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! setup arrayspec
!call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4,rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create a Field from the Grid and arrayspec
field1 = ESMF_FieldCreate(grid, arrayspec, &
    staggerloc=ESMF_STAGGERLOC_CENTER, name="pressure", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(field1, localDe=0, farray=farray2dd, &
    totalLBound=ftlb, totalUBound=ftub, totalCount=ftc, rc=rc)
do i = ftlb(1), ftub(1)
do j = ftlb(2), ftub(2)
    farray2dd(i, j) = sin(i/ftc(1)*PI) * cos(j/ftc(2)*PI)
enddo
enddo
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```
A user can also create an ArraySpec that has a different rank from the Grid. For example, the following code shows creation of of 3D Field from a 2D Grid using a 3D ArraySpec.

This example also demonstrates the technique to create a typical 3D data Field that has 2 gridded dimensions and 1 ungridded dimension.

First we create a 2D grid with an index space of 180x360 equivalent to 180x360 Grid cells (note that for a distributed memory computer, this means each grid cell will be on a separate PE!). In the FieldCreate call, we use gridToFieldMap to indicate the mapping between Grid dimension and Field dimension. For the ungridded dimension (typically the altitude), we use ungriddedLBound and ungriddedUBound to describe its bounds. Internally the ungridded dimension has a stride of 1, so the number of elements of the ungridded dimension is ungriddedUBound - ungriddedLBound + 1. Note that gridToFieldMap in this specific example is (/1,2/) which is the default value so the user can neglect this argument for the FieldCreate call.

```
grid2d = ESMF_GridCreateShapeTile(minIndex=(/1,1/), maxIndex=(/180,360/), &
    regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 3, ESMF_TYPEKIND_R4, rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

field1 = ESMF_FieldCreate(grid2d, arrayspec, &
    staggerloc=ESMF_STAGGERLOC_CENTER, &
    gridToFieldMap=(/1,2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/50/), &
    name="pressure", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

### 18.2.5 Create Field with Grid and Array

A user can create an ESMF_Field from an ESMF_Grid and a ESMF_Array. The Grid was created in the previous example.

This example creates a 2D ESMF_Field from a 2D ESMF_Grid and a 2D ESMF_Array.

```
! Get necessary information from the Grid
call ESMF_GridGet(grid, staggerloc=ESMF_STAGGERLOC_CENTER, &
    computationalEdgeLWidth=compEdgeLWidth, &
    computationalEdgeUWidth=compEdgeUWidth, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_GridGet(grid, distgrid=distgrid, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Create a 2D ESMF_TYPEKIND_R4 arrayspec
call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Create a ESMF_Array from the arrayspec and distgrid
array2d = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, staggerLoc=0, &
    computationalEdgeLWidth=compEdgeLWidth, &
    computationalEdgeUWidth=compEdgeUWidth, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Create a ESMF_Field from the grid and array
field4 = ESMF_FieldCreate(grid, array2d, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```
18.2.6 Create an empty Field and finish it with FieldSetCommit

A user can create an empty ESMF_Field. Then the user can finalize the empty ESMF_Field from a ESMF_Grid and a intrinsic Fortran data array. This interface is overloaded for typekind and rank of the Fortran data array.

In this example, both grid and Fortran array pointer are 2 dimensional and each dimension index maps in order, i.e. 1st dimension of grid maps to 1st dimension of Fortran array pointer, 2nd dimension of grid maps to 2nd dimension of Fortran array pointer, so on and so forth.

In order to create or finish a Field from a Grid and a Fortran array pointer, certain rules of the Fortran array bounds must be obeyed. We will discuss these rules as we progress in Field creation examples. We will make frequent reference to the terminologies for bounds and widths in ESMF. For a better discussion of these terminologies and concepts behind them, e.g. exclusive, computational, total bounds for the lower and upper corner of data region, etc., users can refer to the explanation of these concepts for Grid and Array in their respective sections in the Reference Manual, e.g. Section 20.4.6 on Array and Section 23.2.10 on Grid. The examples here are designed to help a user to get up to speed with creating Fields for typical use.

This example introduces a helper method, part of the ESMF_FieldGet interface that facilitates the computation of Fortran data array bounds and shape to assist ESMF_FieldSetCommit finalizing a Field from a intrinsic Fortran data array and a Grid.

```fortran
! create an empty Field
field3 = ESMF_FieldCreateEmpty("precip", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! use FieldGet to retrieve total counts
call ESMF_FieldGet(grid2d, localDe=0, staggerloc=ESMF_STAGGERLOC_CENTER, &
                   totalCount=ftc, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! allocate the 2d Fortran array based on retrieved total counts
allocate(farray2d(ftc(1), ftc(2)))

! finalize the Field
call ESMF_FieldSetCommit(field3, grid2d, farray2d, rc=rc)
```

18.2.7 Create 7D Field with 5D Grid and 2D ungridded bounds from Fortran data array

In this example, we will show how to create a 7D Field from a 5D ESMF_Grid and 2D ungridded bounds with arbitrary halo widths and gridToFieldMap.

We first create a 5D DistGrid and a 5D Grid based on the DistGrid; then ESMF_FieldGet computes the shape of a 7D array in fsize. We can then create a 7D Field from the 5D Grid and the 7D Fortran data array with other assimilating parameters.

```fortran
! create a 5d distgrid
distgrid5d = ESMF_DistGridCreate(minIndex=(/1,1,1,1,1/), maxIndex=(/10,4,10,4,6/), &
                                   regDecomp=(/2,1,2,1,1/), rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Create a 5d Grid
grid5d = ESMF_GridCreate(distgrid=distgrid5d, name="grid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! use FieldGet to retrieve total counts
call ESMF_FieldGet(grid5d, localDe=0, ungriddedLBound=(/1,2/), &
                   ungriddedUBound=(/4,5/), &
                   maxHaloLWidth=(/1,1,2,2/), maxHaloUWidth=(/1,2,3,4,5/), &
                   gridToFieldMap=(/3,2,5,4,1/), &
```

113
totalCount = fsize, &
rc = rc)
if (rc .NE. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! allocate the 7d Fortran array based on retrieved total counts
allocate(farray7d(fsize(1), fsize(2), fsize(3), fsize(4), fsize(5), fsize(6), fsize(7)))

! create the Field
field7d = ESMF_FieldCreate(grid5d, farray7d, &
ungriddedLBound=(/1,2/), ungriddedUBound=(/4,5/), &
maxHaloLWidth=(/1,1,1,2,2/), maxHaloUWidth=(/1,2,3,4,5/), &
gridToFieldMap=(/3,2,5,4,1/), &
rc = rc)
if (rc .NE. ESMF_SUCCESS) finalrc = ESMF_FAILURE

A user can allocate the Fortran array in a different manner using the lower and upper bounds returned from FieldGet through the optional totalLBound and totalUBound arguments. In the following example, we create another 7D Field by retrieving the bounds and allocate the Fortran array with this approach. In this scheme, indexing the Fortran array is sometimes more convenient than using the shape directly.

call ESMF_FieldGet(grid5d, localDe=0, ungriddedLBound=(/1,2/), &
ungriddedUBound=(/4,5/), &
maxHaloLWidth=(/1,1,1,2,2/), maxHaloUWidth=(/1,2,3,4,5/), &
gridToFieldMap=(/3,2,5,4,1/), &
totalLBound=flbound, totalUBound=fubound, &
rc = rc)
if (rc .NE. ESMF_SUCCESS) finalrc = ESMF_FAILURE

flbound(7):fubound(7))

field7d2 = ESMF_FieldCreate(grid5d, farray7d2, &
ungriddedLBound=(/1,2/), ungriddedUBound=(/4,5/), &
maxHaloLWidth=(/1,1,1,2,2/), maxHaloUWidth=(/1,2,3,4,5/), &
gridToFieldMap=(/3,2,5,4,1/), &
rc = rc)
if (rc .NE. ESMF_SUCCESS) finalrc = ESMF_FAILURE

18.2.8 Create 2D Field with 2D Grid and Fortran data array
A user can create an ESMF_Field directly from an ESMF_Grid and an intrinsic Fortran data array. This interface is overloaded for typekind and rank of the Fortran data array.

In the following example, each dimension size of the Fortran array must be no greater than the maximum value of the computational and exclusive bounds of its corresponding Grid dimension queried from the Grid through ESMF_GridGet() public interface.

Formally let fa_shape(i) be the shape of i-th dimension of user supplied Fortran array, then rule 1 states:

\[ fa\_shape(i) = \max(\text{computationalCount}(i), \text{exclusiveCount}(i)) \]

\[ i = 1 \ldots \text{GridDimCount} \]
fa_shape(i) defines the shape of i-th dimension of the Fortran array. computationalCount and exclusiveCount are the number of data elements of i-th dimension in the computational and exclusive regions queried from ESMF_GridGet interface. Rule 1 assumes that the Grid and the Fortran intrinsic array have same number of dimensions; and optional arguments of FieldCreate from Fortran array are left unspecified using default setup. These assumptions are true for most typical use of FieldCreate from Fortran data array. This is the easiest way to create a Field from a Grid and Fortran intrinsic data array.

Fortran array dimension sizes (called shape in most Fortran language books) are equivalent to the bounds and counts used in this manual. The following equation holds:

\[
fa_shape(i) = shape(i) = counts(i) = upper_bound(i) - lower_bound(i) + 1
\]

These typically mean the same concept unless specifically explained to mean something else. For example, ESMF uses DimCount very often to mean number of dimensions instead of its meaning implied in the above equation. We'll clarify the meaning of a word when ambiguity could occur.

Rule 1 is most useful for a user working with Field creation from a Grid and a Fortran data array in most scenarios. It extends to higher dimension count, 3D, 4D, etc... Typically, as the code example demonstrates, a user first creates a Grid, then uses ESMF_GridGet() to retrieve the computational and exclusive counts. Next the user calculates the shape of each Fortran array dimension according to rule 1. The Fortran data array is allocated and initialized based on the computed shape. A Field can either be created in one shot created empty and finished using ESMF_FieldSetCommit.

There are important details that can be skipped but are good to know for ESMF_FieldSetCommit and ESMF_FieldCreate from a Fortran data array. 1) these methods require each PET contains exactly one DE. This implies that a code using FieldCreate from a data array or FieldSetCommit must have the same number of DEs and PETs, formally \( n_{DE} = n_{PET} \). Violation of this condition will cause run time failures. 2) the bounds and counts retrieved from GridGet are DE specific or equivalently PET specific, which means that the Fortran array shape could be different from one PET to another.

```fortran
grid = ESMF_GridCreateShapeTile(minIndex=(/1,1/), maxIndex=(/10,20/), &
regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

allocate(farray(max(gec(1), gcc(1)), max(gec(2), gcc(2))))

field = ESMF_FieldCreate(grid, farray, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

### 18.2.9 Create 3D Field with 2D Grid and 3D Fortran data array

This example demonstrates a typical use of ESMF_Field combining a 2D grid and a 3D Fortran native data array. One immediate problem follows: how does one define the bounds of the ungridded dimension? This is solved by the optional arguments ungriddedLBound and ungriddedUBound of the ESMF_FieldCreate interface. By definition, ungriddedLBound and ungriddedUBound are both 1 dimensional integer Fortran arrays. Formally, let \( fa_{shape}(j=1...FieldDimCount-GridDimCount) \) be the shape of the ungridded dimensions of a Field relative to the Grid used in Field creation. The Field dimension count is equal to the number of dimensions of the Fortran array, which equals the number of dimensions of the resultant Field. GridDimCount is the number of dimensions of the Grid.

\( fa_{shape}(j) \) is computed as:
\[
\text{fa\_shape}(j) = \text{ungriddedUBound}(j) - \text{ungriddedLBound}(j) + 1
\]

fa\_shape is easy to compute when the gridded and ungridded dimensions do not mix. However, it’s conceivable that at higher dimension count, gridded and ungridded dimensions can interleave. To aid the computation of ungridded dimension shape we formally introduce the mapping concept.

Let \( \text{map}_{A,B}(i = 1 \ldots n_A) = i_B, \) and \( i_B \in [\phi, 1 \ldots n_B] \). \( n_A \) is the number of elements in set A, \( n_B \) the number of elements in set B. \( \text{map}_{A,B}(i) \) defines a mapping from \( i \)-th element of set A to \( i_B \)-th element in set B. \( i_B = \phi \) indicates there does not exist a mapping from \( i \)-th element of set A to set B.

Suppose we have a mapping from dimension index of ungriddedLBound (or ungriddedUBound) to Fortran array dimension index, called ugb2fa. By definition, \( n_A \) equals to the dimension count of ungriddedLBound (or ungriddedUBound), \( n_B \) equals to the dimension count of the Fortran array. We can now formulate the computation of ungridded dimension shape as rule 2:

\[
\text{fa\_shape}(\text{ugb2fa}(j)) = \text{ungriddedUBound}(j) - \text{ungriddedLBound}(j) + 1
\]

\( j = 1 \ldots \text{FortranArrayDimCount} - \text{GridDimCount} \)

The mapping can be computed in linear time proportional to the Fortran array dimension count (or rank) using the following algorithm in pseudocode:

```plaintext
map_index = 1
do i = 1, farray_rank
    if i-th dimension of farray is ungridded
        ugb2fa(map_index) = i
        map_index = map_index + 1
    endif
endo
```

Here we use rank and dimension count interchangeably. These 2 terminologies are typically equivalent. But there are subtle differences under certain conditions. Rank is the total number of dimensions of a tensor object. Dimension count allows a finer description of the heterogeneous dimensions in that object. For example, A Field of rank 5 can have 3 gridded dimensions and 2 ungridded dimensions. Rank is precisely the summation of dimension count of all types of dimensions.

For example, if a 5D array is used with a 3D Grid, there are 2 ungridded dimensions: ungriddedLBound=(/1,2/) and ungriddedUBound=(/5,7/). Suppose the distribution of dimensions look like (O, X, O, X, O), O means gridded, X means ungridded. Then the mapping from ungridded bounds to Fortran array is ugb2fa=(/2, 4/). The shape of 2nd and 4th dimension of Fortran array should equal (5, 8).

Back to our 3D Field created from a 2D Grid and 3D Fortran array example, suppose the 3rd Field dimension is ungridded, ungriddedLBound=(/3/), ungriddedUBound=(/9/). First we use rule 1 to compute shapes of the gridded Fortran array dimension, then we use rule 2 to compute shapes of the ungridded Fortran array dimension. In this example, we used the computational and exclusive bounds obtained in previous example.

```plaintext
fa\_shape(1) = \text{max}(\text{gec}(1), \text{gcc}(1)) \text{ ! rule 1}
fa\_shape(2) = \text{max}(\text{gec}(2), \text{gcc}(2))
fa\_shape(3) = 7 \text{ ! rule 2} 9-3+1
allocate(farray3d(fa\_shape(1), fa\_shape(2), fa\_shape(3)))
field = ESMF\_FieldCreate(grid, farray3d, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    rc=rc)
if(rc .ne. ESMF\_SUCCESS) finalrc = ESMF\_FAILURE
```
18.2.10 Create 3D Field with 2D Grid and 3D Fortran data array with gridToFieldMap

Building upon the previous example, we will create a 3D Field from a 2D grid and 3D array but with a slight twist. In this example, we introduce the gridToFieldMap argument that allows a user to map Grid dimension index to Field dimension index.

In this example, both dimensions of the Grid are distributed and the mapping from DistGrid to Grid is (/1,2/). We will introduce rule 3 assuming distgridToGridMap=(/1,2,3...gridDimCount/), and distgridDimCount equals to gridDimCount. This is a reasonable assumption in typical Field use.

We apply the mapping gridToFieldMap on rule 1 to create rule 3:

\[
(3) \text{fa\_shape(gridToFieldMap(i))} = \text{max(computationalCount(i), exclusiveCount(i))}
\]
\[i = 1,...,\text{GridDimCount}.
\]

Back to our example, suppose the 2nd Field dimension is ungridded, ungriddedLBound=(/3/), ungriddedUBound=(/9/), gridToFieldMap=(/3,1/), meaning the 1st Grid dimension maps to 3rd Field dimension, and 2nd Grid dimension maps to 1st Field dimension.

First we use rule 3 to compute shapes of the gridded Fortran array dimension, then we use rule 2 to compute shapes of the ungridded Fortran array dimension. In this example, we used the computational and exclusive bounds obtained in the previous example.

\[
\text{gridToFieldMap2d(1) = 3}
\]
\[
\text{gridToFieldMap2d(2) = 1}
\]
\[
\text{do i = 1, 2}
\]
\[
\text{fa\_shape(gridToFieldMap2d(i)) = max(gec(i), gcc(i))}
\]
\[
\text{end do}
\]
\[
\text{fa\_shape(2) = 7}
\]
\[
\text{allocate(farray3d(fa\_shape(1), fa\_shape(2), fa\_shape(3)))}
\]
\[
\text{field = ESMF\_FieldCreate(grid, farray3d, ungriddedLBound=/(3/), ungriddedUBound=/(9/), gridToFieldMap=gridToFieldMap2d, &}
\]
\[
\text{rc=rc)}
\]
\[
\text{if(rc .ne. ESMF\_SUCCESS) finalrc = ESMF\_FAILURE}
\]

18.2.11 Create 3D Field with 2D Grid and 3D Fortran data array with halos

This example is similar to example 18.2.10 in addition we will show a user can associate different halo width to a Fortran array to create a Field through the maxHaloLWidth and maxHaloUWdith optional arguments. A diagram of the dimension configuration from Grid, halos, and Fortran data array is shown here.

The ESMF_FieldCreate() interface supports creating a Field from a Grid and a Fortran array padded with halos on the distributed dimensions of the Fortran array. Using this technique one can avoid passing non-contiguous Fortran array slice to FieldCreate. It guarantees the same computational region, and by using halos, it also defines a bigger total region to contain the entire contiguous memory block of the Fortran array.

The elements of maxHaloLWidth and maxHaloUWdith are applied in the order distributed dimensions appear in the Fortran array. By definition, maxHaloLWidth and maxHaloUWdith are 1 dimensional arrays of non-negative integer values. The size of haloWidth arrays is equal to the number of distributed dimensions of the Fortran array, which is also equal to the number of distributed dimensions of the Grid used in the Field creation.

Because the order of maxHaloWidth (representing both maxHaloLWidth and maxHaloUWdith) element is applied to the order distributed dimensions appear in the Fortran array dimensions, it’s quite simple to compute the shape of distributed dimensions of the Fortran array. They are done in a similar manner when applying ungriddedLBound and ungriddedUBound to ungridded dimensions of the Fortran array defined by rule 2.
ESMF_Field created from a 2D ESMF_Grid (Red) and a 3D Intrinsic Fortran data array (Green). The ungridded bounds and halo widths are applied to corresponding dimensions.
Assume we have the mapping from the dimension index of maxHaloWidth to the dimension index of Fortran array, called mhw2fa; and we also have the mapping from dimension index of Fortran array to dimension index of the Grid, called fa2g. The shape of distributed dimensions of a Fortran array can be computed by rule 4:

\[
(4) \text{fa\_shape}(\text{mhw2fa}(k)) = \max((\text{exclusiveCount}(\text{fa2g(}\text{mhw2fa}(k)))) + \\
\text{computationalCount}(\text{fa2g(}\text{mhw2fa}(k))) + \\
\text{maxHaloUWidth}(k) + \text{maxHaloLWidth}(k))
\]

\[k = 1...\text{size(maxHaloWidth)}\]

This rule may seem confusing but algorithmically the computation can be done by the following pseudocode:

```plaintext
fa\_index = 1
\text{do } i = 1, \text{farray\_rank}
  \text{if } i\text{-th dimension of Fortran array is distributed}
    \text{fa\_shape}(i) = \max(\text{exclusiveCount}(\text{fa2g}(i))) + \\
    \text{computationalCount}(\text{fa2g}(i)) + \\
    \text{maxHaloUWidth(}\text{fa\_index}) + \text{maxHaloLWidth(}\text{fa\_index})
  \text{fa\_index} = \text{fa\_index} + 1
\text{endif}
\text{endo}
```

The only complication then is to figure out the mapping from Fortran array dimension index to Grid dimension index. This process can be done by computing the reverse mapping from Field to Grid. Typically, we don’t have to consider these complications if the following conditions are met: 1) All Grid dimensions are distributed. 2) DistGrid in the Grid has a dimension index mapping to the Grid in the form of natural order (/1,2,3,.../). This natural order mapping is the default mapping between various objects throughout ESMF. 3) Grid to Field mapping is in the form of natural order, i.e. default mapping. These seem like a lot of conditions but they are the default case in the interaction among DistGrid, Grid, and Field. When these conditions are met, which is typically true, the shape of distributed dimensions of Fortran array follows rule 5 in a simple form:

\[
(5) \text{fa\_shape}(k) = \max(\text{exclusiveCount}(k) + \text{computationalCount}(k) + \\
\text{maxHaloUWidth}(k) + \text{maxHaloLWidth}(k))
\]

\[k = 1...\text{size(maxHaloWidth)}\]

Let’s examine an example on how to apply rule 5. Suppose we have a 5D array and a 3D Grid that has its first 3 dimensions mapped to the first 3 dimensions of the Fortran array. maxHaloLWidth=(/1,2,3/), maxHaloUWidth=(/7,9,10/), then by rule 5, the following pseudo code can be used to compute the shape of the first 3 dimensions of the Fortran array. The shape of the remaining two ungridded dimensions can be computed according to rule 2.

```plaintext
\text{do } k = 1, 3
  \text{fa\_shape}(k) = \max(\text{exclusiveCount}(k), \text{computationalCount}(k) + \\
  \text{maxHaloUWidth}(k) + \text{maxHaloLWidth}(k))
\text{endo}
```

Suppose now gridToFieldMap=(/2,3,4/) instead which says the first dimension of Grid maps to the 2nd dimension of Field (or Fortran array) and so on and so forth, we can obtain a more general form of rule 5 by introducing first_distdim_index shift when Grid to Field map (gridToFieldMap) is in the form of (/a,a+1,a+2.../).
(6) \( fa\_shape(k+\text{first\_distdim\_index}-1) = \max(\text{exclusiveCount}(k), \text{computationalCount}(k)) + \maxHaloUWidth(k) + \maxHaloLWidth(k) \)
\( k = 1...\text{size}(\maxHaloWidth) \)

It’s obvious that \( \text{first\_distdim\_index} = a \). If the first dimension of the Fortran array is distributed, then rule 6 degenerates into rule 5, which is the typical case.

Back to our example creating a 3D Field from a 2D Grid and a 3D intrinsic Fortran array, we will use the Grid created from previous example that satisfies condition 1 and 2. We’ll also use a simple gridToFieldMap \( (1,2) \) which is the default mapping that satisfies condition 3. First we use rule 5 to compute the shape of distributed dimensions then we use rule 2 to compute the shape of the ungridded dimensions.

```fortran
gridToFieldMap2d(1) = 1
gridToFieldMap2d(2) = 2
maxHaloLWidth2d(1) = 3
maxHaloLWidth2d(2) = 4
maxHaloUWidth2d(1) = 3
maxHaloUWidth2d(2) = 5
do k = 1, 2
    fa\_shape(k) = \max(\text{gec}(k), \text{gcc}(k)+\maxHaloLWidth2d(k)+\maxHaloUWidth2d(k) )
end do
fa\_shape(3) = 7 ! 9-3+1
allocate(farray3d(fa\_shape(1), fa\_shape(2), fa\_shape(3)))
field = ESMF\_FieldCreate(grid, farray3d, &
    ungriddedLBound=/(3/), ungriddedUBound=/(9/), &
    maxHaloLWidth=maxHaloLWidth2d, maxHaloUWidth=maxHaloUWidth2d, &
    gridToFieldMap=gridToFieldMap2d, &
    rc=rc)
if(rc .ne. ESMF\_SUCCESS) finalrc = ESMF\_FAILURE
```

18.3 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.

18.4 Design and Implementation Notes

1. Some methods which have a Field interface are actually implemented at the underlying Grid or Array level; they are 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.

2. The Field class is implemented in Fortran, and as such is defined inside the framework by a Field derived type and a set of subprograms (functions and subroutines) which operate on that derived type. The Field class itself is very thin; it is a container class which groups a Grid and an Array object together.

3. Fields follow the framework-wide convention of the unison creation and operation rule: All PETs which are part of the currently executing VM must create the same Fields at the same point in their execution. Since an early user request was that global object creation not impose the overhead of a barrier or synchronization point,
Field creation does no inter-PET communication. For this to work, each PET must query the total number of PETs in this VM, and which local PET number it is. It can then compute which DE(s) are part of the local decomposition, and any global information can be computed in unison by all PETs independently of the others. In this way the overhead of communication is avoided, at the cost of more difficulty in diagnosing program bugs which result from not all PETs executing the same create calls.

4. Related to the item above, the user request to not impose inter-PET communication at object creation time means that requirement FLD 1.5.1, that all Fields will have unique names, and if not specified, the framework will generate a unique name for it, is difficult or impossible to support. A part of this requirement has been implemented; a unique object counter is maintained in the Base object class, and if a name is not given at create time a name such as “Field003” is generated which is guaranteed to not be repeated by the framework. However, it is impossible to error check that the user has not replicated a name, and it is possible under certain conditions that if not all PETs have created the same number of objects, that the counters on different PETs may not stay synchronized. This remains an open issue.

18.5 Class API

18.5.1 ESMF_FieldCreate - Create a Field from Grid and ArraySpec

INTERFACE:

Private name; call using ESMF_FieldCreate()

function ESMF_FieldCreateFromArraySpec(grid, arrayspec, &
staggerloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
maxHaloLWidth, maxHaloUWidth, name, iospec, rc)

RETURN VALUE:

type(ESMF_Field) :: ESMF_FieldCreateFromArraySpec

ARGUMENTS:

type(ESMF_Grid) :: grid

type(ESMF_ArraySpec), intent(inout) :: arrayspec

type(ESMF_StaggerLoc), intent(in), optional :: staggerloc

integer, intent(in), optional :: gridToFieldMap(:)

integer, intent(in), optional :: ungriddedLBound(:)

integer, intent(in), optional :: ungriddedUBound(:)

integer, intent(in), optional :: maxHaloLWidth(:)

integer, intent(in), optional :: maxHaloUWidth(:)

character (len=*) intent(in), optional :: name

type(ESMF_IOSpec), intent(in), optional :: iospec

integer, intent(out), optional :: rc

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see Section 18.2.2. The arguments are:

grid ESMF_Grid object.

arrayspec Data type and kind specification.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see Section 23.3.3. To create a custom stagger location see Section 23.2.16. The default value is ESMF_STAGGERLOC_CENTER.
**List with number of elements equal to the grid’s dimCount.** The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid’s dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = ([1,2,3,...]). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

**Lower bounds of the ungridded dimensions of the field.** The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

**Upper bounds of the ungridded dimensions of the field.** The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

**Lower bound of halo region.** The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the minHaloLWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

**Upper bound of halo region.** The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the maxHaloUWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

**Field name.**

**I/O specification.** ! NOT IMPLEMENTED

**Return code; equals ESMF_SUCCESS if there are no errors.**

### 18.5.2 ESMF_FieldCreate - Create a Field from Grid and Array

**INTERFACE:**

```fortran
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateFromArray(grid, array, copyflag, staggerloc, &
    gridToFieldMap, ungriddedLBound, ungriddedUBound, maxHaloLWidth, &
    maxHaloUWidth, name, iospec, rc)

RETURN VALUE:
    type(ESMF_Field) :: ESMF_FieldCreateFromArray

ARGUMENTS:
```

122
type (ESMF_Grid), intent(in) :: grid
type (ESMF_Array), intent(in) :: array
type (ESMF_CopyFlag), intent(in), optional :: copyflag
type (ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: maxHaloLWidth(:)
integer, intent(in), optional :: maxHaloUWidth(:)
character (len = *), intent(in), optional :: name
type (ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc

DESCRIPTION:

Create an ESMF_Field. This version of creation assumes the data exists already and is being passed in through an ESMF_Array. For an example and associated documentation using this method see Section 18.2.5. The arguments are:

grid  ESMF_Grid object.
array  ESMF_Array object.

[copyflag] Indicates whether to copy the array or reference it directly. For valid values see 9.1.4. The default is ESMF_DATA_REF.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see Section 23.5.3. To create a custom stagger location see Section 23.2.16. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid’s dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the field.

[maxHaloLWidth] Lower bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the fields size should be max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the minHaloLWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED.
[maxHaloUWidth] Upper bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the maxHaloUWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

[name] Field name.

[iospec] I/O specification. NOT IMPLEMENTED

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

18.5.3 ESMF_FieldCreate - Create a Field from Fortran array pointer

INTERFACE:

! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateFromDataPtr<rank><type><kind>(grid, &
farray, copyflag, staggerloc, gridToFieldMap, ungriddedLBound, &
ungriddedUBound, maxHaloLWidth, maxHaloUWidth, name, iospec, rc)

RETURN VALUE:

type(ESMF_Field) :: ESMF_FieldCreateFromDataPtr<rank><type><kind>

ARGUMENTS:

type(ESMF_Grid) :: grid
<type> (ESMF_KIND_<kind>), dimension(<rank>), target :: farray
type(ESMF_CopyFlag), intent(in), optional :: copyflag
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: maxHaloLWidth(:)
integer, intent(in), optional :: maxHaloUWidth(:)
class (len=*), intent(in), optional :: name
type(ESMF_IOSpec), intent(in), optional :: iospec
integer, intent(out), optional :: rc

DESCRIPTION:

Create an ESMF_Field from a fortran data array and ESMF_Grid. For examples and associated documentations using this method see Section 18.2.8, 18.2.9, 18.2.10, 18.2.11, and 18.2.7.

The arguments are:

grid ESMF_Grid object. The dimCount of the Grid must be smaller than or equal to the rank of the farray.

farray Native fortran data array to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farray.

[copyflag] Whether to copy the farray or reference directly. For valid values see 9.1.4. The default is ESMF_DATA_REF.
Stagger location of data in grid cells. For valid predefined values see Section 23.5.3. To create a custom stagger location see Section 23.2.16. The default value is ESMF_STAGGERLOC_CENTER.

List with number of elements equal to the grid’s dimCount. The list elements map each dimension of the grid to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the grid’s dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farray rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farray dimensions less the total (distributed + undistributed) dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation.

Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the farray.

Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the farray.

Lower bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the farray. Values default to 0. If values for maxHaloLWidth are specified they must be reflected in the size of the farray. That is, for each gridded dimension the farray size should be max( maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount ). Although the halo operation is not implemented, the minHaloLWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

Upper bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the farray. Values default to 0. If values for maxHaloUWidth are specified they must be reflected in the size of the farray. That is, for each gridded dimension the farray size should max( maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount ). Although the halo operation is not implemented, the maxHaloUWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

Field name.

I/O specification. NOT IMPLEMENTED

Return code; equals ESMF_SUCCESS if there are no errors.

18.5.4 ESMF_FieldCreateEmpty - Create an empty Field (no Grid)

INTERFACE:

function ESMF_FieldCreateEmpty(name, iospec, rc)

RETURN VALUE:
type (ESMF_Field) :: ESMF_FieldCreateEmpty

ARGUMENTS:

character (len = *), intent(in), optional :: name

type (ESMF_IOSpec), intent(in), optional :: iospec

integer, intent(out), optional :: rc

DESCRIPTION:

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. Attributes can be added to an empty Field object. For an example and associated documentation using this method see Section 18.2.6.

The arguments are:

[name] Field name.

[iospec] I/O specification. ! NOT IMPLEMENTED

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

---

18.5.5 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] ESMF_Field object.

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

---

18.5.6 ESMF_FieldGet - Return info associated with a Field

INTERFACE:

! Private name; call using ESMF_FieldGet()

subroutine ESMF_FieldGetDefault(field, grid, array, typekind, dimCount, &

staggerloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &

maxHaloLWidth, maxHaloUWidth, name, iospec, rc)

ARGUMENTS:
type (ESMF_Field), intent(inout) :: field
type (ESMF_Grid), intent(out), optional :: grid
type (ESMF_Array), intent(out), optional :: array
type (ESMF_TypeKind), intent(out), optional :: typekind
integer, intent(out), optional :: dimCount
type (ESMF_StaggerLoc), intent(out), optional :: staggerloc
integer, intent(out), optional :: gridToFieldMap(:)
integer, intent(out), optional :: ungriddedLBound(:)
integer, intent(out), optional :: ungriddedUBound(:)
integer, intent(out), optional :: maxHaloLWidth(:)
integer, intent(out), optional :: maxHaloUWidth(:)
character(len=*), intent(out), optional :: name
type (ESMF_IOSpec), intent(out), optional :: iospec ! NOT IMPLEMENTED
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. For an example and associated documentation using this method see Section 18.2.3.

The arguments are:

field  ESMF_Field object to query.
[grid]  ESMF_Grid.
[array]  ESMF_Array.
[typekind]  TypeKind specifier for Field.
[dimCount]  Number of dimensions in field data.
[staggerloc]  Stagger location of data in grid cells. For valid predefined values and interpretation of results see Section 23.5.3.
[gridToFieldMap]  List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound]  Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound]  Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[maxHaloLWidth]  Lower bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(maxHaloLWidth + maxHaloUWidth + computationalCount,
exclusiveCount). Although the halo operation is not implemented, the minHaloLWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

[maxHaloUWidth] Upper bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the maxHaloUWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

[name] Name of queried item.

[iospec] ESMF_IOSpec object which contains settings for options. NOT IMPLEMENTED

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

### 18.5.7 ESMF_FieldGet - Get Fortran data pointer from a Field

**INTERFACE:**

```fortran
subroutine ESMF_FieldGetDataPtr<rank><type><kind>(field, localDe, farray, & exclusiveLBound, exclusiveUBound, exclusiveCount, & computationalLBound, computationalUBound, computationalCount, & totalLBound, totalUBound, totalCount, rc)
```

**ARGUMENTS:**

- `type(ESMF_Field), intent(in) :: field`
- `integer, intent(in) :: localDe`
- `<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: farray`
- `integer, intent(out), optional :: exclusiveLBound(:)`
- `integer, intent(out), optional :: exclusiveUBound(:)`
- `integer, intent(out), optional :: exclusiveCount(:)`
- `integer, intent(out), optional :: computationalLBound(:)`
- `integer, intent(out), optional :: computationalUBound(:)`
- `integer, intent(out), optional :: computationalCount(:)`
- `integer, intent(out), optional :: totalLBound(:)`
- `integer, intent(out), optional :: totalUBound(:)`
- `integer, intent(out), optional :: totalCount(:)`
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Get a Fortran pointer to DE-local memory allocation within field. For convenience DE-local bounds can be queried at the same time. For an example and associated documentation using this method see Section 18.2.2.

The arguments are:

- `field` ESMF_Field object.
- `localDe` The local DE from which to get the information.
- `farray` Fortran array pointer which will be pointed at DE-local memory allocation.
Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the number of items in the exclusive region per dimension (i.e. `exclusiveUBound-exclusiveLBound+1`). `exclusiveCount` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the lower bounds of the computational region. `computationalLBound` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the lower bounds of the computational region. `computationalLBound` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `totalCount` must be allocated to be of size equal to field's `dimCount`. See section 20.2.6 for a description of the regions and their associated bounds and counts.

Return code; equals ESMF_SUCCESS if there are no errors.

18.5.8 ESMF_FieldGet - Get precomputed Fortran data array bounds

INTERFACE:

```fortran
! Private name; call using ESMF_FieldGet()
subroutine ESMF_FieldGetAllocBounds(grid, localDe, staggerloc, gridToFieldMap, &
  ungriddedLBound, ungriddedUBound, &
  maxHaloLWidth, maxHaloUWidth, &
  totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```fortran
type(ESMF_Grid), intent(inout) :: grid
integer, intent(in) :: localDe
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
```
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: maxHaloLWidth(:)
integer, intent(in), optional :: maxHaloUWidth(:)
integer, intent(out), optional :: totalLBound(:)
integer, intent(out), optional :: totalUBound(:)
integer, intent(out), optional :: totalCount(:)
integer, intent(out), optional :: rc

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a ESMF_Field from a ESMF_Grid and the Fortran data array. For an example and associated documentation using this method see Section 18.2.7.

The arguments are:

grid ESMF_Grid.

localDE The local DE number in its PET context to compute the bounds and counts information based on the computational and exclusive bounds and counts information of the grid from that local DE in its PET context.

[staggerloc] Stagger location of data in grid cells. For valid predefined values and interpretation of results see Section 23.5.3.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the field.

[maxHaloLWidth] Lower bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the minHaloLWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

[maxHaloUWidth] Upper bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the maxHaloUWidth is checked for
validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED.

[totalLBound] The relative lower bounds of Fortran data array to be used later in tt ESMF_FieldCreate from ESMF_Grid and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in tt ESMF_FieldCreate from ESMF_Grid and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in tt ESMF_FieldCreate from ESMF_Grid and Fortran data array. This is an output variable from this user interface.

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

18.5.9 ESMF_FieldPrint - Print the contents of a Field

INTERFACE:

    subroutine ESMF_FieldPrint(field, options, rc)

ARGUMENTS:

    type(ESMF_Field), intent(inout) :: field
    character (len = *), intent(in), optional :: options
    integer, intent(out), optional :: rc

DESCRIPTION:

Prints information about the field to stdout. This subroutine goes through the internal data members of a field data type and prints information of each data member.

Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.

The arguments are:

field An ESMF_Field object.

[options] Print options are not yet supported.

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

18.5.10 ESMF_FieldSetCommit - Finishes creating Field started with FieldCreateEmpty

INTERFACE:

    ! Private name; call using ESMF_FieldSetCommit()
    subroutine ESMF_FieldSetCommit<rank><type><kind>(field, grid, &
        farray, copyflag, staggerloc, gridToFieldMap, ungriddedLBound, &
        ungriddedUBound, maxHaloLWidth, maxHaloUWidth, rc)

ARGUMENTS:


DESCRIPTION:

This call completes an ESMF_Field allocated with the ESMF_FieldCreateEmpty() call. For an example and associated documentation using this method see Section 18.2.6. The arguments are:

- **field** The ESMF_Field object to be completed and committed in this call. The field will have the same dimension (dimCount) as the rank of the farray.

- **grid** The ESMF_Grid object to finish the Field. The dimCount of the Grid must be smaller than or equal to the rank of the farray.

- **farray** Native fortran data array to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farray.

- **copyflag** Indicates whether to copy the farray or reference it directly. For valid values see ESMF_DATA_COPY. The default is ESMF_DATA_REF.

- **staggerloc** Stagger location of data in grid cells. For valid predefined values see ESMF_STAGGERLOC. The default value is ESMF_STAGGERLOC_CENTER.

- **gridToFieldMap** List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

- **ungriddedLBound** Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the field.

- **ungriddedUBound** Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the ordering of these ungridded dimensions is the same as their order in the field.

- **maxHaloLWidth** Lower bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(maxHaloLWidth + maxHaloUBound + computationalCount,
exclusiveCount). Although the halo operation is not implemented, the minHaloLWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

**[maxHaloUWidth]** Upper bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for maxHaloUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(maxHaloLWidth + maxHaloUWidth + computationalCount, exclusiveCount). Although the halo operation is not implemented, the maxHaloUWidth is checked for validity and stored in preparation for the implementation of the halo method. HALO OPERATION NOT IMPLEMENTED

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

### 18.5.11 ESMF_FieldValidate - Check validity of a Field

**INTERFACE:**

```fortran
subroutine ESMF_FieldValidate(field, options, rc)
```

**ARGUMENTS:**

- `type(ESMF_Field), intent(inout) :: field`
- `character (len = *), intent(in), optional :: 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. It validates the contained array and grid objects. The code also checks if the array and grid sizes agree. This check compares the distgrid contained in array and grid; then it proceeds to compare the computational bounds contained in array and grid.

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.

### 19 ArrayBundle Class

#### 19.1 Description

The ESMF_ArrayBundle class allows a set of Arrays to be bundled into a single object. The Arrays in an ArrayBundle may be of different type, kind, rank and distribution. Besides ease of use resulting from bundeling the ArrayBundle class offers the opportunity for performance optimization when operating on a bundle of Arrays as a single entity. Especially communication methods are good candidates for performance optimization. Best optimization results are expected for ArrayBundles that contain Arrays that share a common distribution, i.e. DistGrid, and are of same type, kind and rank.

ArrayBundles are one of the data objects that can be added to States, which are used for providing to or consuming data from other components.
19.2 Use and Examples

Examples of creating, destroying and accessing ArrayBundles and their constituent Arrays are provided in this section, along with some notes on ArrayBundle methods.

19.2.1 ArrayBundle creation from a list of Arrays

First create an array of two ESMF_Array objects.

```fortran
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)

  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
                               regDecomp=(/2,3/), rc=rc)

  array(1) = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
  array(2) = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

Now the array of Arrays can be used to create an ArrayBundle object.

```fortran
  arraybundle = ESMF_ArrayBundleCreate(arrayList=array, &
                              name="MyArrayBundle", rc=rc)
```

The ArrayBundle object can be printed.

```fortran
  call ESMF_ArrayBundlePrint(arraybundle, rc=rc)
```

19.2.2 Access Arrays inside the ArrayBundle

Use ESMF_ArrayBundleGet() to determine how many Arrays are stored in an ArrayBundle.

```fortran
  call ESMF_ArrayBundleGet(arraybundle, arrayCount=arrayCount, rc=rc)
```

The arrayCount can be used to correctly allocate the arrayList variable for a second call to ESMF_ArrayBundleGet() to gain access to the bundled Array objects.

```fortran
  allocate(arrayList(arrayCount))
  call ESMF_ArrayBundleGet(arraybundle, arrayList=arrayList, rc=rc)
```

The arrayList variable can be used to access the individual Arrays, e.g. to print them.

```fortran
  do i=1, arrayCount
    call ESMF_ArrayPrint(arrayList(i), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
  enddo
```
19.2.3 Destroy an ArrayBundle and its constituents

The ArrayBundle object can be destroyed.

    call ESMF_ArrayBundleDestroy(arraybundle, rc=rc)

After the ArrayBundle object has been destroyed it is safe to destroy its constituents.

    call ESMF_ArrayDestroy(array(1), rc=rc)
    call ESMF_ArrayDestroy(array(2), rc=rc)
    call ESMF_DistGridDestroy(distgrid, rc=rc)

19.3 Restrictions and Future Work

- Adding Arrays to an existing ArrayBundle is currently not supported. In the future this functionality will be provided via the ESMF_ArrayBundleAdd() method.

- The current implementation of the ArrayBundle communication methods only support ArrayBundles that contain congruent Arrays of the same type, kind and rank. Congruent Arrays are defined on matching DistGrids and the shape of the DE-local tiles matches for each DE.

- The current implementation of the ArrayBundle communication methods have not been optimized for performance or memory usage.

19.4 Design and Implementation Notes

The following is a list of implementation specific details about the current ESMF ArrayBundle.

- Implementation language is C++.

- All precomputed communication methods are based on sparse matrix multiplication.

19.5 Class API

19.5.1 ESMF_ArrayBundleCreate - Create an ArrayBundle from a list of Arrays

INTERFACE:

    ! Private name; call using ESMF_ArrayBundleCreate()
    function ESMF_ArrayBundleCreate(arrayList, arrayCount, name, rc)

ARGUMENTS:

    type(ESMF_Array), intent(in) :: arrayList(:)
    integer, intent(in), optional :: arrayCount
    character (len=*) , intent(in), optional :: name
    integer, intent(out), optional :: rc

RETURN VALUE:

    type(ESMF_ArrayBundle) :: ESMF_ArrayBundleCreate
DESCRIPTION:

Create an ESMF_ArrayBundle object from a list of Arrays. The creation of an ArrayBundle leaves the bundled Arrays unchanged, they remain valid individual objects. An ArrayBundle is a light weight container of Array references. The actual data remains in place, there are no data movements or duplications associated with the creation of an ArrayBundle.

arrayList  List of ESMF_Array objects to be bundled.

[arrayCount] If provided specifies that only first arrayCount Arrays in the arrayList argument are to be included in the ArrayBundle. By default arrayCount is equal to size(arrayList).

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

19.5.2   ESMF_ArrayBundleDestroy - Destroy ArrayBundle object

INTERFACE:

    subroutine ESMF_ArrayBundleDestroy(arraybundle, rc)

ARGUMENTS:

    type(ESMF_ArrayBundle), intent(inout) :: arraybundle
    integer, intent(out), optional :: rc

DESCRIPTION:

Destroy an ESMF_ArrayBundle object. The member Arrays are not touched by this operation and remain valid objects that need to be destroyed individually if necessary. The arguments are:

arraybundle  ESMF_ArrayBundle object to be destroyed.

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

19.5.3   ESMF_ArrayBundleGet - Get list of Arrays out of an ArrayBundle

INTERFACE:

    ! Private name; call using ESMF_ArrayBundleGet()
    subroutine ESMF_ArrayBundleGet(arraybundle, arrayCount, arrayList, &
     name, rc)

ARGUMENTS:

    type(ESMF_ArrayBundle), intent(in) :: arraybundle
    integer, intent(out), optional :: arrayCount
    type(ESMF_Array), intent(inout), optional :: arrayList(:)
    character(len=*)& integer, intent(out), optional :: name
    integer, intent(out), optional :: rc

DESCRIPTION:

Get the list of Arrays bundled in an ArrayBundle.
arraybundle  ESMF_ArrayBundle to be queried.

[arrayCount] Upon return holds the number of Arrays bundled in the ArrayBundle.

[arrayList] Upon return holds a List of Arrays bundled in ArrayBundle. The argument must be allocated to be at least of size arrayCount.

[name] Name of the ArrayBundle object.

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

19.5.4 ESMF_ArrayBundlePrint - Print ArrayBundle internals

INTERFACE:

    subroutine ESMF_ArrayBundlePrint(arraybundle, options, rc)

ARGUMENTS:

    type(ESMF_ArrayBundle), intent(in) :: arraybundle
    character(len=*) , intent(in), optional :: options
    integer, intent(out) , optional :: rc

DESCRIPTION:

Print internal information of the specified ESMF_ArrayBundle object.
Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush (6) as a workaround until this issue is fixed in a future release.

The arguments are:

arraybundle  ESMF_ArrayBundle object.

[options] Print options are not yet supported.

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

19.5.5 ESMF_ArrayBundleRedist - Execute an ArrayBundle redistribution

INTERFACE:

    subroutine ESMF_ArrayBundleRedist(srcArrayBundle, dstArrayBundle, 
      routehandle, checkflag, rc)

ARGUMENTS:

    type(ESMF_ArrayBundle), intent(in), optional :: srcArrayBundle
    type(ESMF_ArrayBundle), intent(out), optional :: dstArrayBundle
    type(ESMF_RouteHandle), intent(inout) :: routehandle
    type(ESMF_Logical), intent(in), optional :: checkflag
    integer, intent(out) , optional :: rc
DESCRIPTION:

Execute a precomputed ArrayBundle redistribution from the Arrays in `srcArrayBundle` to the Arrays in `dstArrayBundle`. This call is collective across the current VM.

[srcArrayBundle] ESMF_ArrayBundle with source data.
[dstArrayBundle] ESMF_ArrayBundle with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `ESMF_TRUE` the input Array pairs will be checked for consistency with the precomputed operation provided by routehandle. If set to `ESMF_FALSE` (default) only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to `ESMF_FALSE` to achieve highest performance.

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

---

19.5.6 ESMF_ArrayBundleRedistRelease - Release resources associated with ArrayBundle redistribution

INTERFACE:

```
subroutine ESMF_ArrayBundleRedistRelease(routehandle, rc)
ARGUMENTS:
  type(ESMF_RouteHandle), intent(inout) :: routehandle
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Release resources associated with an ArrayBundle redistribution. After this call routehandle becomes invalid.

routehandle Handle to the precomputed Route.

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

---

19.5.7 ESMF_ArrayBundleRedistStore - Precompute ArrayBundle redistribution with local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleRedistStore()
subroutine ESMF_ArrayBundleRedistStore<type><kind>(srcArrayBundle, &
  dstArrayBundle, routehandle, factor, srcToDstTransposeMap, rc)
ARGUMENTS:
  type(ESMF_ArrayBundle), intent(in) :: srcArrayBundle
  type(ESMF_ArrayBundle), intent(inout) :: dstArrayBundle
  type(ESMF_RouteHandle), intent(inout) :: routehandle
  <type>(ESMF_KIND_<kind>), intent(in) :: factor
  integer, intent(in), optional :: srcToDstTransposeMap(:)
  integer, intent(out), optional :: rc
```
DESCRIPTION:

Store an ArrayBundle redistribution operation from srcArrayBundle to dstArrayBundle. The redistribution between ArrayBundles is defined as the sequence of individual Array redistributions over all source and destination Array pairs in sequence. The method requires that srcArrayBundle and dstArrayBundle reference an identical number of ESMF_Array objects.

The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify a factor argument must use the <type><kind> overloaded interface. Other PETs call into the interface without factor argument. If multiple PETs specify the factor argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a factor argument the default will be a factor of 1.

See the description of method ESMF_ArrayRedistStore() for the definition of the Array based operation.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayBundleRedist() on any pair of ArrayBundles that are congruent and typekind conform with the Arrays contained in srcArrayBundle and dstArrayBundle. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is collective across the current VM.

srcArrayBundle  ESMF_ArrayBundle with source data.

dstArrayBundle  ESMF_ArrayBundle with destination data.

routehandle  Handle to the precomputed Route.

[factor]  Factor by which to multiply source data. Default is 1.

[srcToDstTransposeMap]  List with as many entries as there are dimensions in the Arrays in srcArrayBundle. Each entry maps the corresponding source Array dimension against the specified destination Array dimension. Mixing of distributed and undistributed dimensions is supported.

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

19.5.8  ESMF_ArrayBundleRedistStore - Precompute ArrayBundle redistribution without local factor argument

INTERFACE:

! Private name; call using ESMF_ArrayBundleRedistStore()
subroutine ESMF_ArrayBundleRedistStoreNF(srcArrayBundle, dstArrayBundle, & routehandle, srcToDstTransposeMap, rc)

ARGUMENTS:

type (ESMF_ArrayBundle), intent (in) :: srcArrayBundle

type (ESMF_ArrayBundle), intent (inout) :: dstArrayBundle

type (ESMF_RouteHandle), intent (inout) :: routehandle

integer, intent (in), optional :: srcToDstTransposeMap()

integer, intent (out), optional :: rc
Store an ArrayBundle redistribution operation from `srcArrayBundle` to `dstArrayBundle`. The redistribution between ArrayBundles is defined as the sequence of individual Array redistributions over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.

The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify a `factor` argument must use the `<type><kind>` overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1. See the description of method `ESMF_ArrayRedistStore()` for the definition of the Array based operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleRedist()` on any pair of ArrayBundles that are congruent and typekind conform with the Arrays contained in `srcArrayBundle` and `dstArrayBundle`. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.

This call is `collective` across the current VM.

### srcArrayBundle
`ESMF_ArrayBundle` with source data.

### dstArrayBundle
`ESMF_ArrayBundle` with destination data.

### routehandle
Handle to the precomputed Route.

### [srcToDstTransposeMap]
List with as many entries as there are dimensions in the Arrays in `srcArrayBundle`. Each entry maps the corresponding source Array dimension against the specified destination Array dimension. Mixing of distributed and undistributed dimensions is supported.

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

---

### 19.5.9 `ESMF_ArrayBundleSMM` - Execute an ArrayBundle sparse matrix multiplication

**INTERFACE:**

```fortran
subroutine ESMF_ArrayBundleSMM(srcArrayBundle, dstArrayBundle, routehandle, &
    zeroflag, checkflag, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_ArrayBundle), intent(in), optional :: srcArrayBundle
type(ESMF_ArrayBundle), intent(inout),optional :: dstArrayBundle
type(ESMF_RouteHandle), intent(inout) :: routehandle
type(ESMF_RegionFlag), intent(in), optional :: zeroflag
type(ESMF_Logical), intent(in), optional :: checkflag
integer, intent(out), optional :: rc
```

**DESCRIPTION:**

Execute a precomputed ArrayBundle sparse matrix multiplication from the Arrays in `srcArrayBundle` to the Arrays in `dstArrayBundle`. This call is `collective` across the current VM.

### [srcArrayBundle]
`ESMF_ArrayBundle` with source data.

### [dstArrayBundle]
`ESMF_ArrayBundle` with destination data.

### routehandle
Handle to the precomputed Route.
[zeroflag] If set to ESMF_REGION_TOTAL (default) the total regions of all DEs in all Arrays in dstArrayBundle will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to ESMF_REGION_EMPTY the elements in the Arrays in dstArrayBundle will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting zeroflag to ESMF_REGION_SELECT will only zero out those elements in the destination Arrays that will be updated by the sparse matrix multiplication. See section 9.1.11 for a complete list of valid settings.

[checkflag] If set to ESMF_TRUE the input Array pairs will be checked for consistency with the precomputed operation provided by routehandle. If set to ESMF_FALSE (default) only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to ESMF_FALSE to achieve highest performance.

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

19.5.10 ESMF_ArrayBundleSMMRelease - Release resources associated with ArrayBundle sparse matrix multiplication

INTERFACE:

subroutine ESMF_ArrayBundleSMMRelease(routehandle, rc)

ARGUMENTS:

  type(ESMF_RouteHandle), intent(inout) :: routehandle
type(ESMF_RouteHandle), intent(inout), optional :: rc

DESCRIPTION:

Release resources associated with an ArrayBundle sparse matrix multiplication. After this call routehandle becomes invalid.

routehandle Handle to the precomputed Route.

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

19.5.11 ESMF_ArrayBundleSMMStore - Precompute ArrayBundle sparse matrix multiplication with local factors

INTERFACE:

! Private name; call using ESMF_ArrayBundleSMMStore()
subroutine ESMF_ArrayBundleSMMStore<type><kind>(srcArrayBundle, & dstArrayBundle, routehandle, factorList, factorIndexList, rc)

ARGUMENTS:

  type(ESMF_ArrayBundle), intent(in) :: srcArrayBundle
type(ESMF_ArrayBundle), intent(inout) :: dstArrayBundle
type(ESMF_RouteHandle), intent(inout) :: routehandle
<type>(ESMF_KIND_<kind>), target, intent(in) :: factorList(:)
integer, intent(in) :: factorIndexList(:,:)
type(ESMF_RouteHandle), intent(inout), optional :: rc
DESCRIPTION:

Store an ArrayBundle sparse matrix multiplication operation from `srcArrayBundle` to `dstArrayBundle`. The sparse matrix multiplication between ArrayBundles is defined as the sequence of individual Array sparse matrix multiplications over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of ESMF_Array objects. The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify non-zero matrix coefficients must use the `<type><kind>` overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface without `factorList` and `factorIndexList` arguments. See the description of method `ESMF_ArraySMMStore()` for the definition of the Array based operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleSMM()` on any pair of ArrayBundles that are congruent and typekind conform with the Arrays contained in `srcArrayBundle` and `dstArrayBundle`. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE. This method is overloaded for:

- `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is `collective` across the current VM.

- `srcArrayBundle` ESMF_ArrayBundle with source data.
- `dstArrayBundle` ESMF_ArrayBundle with destination data.
- `routehandle` Handle to the precomputed Route.
- `factorList` List of non-zero coefficients.
- `factorIndexList` Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4. In the `size 2 format` `factorIndexList(1,:)` specifies the sequence index of the source element in the source Array while `factorIndexList(2,:)` specifies the sequence index of the destination element in the destination Array. For this format to be a valid option source and destination Arrays must have matching number of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The `size 4 format` is more general and does not require a matching tensor element count. Here the `factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the source Array. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the destination Array.

See section 20.2.15 for details on the definition of Array sequence indices and tensor sequence indices.

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

19.5.12 ESMF_ArrayBundleSMMStore - Precompute ArrayBundle sparse matrix multiplication without local factors

INTERFACE:
subroutine ESMF_ArrayBundleSMMStoreNF(srcArrayBundle, dstArrayBundle, &
routehandle, rc)

ARGUMENTS:

  type(ESMF_ArrayBundle), intent(in)            :: srcArrayBundle
  type(ESMF_ArrayBundle), intent(inout)         :: dstArrayBundle
  type(ESMF_RouteHandle), intent(inout)         :: routehandle
  integer, intent(out), optional               :: rc

DESCRIPTION:

Store an ArrayBundle sparse matrix multiplication operation from srcArrayBundle to dstArrayBundle. The sparse matrix multiplication between ArrayBundles is defined as the sequence of individual Array sparse matrix multiplications over all source and destination Array pairs in sequence. The method requires that srcArrayBundle and dstArrayBundle reference an identical number of ESMF_Array objects. The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the factorList and factorIndexList arguments. Providing factorList and factorIndexList arguments with size(factorList) = (/0/) and size(factorIndexList) = (/2,0/) or (/4,0/) indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface without factorList and factorIndexList arguments.

See the description of method ESMF_ArraySMMStore() for the definition of the Array based operation. The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayBundleSMM() on any pair of ArrayBundles that are congruent and typekind conform with the Arrays contained in srcArrayBundle and dstArrayBundle. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is collective across the current VM.

srcArrayBundle  ESMF_ArrayBundle with source data.
dstArrayBundle  ESMF_ArrayBundle with destination data.
routehandle     Handle to the precomputed Route.

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

20 Array Class

20.1 Description

The Array class is an alternative to the Field class for representing distributed, structured data. Unlike Fields, which are built to carry grid coordinate information, Arrays can only carry information about the indices associated with grid cells. Since they do not have coordinate information, Arrays cannot be used to calculate interpolation weights. However, if the user can supply interpolation weights (using a package such as SCRIP), the Array sparse matrix multiply operation can be used to apply the weights and transfer data to the new grid. Arrays can also perform redistribution, scatter, and gather operations.

Like Fields, Arrays can be added to a State and used in inter-component data communications. Arrays can also be grouped together into ArrayBundles so that collective operations can be performed on the whole group. One motivation for this is convenience; another is the ability to schedule optimized, collective data transfers.

From a technical standpoint, the ESMF_Array class is an index space based, distributed data storage class. It provides DE-local memory allocations within DE-centric index regions and defines the relationship to the index space described
by DistGrid. The Array class offers common communication patterns within the index space formalism. As part of the ESMF index space layer Array has close relationship to the DistGrid and DELayout classes.

## 20.2 Use and Examples

An `ESMF_Array` is a distributed object that must exist on all PETs of the current context. Each PET-local instance of an Array object contains memory allocations for all PET-local DEs. There may be 0, 1, or more DEs per PET and the number of DEs per PET can differ between PETs for the same Array object. Memory allocations may be provided for each PET by the user during Array creation or can be allocated as part of the Array create call. Many of the concepts of the proposed `ESMF_Array` class are illustrated by the following examples.

### 20.2.1 Array from native Fortran90 array with 1 DE per PET

The create call of the `ESMF_Array` class has been overloaded extensively to facilitate the need for generality while keeping simple cases simple. The following program demonstrates one of the simpler cases, where existing local Fortran90 arrays are to be used to provide the PET-local memory allocations for the Array object.

```fortran
program ESMF_ArrayFarrayEx
    use ESMF_Mod

    implicit none

    ! local variables
    real(ESMF_KIND_R8) :: farrayE(10,10) ! explicit shape F90 array

    ! allocate and initialize farrayE
    ! ... (code for allocation and initialization)

    ! allocate and initialize farrayA
    ! ... (code for allocation and initialization)

    ! call ESMF_CreateArray
    ! ... (call to ESMF_CreateArray)

    ! ... (code for Array operations)
end program ESMF_ArrayFarrayEx
```

The Fortran90 language provides a variety of ways to define and allocate an array. Actual Fortran90 array objects must either be explicit-shape or deferred-shape. In the first case the memory allocation and deallocation is automatic from the user’s perspective and the details of the allocation (static or dynamic, heap or stack) are left to the compiler. (Compiler flags may be used to control some of the details). In the second case, i.e. for deferred-shape actual objects, the array definition must include the pointer or allocatable attribute and it is the user’s responsibility to allocate memory. While it is also the user’s responsibility to deallocate memory for arrays with pointer attribute the compiler will automatically deallocate allocatable arrays under certain circumstances defined by the Fortran standard.

The `ESMF_ArrayCreate()` interface has been written to accept native Fortran90 arrays of any flavor as a means to allow user-controlled memory management. The Array create call will check on each PET if sufficient memory has been provided by the specified Fortran90 arrays and will indicate an error if a problem is detected. However, the Array create call cannot validate the lifetime of the provided memory allocations. If, for instance, an Array object was created in a subroutine from an automatic explicit-shape array or an allocatable array, the memory allocations referenced by the Array object will be automatically deallocated on return from the subroutine unless provisions are made by the application writer to prevent such behavior. The Array object cannot control when memory that has been provided by the user during Array creation becomes deallocated, however, the Array will indicate an error if it’s memory references have been invalidated.

The easiest, portable way to provide safe native Fortran90 memory allocations to Array create is to use arrays with the pointer attribute. Memory allocated for an Array pointer will not be deallocated automatically. However, in this case the possibility of memory leaks becomes an issue of concern. The deallocation of memory provided to an Array in form of a native Fortran90 allocation will remain the users responsibility.

None of the concerns discussed above are an issue in this example where the native Fortran90 array `farray` is defined in the main program. All different types of array memory allocation are demonstrated in this example. First `farrayE` is defined as a 2D explicit-shape array on each PET which will automatically provide memory for $10 \times 10$ elements.

```
! local variables
real(ESMF_KIND_R8) :: farrayE(10,10) ! explicit shape F90 array
```

Then an allocatable array `farrayA` is declared which will be used to show user-controlled dynamic memory allocation.
real(ESMF_KIND_R8), allocatable :: farrayA(:,:) ! allocatable F90 array

Finally an array with pointer attribute farrayP is declared, also used for user-controlled dynamic memory allocation.

real(ESMF_KIND_R8), pointer :: farrayP(:,:) ! F90 array pointer

A matching array pointer must also be available to gain access to the arrays held by an Array object.

real(ESMF_KIND_R8), pointer :: farrayPtr(:,:) ! matching F90 array pointer
type(ESMF_DistGrid) :: distgrid ! DistGrid object
type(ESMF_Array) :: array ! Array object
integer :: rc

call ESMF_Initialize(rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)

On each PET farrayE can be accessed directly to initialize the entire PET-local array.

farrayE = 12.45d0 ! initialize to some value

In order to create an Array object a DistGrid must first be created that describes the total index space and how it is decomposed and distributed. In the simplest case only the minIndex and maxIndex of the total space must be provided.

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)

This example is assumed to run on 4 PETs. The default 2D decomposition will then be into 4 x 1 DEs as to ensure 1 DE per PET.
Now the Array object can be created using the farrayE and the DistGrid just created.

array = ESMF_ArrayCreate(farray=farrayE, distgrid=distgrid, rc=rc)

The 40 x 10 index space defined by the minIndex and maxIndex arguments paired with the default decomposition will result in the following distributed Array.

```
+---------------------------> 2nd dimension
| (1,1)--------+
|  (DE 0)    <---- farray on PET 0
| +--------(10,10)
| (11,1)--------+
|   (DE 1)    <---- farray on PET 1
| +--------(20,10)
| (21,1)--------+
|     (DE 2)    <---- farray on PET 2
```
Providing \texttt{farrayE} during Array creation does not change anything about the actual \texttt{farrayE} object. This means that each PET can use its local \texttt{farrayE} directly to access the memory referenced by the Array object.

\begin{verbatim}
print *, farrayE
\end{verbatim}

Another way of accessing the memory associated with an Array object is to use \texttt{ArrayGet()} to obtain an Fortran90 pointer that references the PET-local array.

\begin{verbatim}
call ESMF_ArrayGet(array, localDe=0, farrayPtr=farrayPtr, rc=rc)
print *, farrayPtr
\end{verbatim}

Finally the Array object must be destroyed. The PET-local memory of the \texttt{farrayE}s will remain in user control and will not be altered by \texttt{ArrayDestroy()}.

\begin{verbatim}
call ESMF_ArrayDestroy(array, rc=rc)
\end{verbatim}

Since the memory allocation for each \texttt{farrayE} is automatic there is nothing more to do. The interaction between \texttt{farrayE} and the Array class is representative also for the two other cases \texttt{farrayA} and \texttt{farrayP}. The only difference is in the handling of memory allocations.

\begin{verbatim}
allocation(farrayA(10,10)) ! user controlled allocation
farrayA = 23.67d0 ! initialize to some value
array = ESMF_ArrayCreate(farray=farrayA, distgrid=distgrid, rc=rc)
print *, farrayA ! print PET-local farrayA directly
\end{verbatim}

\begin{verbatim}
call ESMF_ArrayGet(array, localDe=0, farrayPtr=farrayPtr, rc=rc)! obtain array pointer
print *, farrayPtr ! print PET-local piece of Array through pointer
call ESMF_ArrayDestroy(array, rc=rc) ! destroy the Array
\end{verbatim}

\begin{verbatim}
deallocate(farrayA) ! user controlled de-allocation
\end{verbatim}

The \texttt{farrayP} case is identical.

\begin{verbatim}
allocation(farrayP(10,10)) ! user controlled allocation
farrayP = 56.81d0 ! initialize to some value
array = ESMF_ArrayCreate(farray=farrayP, distgrid=distgrid, rc=rc)
\end{verbatim}
20.2.2 Array from native Fortran90 array with elements for halo

The example of the previous section showed how easy it is to create an Array object from existing PET-local Fortran90 arrays. The example did, however, not define any halos around the DE-local regions. The following code demonstrates how an Array object with space for a halo can be set up.

```fortran
program ESMF_ArrayFarrayHaloEx
  use ESMF_Mod
  implicit none
  The allocatable array farrayA will be used to provide the PET-local Fortran90 array for this example.
  
  ! local variables
  real(ESMF_KIND_R8), allocatable :: farrayA(:,,:) ! allocatable F90 array
  real(ESMF_KIND_R8), pointer :: farrayPtr(:,,:) ! matching F90 array pointer
  type(ESMF_DistGrid) :: distgrid ! DistGrid object
  type(ESMF_Array) :: array ! Array object
  integer :: rc, i, j
  real :: localSum
  
  call ESMF_Initialize(rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
  
  The Array is to cover the exact same index space as in the previous example. Furthermore decomposition and distribution are also kept the same. Hence the same DistGrid object will be created and it is expected to execute this example with 4 PETs.
  
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

To wrap things up the DistGrid object is destroyed and ESMF can be finalized.

```fortran
  call ESMF_DistGridDestroy(distgrid, rc=rc) ! destroy the DistGrid
  call ESMF_Finalize(rc=rc)
  end program
```
This DistGrid describes a 40 x 10 index space that will be decomposed into 4 DEs when executed on 4 PETs, associating 1 DE per PET. Each DE-local exclusive region contains 10 x 10 elements. The DistGrid also stores and provides information about the relationship between DEs in index space, however, DistGrid does not contain information about halos. Arrays contain halo information and it is possible to create multiple Arrays covering the same index space with identical decomposition and distribution using the same DistGrid object, while defining different, Array-specific halo regions.

The extra memory required to cover the halo in the Array object must be taken into account when allocating the PET-local farrayA arrays. For a halo of 2 elements in each direction the following allocation will suffice.

\[
\text{allocate(farrayA(14,14)) ! Fortran90 array with halo: 14 = 10 + 2 * 2}
\]

The farrayA can now be used to create an Array object with enough space for a two element halo in each direction. The Array creation method checks for each PET that the local Fortran90 array can accommodate the requested regions. The default behavior of ArrayCreate() is to center the exclusive region within the total region. Consequently the following call will provide the 2 extra elements on each side of the exclusive 10 x 10 region without having to specify any additional arguments.

\[
\text{array = ESMF_ArrayCreate(farray=farrayA, distgrid=distgrid, rc=rc)}
\]

The exclusive Array region on each PET can be accessed through a suitable Fortran90 array pointer. See section 20.2.6 for more details on Array regions.

\[
\text{call ESMF_ArrayGet(array, localDe=0, farrayPtr=farrayPtr, rc=rc)}
\]

Following Array bounds convention, which by default puts the beginning of the exclusive region at (1, 1, ...), the following loop will add up the values of the local exclusive region for each DE, regardless of how the bounds were chosen for the original PET-local farrayA arrays.

\[
\text{localSum = 0.}
\text{do j=1, 10}
\text{do i=1, 10}
\text{\hspace{1cm} localSum = localSum + farrayPtr(i, j)}
\text{\hspace{1cm} enddo}
\text{\hspace{1cm} enddo}
\]

Elements with \(i\) or \(j\) in the [-1,0] or [11,12] ranges are located outside the exclusive region and may be used to define extra computational points or halo operations.

Cleanup and shut down ESMF.

\[
\text{call ESMF_ArrayDestroy(array, rc=rc)}
\text{deallocate(farrayA)}
\text{call ESMF_DistGridDestroy(distgrid, rc=rc)}
\text{call ESMF_Finalize(rc=rc)}
\]

end program
20.2.3 Array from ESMF_LocalArray

Alternative to the direct usage of Fortran90 arrays during Array creation it is also possible to first create an ESMF_LocalArray and create the Array from it. While this may seem more burdensome for the 1 DE per PET cases discussed in the previous sections it allows a straightforward generalization to the multiple DE per PET case. The following example first recaptures the previous example using an ESMF_LocalArray and then expands to the multiple DE per PET case.

```fortran
program ESMF_ArrayLocalArrayEx
  use ESMF_Mod

  implicit none

  The current ESMF_LocalArray interface requires Fortran90 arrays to be defined with pointer attribute.

  ! local variables
  real(ESMF_KIND_R8), pointer :: farrayP(:, :) ! F90 array pointer
  real(ESMF_KIND_R8), pointer :: farrayPtr(:, :) ! matching F90 array pointer
  type(ESMF_LocalArray) :: larray ! ESMF_LocalArray object
  type(ESMF_LocalArray) :: larrayRef ! ESMF_LocalArray object
  type(ESMF_DistGrid) :: distgrid ! DistGrid object
  type(ESMF_Array) :: array ! Array object
  integer :: rc, i, j, de
  real :: localSum
  type(ESMF_LocalArray), allocatable :: larrayList(:) ! ESMF_LocalArray object list
  type(ESMF_LocalArray), allocatable :: larrayRefList(:) ! ESMF_LocalArray object list
  type(ESMF_VM) :: vm
  integer :: localPet, petCount

  call ESMF_Initialize(vm=vm, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
  call ESMF_VMGet(vm, localPet=localPet, petCount=petCount, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)

  if (petCount /= 4) goto 10 ! TODO: use EXAMPLES_MULTI_ONLY once available

  DistGrid and array allocation remains unchanged.

  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)

  allocate(farrayP(14,14)) ! allocate Fortran90 array on each PET with halo

  Now instead of directly creating an Array object using the PET-local farrayPs an ESMF_LocalArray object will be created on each PET.

  larray = ESMF_LocalArrayCreate(farrayP, ESMF_DATA_REF, rc=rc)

  The Array object can now be created from larray. The Array creation method checks for each PET that the LocalArray can accommodate the requested regions.
```

149
array = ESMF_ArrayCreate(larrayList=('/larray/'), distgrid=distgrid, rc=rc)

Once created there is no difference in how the Array object can be used. The exclusive Array region on each PET can be accessed through a suitable Fortran90 array pointer as before.

call ESMF_ArrayGet(array, localDe=0, farrayPtr=farrayPtr, rc=rc)

Alternatively it is also possible (independent of how the Array object was created) to obtain the reference to the array allocation held by Array in form of an ESMF_LocalArray object. The farrayPtr can then be extracted using LocalArray methods.

call ESMF_ArrayGet(array, localDe=0, larray=larrayRef, rc=rc)

call ESMF_LocalArrayGet(larrayRef, farrayPtr, rc=rc)

Either way the farrayPtr reference can be used now to add up the values of the local exclusive region for each DE. The following loop works regardless of how the bounds were chosen for the original PET-local farrayP arrays and consequently the PET-local larray objects.

localSum = 0.
do j=1, 10
   do i=1, 10
      localSum = localSum + farrayPtr(i, j)
   enddo
enddo
print *, "localSum=", localSum

Cleanup.

call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_LocalArrayDestroy(larray, rc=rc)
deallocate(farrayP) ! use the pointer that was used in allocate statement
call ESMF_DistGridDestroy(distgrid, rc=rc)

While the usage of LocalArrays is unnecessarily cumbersome for 1 DE per PET Arrays, it provides a straight forward path for extenting the interfaces to multiple DEs per PET. In the following example a 8 x 8 index space will be decomposed into 2 x 4 = 8 DEs. The situation is captured by the following DistGrid object.

distgrid = ESMF_DistGridCreate(minIndex= (/1,1/), maxIndex= (/8,8/), regDecomp= (/2,4/), rc=rc)

The distgrid object created in this manner will contain 8 DEs no matter how many PETs are available during execution. Assuming an execution on 4 PETs will result in the following distribution of the decomposition.
Obviously each PET is associated with 2 DEs. Each PET must allocate enough space for all its DEs. This is done by allocating as many DE-local arrays as there are DEs on the PET. The reference to these array allocations is passed into ArrayCreate via a LocalArray list argument that holds as many elements as there are DEs on the PET. Here each PET must allocate for two DEs.

```
allocate(larrayList(2)) ! 2 DEs per PET
allocate(farrayP(4, 2)) ! without halo each DE is of size 4 x 2
farrayP = 123.456d0
larrayList(1) = ESMF_LocalArrayCreate(farrayP, ESMF_DATA_REF, rc=rc) ! 1st DE
allocate(farrayP(4, 2)) ! without halo each DE is of size 4 x 2
farrayP = 456.789d0
larrayList(2) = ESMF_LocalArrayCreate(farrayP, ESMF_DATA_REF, rc=rc) ! 2nd DE
```

Notice that it is perfectly fine to re-use farrayP for all allocations of DE-local Fortran90 arrays. The allocated memory can be deallocated at the end using the array pointer contained in the larrayList.

With this information an Array object can be created. The distgrid object indicates 2 DEs for each PET and ArrayCreate() expects to find two LocalArray elements in larrayList.

```
array = ESMF_ArrayCreate(larrayList=larrayList, distgrid=distgrid, rc=rc)
```

Usage of a LocalArray list is the only way to provide a list of variable length of Fortran90 array allocations to ArrayCreate() for each PET. The array object created by the above call is an ESMF distributed object. As such it must follow the ESMF convention that requires that the call to ESMF_ArrayCreate() must be issued in unison by all PETs of the current context. Each PET only calls ArrayCreate() once, even if there are multiple DEs per PET. The ArrayGet() method provides access to the list of LocalArrays on each PET.

```
allocate(larrayRefList(2))
call ESMF_ArrayGet(array, larrayList=larrayRefList, rc=rc)
```

Finally, access to the actual Fortran90 pointers is done on a per DE basis. Generally each PET will loop over its DEs.

```
do de=1, 2
```
call ESMF_LocalArrayGet(larrayRefList(de), farrayPtr, rc=rc)
localSum = 0.
do j=1, 2
    do i=1, 4
        localSum = localSum + farrayPtr(i, j)
    enddo
enddo
print *, "localSum=", localSum
enddo

Note: If the VM associates multiple PEs with a PET the application writer may decide to use OpenMP loop parallelization on the de loop.
Cleanup requires that the PET-local deallocations are done before the pointers to the actual Fortran90 arrays are lost.
Notice that larrayList is used to obtain the pointers used in the deallocate statement. Pointers obtained from the larrayRefList, while pointing to the same data, cannot be used to deallocated the array allocations!

do de=1, 2
    call ESMF_LocalArrayGet(larrayList(de), farrayPtr, rc=rc)
deallocate(farrayPtr)
call ESMF_LocalArrayDestroy(larrayList(de), rc=rc)
enddo
deallocate(larrayList)
deallocate(larrayRefList)
call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_DistGridDestroy(distgrid, rc=rc)

With that ESMF can be shut down cleanly.

call ESMF_Finalize(rc=rc)

end program

20.2.4 Array creation with automatic memory allocation
The examples of the previous sections made the user responsible for providing memory allocations for the PET-local regions of the Array object. The user was able to use any of the Fortran90 array methods or go through the ESMF_LocalArray interfaces to obtain memory allocations before passing them into ArrayCreate(). Alternatively, users may wish for ESMF to handle memory allocation of an Array object directly. The following example shows the interfaces that are available to the user to do just this.
To create an ESMF_Array object without providing an existing Fortran90 array or ESMF_LocalArray the type, kind and rank (tkr) of the Array must be specified in form of an ESMF_ArraySpec argument. Here a 2D Array of double precision real numbers is to be created:
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)

Further an ESMF_DistGrid argument must be constructed that holds information about the entire domain (patchwork) and the decomposition into DE-local exclusive regions. The following line creates a DistGrid for a 5x5 global LR domain that is decomposed into 2 x 3 = 6 DEs.
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
regDecomp=(/2,3/), rc=rc)

This is enough information to create a Array object with default settings.
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)

The array object created by the above call is an ESMF distributed object. As such it must follow the ESMF convention that requires that the call to ESMF_ArrayCreate() must be issued in unison by all PETs of the current context.
The index space covered by the Array object and the decomposition into DE-local exclusive regions, as it is described by the DistGrid object, is illustrated in the following diagram. Each asterix (*) represents a single element.

![Diagram of index space and DE decomposition](https://example.com/diagram.png)

20.2.5 Native language memory access – the most general way

The exact decomposition of the index space covered by the array object into DEs is contained in the distgrid object. Further, the layout of the DEs across the PETs of the component is stored in the delayout contained within the distgrid object. In the above example a default DELayout was created during the ESMF_DistGridCreate() call (see the refDoc / proposal for ESMF_DELayout and ESMF_DistGrid for details). In order to use the array object it is necessary to know the local DEs located on each calling PET.

call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)
allocate(localDeList(localDeCount))
call ESMF_ArrayGet(array, localDeList=localDeList, rc=rc)

In general it must be assumed that there may be multiple DEs associated with the calling PET, i.e. localDeCount >= 1. The situation where there is exactly one DE for each PET, i.e. localDeCount = 1 on every PET, is merely a special case of the more general formulation.
Consequently, in order to gain access to the DE-local memory segments that have been allocated on each PET by the
ArrayCreate() call the Array must be queried for a list of LocalArray objects, each element corresponding to
one PET-local DE.

```fortran
allocate(larrayList(localDeCount))
call ESMF_ArrayGet(array, larrayList=larrayList, rc=rc)
```

Now each PET can loop through its local list of DEs and access the associated memory through a suitable Fortran90
pointer. In the current example the native pointer `myF90Array` must be declared as

```fortran
real(ESMF_KIND_R8), pointer:: myF90Array(:,:)
```

in order to match the `arrayspec` that was used to create the array object. The following loop uses the native
language access to initialize the entire memory chunks of all PET-local DEs to 0 using Fortran90 array syntax.

```fortran
do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myF90Array, ESMF_DATA_REF, rc=rc)
  myF90Array = 0.
enddo
```

### 20.2.6 Regions and default bounds

Each ESMF_Array object is decomposed into DEs as specified by the associated ESMF_DistGrid object. Each
piece of this decomposition, i.e. each DE, holds a chunk of the Array data in its own local piece of memory. The
details of the Array decomposition are described in the following paragraphs.

At the center of the Array decomposition is the ESMF_DistGrid class. The DistGrid object specified during Array
creation contains three essential pieces of information:

- The extent and topology of the global domain covered by the Array object in terms of indexed elements. The
total extent may be a composition or patchwork of smaller logically rectangular (LR) domain pieces or patches.

- The decomposition of the entire domain into "element exclusive" DE-local LR chunks. *Element exclusive*
means that there is no element overlap between DE-local chunks. This, however, does not exclude degeneracies
between staggering locations for certain topologies (e.g. bipolar).

- The layout of DEs over the available PETs and thus the distribution of the Array data.

Each element of an Array is associated with a *single* DE. The union of elements associated with a DE, as defined by
the DistGrid above, corresponds to a LR chunk of index space, called the *exclusive region* of the DE.

There is a hierarchy of four regions that can be identified for each DE in an Array object. Their definition and
relationship to each other is as follows:

- **Interior Region**: Region that only contains local elements that are *not* mapped into the halo of any other DE.
The shape and size of this region for a particular DE depends non-locally on the halos defined by other DEs and
may change during computation as halo operations are precomputed and released. Knowledge of the interior
elements may be used to improve performance by overlapping communications with ongoing computation for
a DE.

- **Exclusive Region**: Elements for which a DE claims exclusive ownership. Practically this means that the DE
will be the sole source for these elements in halo and reduce operations. There are exceptions to this for certain
staggering locations in some topologies. These cases remain well-defined with the information available through
the associated DistGrid. The exclusive region includes all elements of the interior region.

- **Computational Region**: Region that can be set arbitrarily within the bounds of the total region (defined next).
The typical use of the computation region is to define bounds that only include elements that are updated by a
DE-local computation kernel. The computational region does not need to include all exclusive elements and it
may also contain elements that lie outside the exclusive region.
- **Total (Memory) Region:** Total of all DE-locally allocated elements. The size and shape of the total memory region must accommodate the union of exclusive and computational region but may contain additional elements. Elements outside the exclusive region may overlap with the exclusive region of another DE which makes them potential receivers for Array halo operations. Elements outside the exclusive region that do not overlap with the exclusive region of another DE can be used to set boundary conditions and/or serve as extra memory padding.

\[
\begin{align*}
\text{computationalLWidth}(:) &= \text{exclusiveLBound}(:) - \text{computationalLBound}(:) \\
\text{computationalUWidth}(:) &= \text{computationalUBound}(:) - \text{exclusiveLBound}(:)
\end{align*}
\]

and

\[
\begin{align*}
\text{totalLWidth}( :) &= \text{computationalLBound}(:) - \text{totalLBound}( :) \\
\text{totalUWidth}( :) &= \text{totalUBound}( :) - \text{computationalUBound}( :) \\
\end{align*}
\]

The exclusive region is determined during Array creation by the DistGrid argument. Optional arguments may be used to specify the computational region when the Array is created, by default it will be set equal to the exclusive region. The total region, i.e. the actual memory allocation for each DE, is also determined during Array creation. When
creating the Array object from existing Fortran90 arrays the total region is set equal to the memory provided by the
Fortran90 arrays. Otherwise the default is to allocate as much memory as is needed to accommodate the union of the
DE-local exclusive and computational region. Finally it is also possible to use optional arguments to the ArrayCreate() call to specify the total region of the object explicitly.

The ESMF_ArrayCreate() call checks that the input parameters are consistent and will result in an Array that
fulfills all of the above mentioned requirements for its DE-local regions.

Once an Array object has been created the exclusive and total regions are fixed. The computational region, however,
may be adjusted within the limits of the total region using the ArraySet() call.

The interior region is very different from the other regions in that it cannot be specified. The interior region for each
DE is a consequence of the choices made for the other regions collectively across all DEs into which an Array object
is decomposed. An Array object can be queried for its DE-local interior regions as to offer additional information to
the user necessary to write more efficient code. See section ?? (not yet implemented) for more details.

By default the bounds of each DE-local total region are defined as to put the start of the DE-local exclusive region at
the “origin” of the local index space, i.e. at \((1, 1, \ldots, 1)\). With that definition the following loop will access
each element of the DE-local memory segment for each PET-local DE of the Array object used in the previous sections
and print its content.

```fortran
  do de=1, localDeCount
    call ESMF_LocalArrayGet(larrayList(de), myF90Array, ESMF_DATA_REF, rc=rc)
    do i=1, size(myF90Array, 1)
      do j=1, size(myF90Array, 2)
        print *, "PET-local DE=", de, ": array("i",",","j")="", myF90Array(i,j)
      enddo
    enddo
  enddo
```

### 20.2.7 Array bounds

The loop over Array elements at the end of the last section only works correctly because of the default definition of the computational and total regions used in the example. In general, without such specific knowledge about an Array object, it is necessary to use a more formal approach to access its regions with DE-local indices.

The DE-local exclusive region takes a central role in the definition of Array bounds. Even as the computational region
may adjust during the course of execution the exclusive region remains unchanged. Furthermore the exclusive region
is identical for all stagger locations (discussed in a later section) and as such provides a unique reference frame for the
index space of all Arrays associated with the same DistGrid.

There is a choice between two indexing options that needs to be made during Array creation. By default each DE-local
exclusive region starts at \((1, 1, \ldots, 1)\). However, for some computational kernels it may be more convenient
to choose the index bounds of the DE-local exclusive regions to match the index space coordinates as they are defined
in the corresponding DistGrid object. The second option is only available if the DistGrid object does not contain any
non-contiguous decompositions (such as cyclically decomposed dimensions).

The following example code demonstrates the safe way of dereferencing the DE-local exclusive regions of the previously created array object.

```fortran
allocate(exclusiveUBound(2, localDeCount)) ! dimCount=2
allocate(exclusiveLBound(2, localDeCount)) ! dimCount=2

call ESMF_ArrayGet(array, indexflag=indexflag, &
  exclusiveLBound=exclusiveLBound, exclusiveUBound=exclusiveUBound, rc=rc)
if (indexflag == ESMF_INDEX_DELOCAL) then
  ! this is the default
  do de=1, localDeCount
    call ESMF_LocalArrayGet(larrayList(de), myF90Array, ESMF_DATA_REF, rc=rc)
    do i=1, exclusiveUBound(1, de)
      do j=1, exclusiveUBound(2, de)
        ! print *, "DE-local exclusive region for PET-local DE=", de, &
```
Obviously the second branch of this simple code will work for either case, however, if a complex computational kernel was written assuming ESMF_INDEX_DELOCAL type bounds the second branch would simply be used to indicate the problem and bail out.

The advantage of the ESMF_INDEX_GLOBAL index option is that the Array bounds directly contain information on where the DE-local Array piece is located in a global index space sense. When the ESMF_INDEX_DELOCAL option is used the correspondence between local and global index space must be made by querying the associated DistGrid for the DE-local indexList arguments.

20.2.8 Computational region and extra elements for halo or padding

In the previous examples the computational region of array was chosen by default to be identical to the exclusive region defined by the DistGrid argument during Array creation. In the following the same arrayspec and distgrid objects as before will be used to create an Array but now a larger computational region shall be defined around each DE-local exclusive region. Furthermore, extra space will be defined around the computational region of each DE to accommodate a halo and/or serve as memory padding.

In this example the indexflag argument is set to ESMF_INDEX_GLOBAL indicating that the bounds of the exclusive region correspond to the index space coordinates as they are defined by the DistGrid object.

The same arrayspec and distgrid objects as before are used which also allows the reuse of the already allocated larrayList variable.

```fortran
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
computationalLWidth= (/0,3/), computationalUWidth= (/1,1/), &
totalLWidth= (/1,4/), totalUWidth= (/3,1/), &
indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

Obtain the larrayList on every PET.

```fortran
call ESMF_ArrayGet(array, larrayList=larrayList, rc=rc)
```

The bounds of DE 1 for array are shown in the following diagram to illustrate the situation. Notice that the totalLWidth and totalUWidth arguments in the ArrayCreate() call define the total region with respect to the exclusive region given for each DE by the distgrid argument.
When working with this array it is possible for the computational kernel to overstep the exclusive region for both read/write access (computational region) and potentially read-only access into the total region outside of the computational region, if a halo operation provides valid entries for these elements.

The Array object can be queried for absolute bounds:

```fortran
allocate(computationalLLBound(2, localDeCount)) ! dimCount=2
allocate(computationalUBound(2, localDeCount)) ! dimCount=2
allocate(totalLLBound(2, localDeCount)) ! dimCount=2
allocate(totalUBound(2, localDeCount)) ! dimCount=2
call ESMF_ArrayGet(array, exclusiveLBound=exclusiveLB, &
computationalLBound=computationalLBound, &
totalLBound=totalLBound, rc=rc)
```

or for the relative widths:

```fortran
allocate(computationalLLWidth(2, localDeCount)) ! dimCount=2
allocate(computationalUWidth(2, localDeCount)) ! dimCount=2
allocate(totalLLWidth(2, localDeCount)) ! dimCount=2
allocate(totalUWidth(2, localDeCount)) ! dimCount=2
call ESMF_ArrayGet(array, computationalLLWidth=computationalLLWidth, &
computationalUWidth=computationalUWidth, totalLLWidth=totalLLWidth, &
totalUWidth=totalUWidth, rc=rc)
```

Either way the dereferencing of Array data is centered around the DE-local exclusive region:

```fortran
do de=1, localDeCount
    call ESMF_LocalArrayGet(larrayList(de), myF90Array, ESMF_DATA_REF, rc=rc)
    ! initialize the DE-local array
    myF90Array = 0.1d0 * localDeList(de)
    ! first time through the total region of array
    ! print *, "myF90Array bounds for DE=", localDeList(de), lbound(myF90Array), &
    ! ubound(myF90Array)
    do j=exclusiveLBound(2, de), exclusiveUBound(2, de)
        do i=exclusiveLBound(1, de), exclusiveUBound(1, de)
            print *, "Excl region DE=", localDeList(de), ": array("",i",",",j",")="", &
            myF90Array(i,j)
        enddo
    enddo
enddo
do j=computationalLLBound(2, de), computationalUBound(2, de)
```
do i=computationalLBound(1, de), computationalUBound(1, de)
    ! print *, "Excl region DE=", localDeList(de), ": array("i","j")="
    myF90Array(i,j)
enddo
enddo
do j=totalLBound(2, de), totalUBound(2, de)
    do i=totalLBound(1, de), totalUBound(1, de)
        ! print *, "Total region DE=", localDeList(de), ": array("i","j")="
        myF90Array(i,j)
    enddo
enddo

! second time through the total region of array
    do j=exclusiveLBound(2, de)-totalLWidth(2, de), 
        exclusiveUBound(2, de)+totalUWidth(2, de)
    do i=exclusiveLBound(1, de)-totalLWidth(1, de), 
        exclusiveUBound(1, de)+totalUWidth(1, de)
        ! print *, "Excl region DE=", localDeList(de), ": array("i","j")="
        myF90Array(i,j)
    enddo
enddo
enddo

20.2.9 1D and 3D Arrays

All previous examples were written for the 2D case. There is, however, no restriction within the Array or DistGrid class that limits the dimensionality of Array objects beyond the language specific limitations (7D for Fortran).

In order to create an n-dimensional Array the rank indicated by both the arrayspec and the distgrid arguments specified during Array create must be equal to n. A 1D Array of double precision real data hence requires the following arrayspec.

    call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=1, rc=rc)

The index space covered by the Array and the decomposition description is provided to the Array create method by the distgrid argument. The index space in this example has 16 elements and covers the interval \([-10, 5]\). It is decomposed into as many DEs as there are PETs in the current context.

    distgrid1D = ESMF_DistGridCreate(minIndex=(/-10/), maxIndex=(/5/), 
        regDecomp=(/petCount/), rc=rc)

A 1D Array object with default regions can now be created.

    array1D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid1D, rc=rc)

The creation of a 3D Array proceeds analogous to the 1D case. The rank of the arrayspec must be changed to 3

    call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)

and an appropriate 3D DistGrid object must be created

    distgrid3D = ESMF_DistGridCreate(minIndex=(/1,1,1/), maxIndex=(/16,16,16/), 
        regDecomp=(/4,4,4/), rc=rc)
before an Array object can be created.

\[
\text{array3D} = \text{ESMF\_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid3D, rc=rc)}
\]

The \text{distgrid3D} object decomposes the 3-dimensional index space into \(4 \times 4 \times 4 = 64\) DEs. These DEs are laid out across the computational resources (PETS) of the current component according to a default DELayout that is created during the DistGrid create call. Notice that in the index space proposal a DELayout does not have a sense of dimensionality. The DELayout function is simply to map DEs to PETS. The DistGrid maps chunks of index space against DEs and thus its rank is equal to the number of index space dimensions.

The previously defined DistGrid and the derived Array object decompose the index space along all three dimension. It is, however, not a requirement that the decomposition be along all dimensions. An Array with the same 3D index space could as well be decomposed along just one or along two of the dimensions. The following example shows how for the same index space only the last two dimensions are decomposed while the first Array dimension has full extent on all DEs.

\[
\begin{align*}
\text{call ESMF\_ArrayDestroy(array3D, rc=rc)} \\
\text{call ESMF\_DistGridDestroy(distgrid3D, rc=rc)} \\
\text{distgrid3D} &= \text{ESMF\_DistGridCreate(minIndex=(/1,1,1/), maxIndex=(/16,16,16/), &} \\
&\quad \text{regDecomp=(/1,4,4/), rc=rc)} \\
\text{array3D} &= \text{ESMF\_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid3D, rc=rc)}
\end{align*}
\]

Finally, the definition and usage of the stagger location index as it was described in sections ?? and ?? for the 2D case applies without change to 1D, 3D or any other dimensionality. Connections defined in the DistGrid object may utilize the stagger location index in order to express characteristics of the index space topology. The concept is completely rank independent.

### 20.2.10 Working with Arrays of different rank

Assume a computational kernel that involves the \text{array3D} object as it was created at the end of the previous section. Assume further that the kernel also involves a 2D Array on a 16x16 index space where each point (j,k) was interacting with each (i,j,k) column of the 3D Array. An efficient formulation would require that the decomposition of the 2D Array must match that of the 3D Array and further the DELayout be identical. The following code shows how this can be accomplished.

\[
\begin{align*}
\text{call ESMF\_DistGridGet(distgrid3D, delayout=delayout, rc=rc)} & \text{ ! get DELayout} \\
\text{distgrid2D} &= \text{ESMF\_DistGridCreate(minIndex=(/1,1/), maxIndex=(/16,16/), &} \\
&\quad \text{regDecomp=(/4,4/), delayout=delayout, rc=rc)} \\
\text{call ESMF\_ArraySpecSet(arrayspec, typekind=ESMF\_TYPEKIND\_R8, rank=2, rc=rc)} \\
\text{array2D} &= \text{ESMF\_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid2D, rc=rc)}
\end{align*}
\]

Now the following kernel is sure to work with \text{array3D} and \text{array2D}.

\[
\begin{align*}
\text{call ESMF\_DELayoutGet(delayout, localDeCount=localDeCount, rc=rc)} & \text{ ! allocate (larrayList1(localDeCount))} \\
\text{call ESMF\_ArrayGet(array3D, larrayList=larrayList1, rc=rc)} & \text{ ! allocate (larrayList2(localDeCount))} \\
\text{call ESMF\_ArrayGet(array2D, larrayList=larrayList2, rc=rc)} \\
\text{do de=1, localDeCount} \\
\text{call ESMF\_LocalArrayGet(larrayList1(de), myF90Array3D, ESMF\_DATA\_REF, &} \\
&\quad \text{rc=rc)} \\
\text{myF90Array3D} &= 0.1d0 * \text{de} \text{ ! initialize} \\
\text{call ESMF\_LocalArrayGet(larrayList2(de), myF90Array2D, ESMF\_DATA\_REF, &} \\
&\quad \text{rc=rc)} \\
\text{myF90Array2D} &= 0.5d0 * \text{de} \text{ ! initialize}
\end{align*}
\]
\begin{verbatim}
do k=1, 4
  do j=1, 4
    dummySum = 0.d0
    do i=1, 16
      dummySum = dummySum + myF90Array3D(i,j,k) ! sum up the (j,k) column
    enddo
    dummySum = dummySum * myF90Array2D(j,k) ! multiply with local 2D element
    ! print *, "dummySum(",j,k,")=" , dummySum
  enddo
enddo
enddo
\end{verbatim}

20.2.11 Array and DistGrid rank – 2D+1 Arrays

Except for the special Array create interface that implements a copy from an existing Array object all other Array create interfaces require the specification of at least two arguments: farray and distgrid, larrayList and distgrid, or arrayspec and distgrid. In all these cases both required arguments contain a sense of dimensionality. The relationship between these two arguments deserves extra attention.

The first argument, farray, larrayList or arrayspec, determines the rank of the created Array object, i.e. the dimensionality of the actual data storage. The rank of a native language array, extracted from an Array object, is equal to the rank specified by either of these arguments. So is the rank that is returned by the ESMF_ArrayGet() call. The rank specification contained in the distgrid argument, which is of type ESMF_DistGrid, on the other hand has no affect on the rank of the Array. The dimCount specified by the DistGrid object, which may be equal, greater or less than the Array rank, determines the dimensionality of the decomposition.

While there is no constraint between DistGrid dimCount and Array rank, there is an important relationship between the two, resulting in the concept of index space dimensionality. Array dimensions can be arbitrarily mapped against DistGrid dimension, rendering them decomposed dimensions. The index space dimensionality is equal to the number of decomposed Array dimensions.

Array dimensions that are not mapped to DistGrid dimensions are the undistributed dimensions of the Array. They are not part of the index space. The mapping is specified during ESMF_ArrayCreate() via the the distgridToarrayMap argument. DistGrid dimensions that have not been associated with Array dimensions are replicating dimensions. The Array will be replicates across the DEs that lie along replication DistGrid dimensions.

Undistributed Array dimensions can be used to store multi-dimensional data for each Array index space element. A special purpose of undistributed dimensions is to store multiple data arrays in the same Array object. It is, for example, possible to store array1 and array2 of section ?? in a single Array object using one undistributed dimension of size 2. The same distgrid object as before can be used to create the Array.

\begin{verbatim}
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
\end{verbatim}

The rank in the arrayspec argument, however, must change from 2 to 3 in order to provide for the extra Array dimension.

\begin{verbatim}
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
\end{verbatim}

During Array creation with extra dimension(s) it is necessary to specify the bounds of these undistributed dimension(s). This requires two additional arguments, undistLBound and undistUBound, which are vectors in order to accommodate multiple undistributed dimensions. The other arguments remain unchanged and apply across all undistributed components.

The optional arguments used in the following call are identical to those used to create array1 of section ?? . This will set the total region and the stagger location of both undistributed components to be those of array1.

\begin{verbatim}
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
\end{verbatim}
This will create array with 2+1 dimensions. The 2D DistGrid is used to describe decomposition into DEs with 2 Array dimensions mapped to the DistGrid dimensions resulting in a 2D index space. The extra Array dimension provides storage for multiple 2D user data arrays that are kept in a single Array object. By default the distgrid dimensions are associated with the first Array dimensions in sequence. For the example above this means that the first 2 Array dimensions are decomposed according to the provided 2D DistGrid. The 3rd Array dimension does not have an associated DistGrid dimension, rendering it an undistributed Array dimension.

The optional arguments that were used to create array ensure that the total region is large enough to accommodate the arrays for undistributed component 1 and 2. The Array class provides a special Set() method that allows to individually address undistributed elements in an Array and set staggerLoc and vectorDim arguments.

call ESMF_ArraySet(array, tensorIndex=(/2/), staggerLoc=2, rc=rc)

Native language access to an Array with undistributed dimensions is in principle the same as without extra dimensions.

call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)
allocate(larrayList(localDeCount))
call ESMF_ArrayGet(array, larrayList=larrayList, rc=rc)

do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myF90Array3D, ESMF_DATA_REF, rc=rc)
  myF90Array3D = 0.0 ! initialize
  myF90Array3D(exclusiveLBound(1,de):exclusiveUBound(1,de),
  exclusiveLBound(2,de):exclusiveUBound(2,de), 1) = 5.1 ! dummy assignment
  myF90Array3D(exclusiveLBound(1,de):exclusiveUBound(1,de),
  exclusiveLBound(2,de):exclusiveUBound(2,de), 2) = 2.5 ! dummy assignment
endo
deallocate(larrayList)

For some applications the default association rules between DistGrid and Array dimensions may not satisfy the user’s needs. The optional distgridToArrayMap argument can be used during Array creation to explicitly specify the mapping between DistGrid and Array dimensions. To demonstrate this the following lines of code reproduce the above example but with rearranged dimensions. Here the distgridToArrayMap argument is a list with two elements corresponding to the DistGrid dimCount of 2. The first element indicates which Array dimension the first DistGrid dimension is mapped against. Here the 1st DistGrid dimension maps against the 3rd Array dimension and the 2nd DistGrid dimension maps against the 1st Array dimension. This leaves the 2nd Array dimension to be the extra and undistributed dimension in the resulting Array object.

call ESMF_ArrayDestroy(array, rc=rc)
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
distgridToArrayMap=(/3, 1/), totalLWidth=(/0,1/), totalUWidth=(/0,1/), &
undistLBound=(/1/), undistUBound=(/2/), rc=rc)
call ESMF_ArraySet(array, tensorIndex=(/1/), staggerLoc=1, rc=rc)
call ESMF_ArraySet(array, tensorIndex=(/2/), staggerLoc=2, rc=rc)
Operations on the Array object as a whole are unchanged by the different mapping of dimensions. When working with Arrays that contain explicitly mapped Array and DistGrid dimensions it is critical to know the order in which the entries of width and bound arguments that are associated with distributed Array dimensions are specified. The size of these arguments is equal to the DistGrid dimCount, because the maximum number of distributed Array dimensions is given by the dimensionality of the index space.

The order of dimensions in these arguments, however, is not that of the associated DistGrid. Instead each entry corresponds to the distributed Array dimensions in sequence. In the example above the entries in totalLWidth and totalUWidth correspond to Array dimensions 1 and 3 in this sequence.

The distgridToArrayMap argument optionally provided during Array create indicates how the DistGrid dimensions map to Array dimensions. The inverse mapping, i.e. Array to DistGrid dimensions, is just as important. The ESMF_ArrayGet() call offers both mappings as distgridToArrayMap and arrayToDistGridMap, respectively. The number of elements in arrayToDistGridMap is equal to the rank of the Array. Each element corresponds to an Array dimension and indicates the associated DistGrid dimension by an integer number. An entry of "0" in arrayToDistGridMap indicates that the corresponding Array dimension is undistributed.

Correct understanding about the association between Array and DistGrid dimensions becomes critical for correct data access into the Array.

allocate(arrayToDistGridMap(3)) ! arrayRank = 3
call ESMF_ArrayGet(array, arrayToDistGridMap=arrayToDistGridMap, &
   exclusiveLBound=exclusiveLBound, exclusiveUBound=exclusiveUBound, &
   localDeCount=localDeCount, rc=rc)
if (arrayToDistGridMap(2) /= 0) then ! check if extra dimension at expected index
   ! indicate problem and bail out
endif
! obtain larrayList for local DEs
allocate(larrayList(localDeCount))
call ESMF_ArrayGet(array, larrayList=larrayList, rc=rc)
do de=1, localDeCount
   call ESMF_LocalArrayGet(larrayList(de), myF90Array3D, ESMF_DATA_REF, rc=rc)
   myF90Array3D(exclusiveLBound(1,de):exclusiveUBound(1,de), &
      1, exclusiveLBound(2,de):exclusiveUBound(2,de)) = 10.5 ! dummy assignment
   myF90Array3D(exclusiveLBound(1,de):exclusiveUBound(1,de), &
      2, exclusiveLBound(2,de):exclusiveUBound(2,de)) = 23.3 ! dummy assignment
endoallocate(exclusiveLBound, exclusiveUBound)
deallocate(arrayToDistGridMap)
deallocate(larrayList)
call ESMF_ArrayDestroy(array, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)

20.2.12 Arrays with replicated dimensions

Thus far most examples demonstrated cases where the DistGrid dimCount was equal to the Array rank. The previous section introduced the concept of Array tensor dimensions when dimCount < rank. In this section dimCount and rank are assumed completely unconstrained and the relationship to distgridToArrayMap and arrayToDistGridMap will be discussed.

The Array class allows completely arbitrary mapping between Array and DistGrid dimensions. Most cases considered in the previous sections used the default mapping which assigns the DistGrid dimensions in sequence to the lower Array dimensions. Extra Array dimensions, if present, are considered non-distributed tensor dimensions for which the optional undistLBound and undistUBound arguments must be specified.

The optional distgridToArrayMap argument provides the option to override the default DistGrid to Array dimension mapping. The entries of the distgridToArrayMap array correspond to the DistGrid dimensions in sequence and assign a unique Array dimension to each DistGrid dimension. DistGrid and Array dimensions are indexed starting at 1 for the lowest dimension. A value of "0" in the distgridToArrayMap array indicates that the respective DistGrid dimension is not mapped against any Array dimension. What this means is that the Array will be replicated
along this DistGrid dimension. 
As a first example consider the case where a 1D Array

```fortran
    call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=1, rc=rc)
```

is created on the 2D DistGrid used during the previous section.

```fortran
    array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

Here the default DistGrid to Array dimension mapping is used which assigns the Array dimensions in sequence to the DistGrid dimensions starting with dimension "1". Extra DistGrid dimensions are considered replicator dimensions because the Array will be replicated along those dimensions. In the above example the 2nd DistGrid dimension will cause 1D Array pieces to be replicated along the DEs of the 2nd DistGrid dimension. Replication in the context of `ESMF_ArrayCreate()` does not mean that data values are communicated and replicated between different DEs, but it means that different DEs provide memory allocations for *identical* exclusive elements. Access to the data storage of an Array that has been replicated along DistGrid dimensions is the same as for Arrays without replication.

```fortran
    call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)
```

```fortran
allocate(larrayList(localDeCount))
allocate(localDeList(localDeCount))
call ESMF_ArrayGet(array, larrayList=larrayList, localDeList=localDeList, &
                     rc=rc)
```

The array object was created without additional padding which means that the bounds of the Fortran array pointer correspond to the bounds of the exclusive region. The following loop will cycle through all local DEs, print the DE number as well as the Fortran array pointer bounds. The bounds should be:

<table>
<thead>
<tr>
<th>DE</th>
<th>lbound</th>
<th>ubound</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
| 2    | 1      | 3      | --|--+ 1st replication set
| 4    | 1      | 3      | ---+ |
| 1    | 1      | 2      | ---+ |
| 3    | 1      | 2      | --|-- 2nd replication set
| 5    | 1      | 2      | ---+ |

```fortran
do de=1, localDeCount
call ESMF_LocalArrayGet(larrayList(de), myF90Array1D, ESMF_DATA_REF, &
                        rc=rc)

print *, "DE ",localDeList(de)," [", lbound(myF90Array1D), &
                        ubound(myF90Array1D),"]"
enddo
deallocate(larrayList)
deallocate(localDeList)
call ESMF_ArrayDestroy(array, rc=rc)
```
The Fortran array pointer in the above loop was of rank 1 because the Array object was of rank 1. However, the distgrid object associated with array is 2-dimensional! Consequently DistGrid based information queried from array will be 2D. The distgridToArrayMap and arrayToDistGridMap arrays provide the necessary mapping to correctly associate DistGrid based information with Array dimensions.

The next example creates a 2D Array

```fortran
    call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

on the previously used 2D DistGrid. By default, i.e. without the distgridToArrayMap argument, both DistGrid dimensions would be associated with the two Array dimensions. However, the distgridToArrayMap specified in the following call will only associate the second DistGrid dimension with the first Array dimension. This will render the first DistGrid dimension a replicator dimension and the second Array dimension a tensor dimension for which 1D undistLBound and undistUBound arguments must be supplied.

```fortran
    array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
                           distgridToArrayMap=(/0,1/), undistLBound=(/11/), undistUBound=(/14/), rc=rc)
```

```fortran
    call ESMF_ArrayDestroy(array, rc=rc)
```

Finally, the same arrayspec and distgrid arguments are used to create a 2D Array that is fully replicated in both dimensions of the DistGrid. Both Array dimensions are now tensor dimensions and both DistGrid dimensions are replicator dimensions.

```fortran
    array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
                           distgridToArrayMap=(/0,0/), undistLBound=(/11,21/), undistUBound=(/14,22/), &
                           rc=rc)
```

The result will be an Array with local lower bound (/11,21/) and upper bound (/14,22/) on all 6 DEs of the DistGrid.

```fortran
    call ESMF_ArrayDestroy(array, rc=rc)
```

```fortran
    call ESMF_DistGridDestroy(distgrid, rc=rc)
```

Replicated Arrays can also be created from existing local Fortran arrays. The following Fortran array allocation will provide a 3 x 10 array on each PET.

```fortran
    allocate(myFortranArray2D(3,10))
```

Assuming a petCount of 4 the following DistGrid defines a 2D index space that is distributed across the PETs along the first dimension.

```fortran
    distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

The following call creates an Array object on the above distgrid using the locally existing myFortranArray2D Fortran arrays. The difference compared to the case with automatic memory allocation is that instead of arrayspec the Fortran array is provided as argument. Furthermore, the undistLBound and undistUBound arguments can be omitted, defaulting into Array tensor dimension lower bound of 1 and an upper bound equal to the size of the respective Fortran array dimension.
array = ESMF_ArrayCreate(farray=myF90Array2D, distgrid=distgrid, &
distgridToArrayMap=(/0,2/), rc=rc)

The array object associates the 2nd DistGrid dimension with the 2nd Array dimension. The first DistGrid dimension is not associated with any Array dimension and will lead to replication of the Array along the DEs of this direction.

call ESMF_ArrayDestroy(array, rc=rc)

call ESMF_DistGridDestroy(distgrid, rc=rc)

20.2.13 Communication – Scatter and Gather

It is a common situation, particularly in legacy code, that an ESMF Array object must be filled with data originating from a large Fortran array stored on a single PET.

if (localPet == 0) then
    allocate(farray(10,20,30))
    do k=1, 30
        do j=1, 20
            do i=1, 10
                farray(i, j, k) = k*1000 + j*100 + i
            enddo
        enddo
    enddo
endif

distgrid = ESMF_DistGridCreate(minIndex=(/1,1,1/), maxIndex=(/10,20,30/), &
rc=rc)

call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=3, rc=rc)

array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)

The ESMF_ArrayScatter() method provides a convenient way of scattering array data from a single root PET across the DEs of an ESMF Array object.

call ESMF_ArrayScatter(array, farray=farray, rootPet=0, rc=rc)

if (localPet == 0) then
    deallocate(farray)
endif

The destination of the ArrayScatter() operation are all the DEs of a single patch. For multi-patch Arrays the destination patch can be specified. The shape of the scattered Fortran array must match the shape of the destination patch in the ESMF Array.

Gathering data decomposed and distributed across the DEs of an ESMF Array object into a single Fortran array on root PET is accomplished by calling ESMF_ArrayGather().
if (localPet == 3) then
    allocate(farray(10,20,30))
endif

call ESMF_ArrayGather(array, farray=farray, rootPet=3, rc=rc)

if (localPet == 3) then
    deallocate(farray)
endif

The source of the ArrayGather() operation are all the DEs of a single patch. For multi-patch Arrays the source patch can be specified. The shape of the gathered Fortran array must match the shape of the source patch in the ESMF Array.

The ESMF_ArrayScatter() operation allows to fill entire replicated Array objects with data coming from a single root PET.

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), rc=rc)

call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)

array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
    distgridToArrayMap=(/0,0/), undistLBound=(/11,21/), undistUBound=(/14,22/), &
    rc=rc)

The shape of the Fortran source array used in the Scatter() call must be that of the contracted Array, i.e. contracted DistGrid dimensions do not count. For the array just created this means that the source array on rootPet must be of shape 4 x 2.

if (localPet == 0) then
    allocate(myF90Array2D(4,2))
    do j=1,2
        do i=1,4
            myF90Array2D(i,j) = i * 100.d0 + j * 1.2345d0 ! initialize
        enddo
    enddo
endif

call ESMF_ArrayScatter(array, farray=myF90Array2D, rootPet=0, rc=rc)

if (localPet == 0) then
    deallocate(myF90Array2D)
endif

This will have filled each local 4 x 2 Array piece with the replicated data of myF90Array2D.

call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_DistGridDestroy(distgrid, rc=rc)
As a second example for the use of Scatter() and Gather() consider the following replicated Array created from existing local Fortran arrays.

```fortran
allocate(myF90Array2D(3,10))
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
array = ESMF_ArrayCreate(farray=myF90Array2D, distgrid=distgrid, &
                        distgridToArrayMap=(/0,2/), rc=rc)
```

The `array` object associates the 2nd DistGrid dimension with the 2nd Array dimension. The first DistGrid dimension is not associated with any Array dimension and will lead to replication of the Array along the DEs of this direction. Still, the local arrays that comprise the `array` object refer to independent pieces of memory and can be initialized independently.

```fortran
myF90Array2D = localPet ! initialize
```

However, the notion of replication becomes visible when an array of shape 3 x 10 on root PET 0 is scattered across the Array object.

```fortran
if (localPet == 0) then
    allocate(myF90Array2D2(5:7,11:20))
    do j=11,20
        do i=5,7
            myF90Array2D2(i,j) = i * 100.d0 + j * 1.2345d0 ! initialize
        enddo
    enddo
endif

call ESMF_ArrayScatter(array, farray=myF90Array2D2, rootPet=0, rc=rc)
```

The Array pieces on every DE will receive the same source data, resulting in a replication of data along DistGrid dimension 1. When the inverse operation, i.e. `ESMF_ArrayGather()` , is applied to a replicated Array an intrinsic ambiguity needs to be considered. ESMF defines the gathering of data of a replicated Array as the collection of data originating from the numerically higher DEs. This means that data in replicated elements associated with numerically lower DEs will be ignored during `ESMF_ArrayGather()` . For the current example this means that changing the Array contents on PET 1, which here corresponds to DE 1,

```fortran
if (localPet == 1) then
    myF90Array2D = real(1.2345, ESMF_KIND_R8)
endif
```

will not affect the result of

```fortran
allocate(myF90Array2D2(3,10))
myF90Array2D2 = 0.d0 ! initialize to a known value
call ESMF_ArrayGather(array, farray=myF90Array2D2, rootPet=0, rc=rc)
```
The result remains completely defined by the unmodified values of Array in DE 3, the numerically highest DE. However, overriding the DE-local Array piece on DE 3

```cpp
if (localPet==3) then
  myF90Array2D = real(5.4321, ESMF_KIND_R8)
endif
```

will change the outcome of

```cpp
call ESMF_ArrayGather(array, farray=myF90Array2D2, rootPet=0, rc=rc)
```

as expected.

```cpp
deallocate(myF90Array2D2)
call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

### 20.2.14 Communication – Redist

Arrays used in different models often cover the same index space region, however, the distribution of the Arrays may be different, e.g. the models run on exclusive sets of PETs. Even if the Arrays are defined on the same list of PETs the decomposition may be different.

```cpp
srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/4,1/), rc=rc)

dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/1,4/), rc=rc)
```

The number of elements covered by `srcDistgrid` is identical to the number of elements covered by `dstDistgrid`—in fact the index space regions covered by both DistGrid objects are congruent.

```cpp
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
	srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, rc=rc)
	dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

By construction `srcArray` and `dstArray` are of identical type and kind. Further the number of exclusive elements matches between both Arrays. These are the prerequisites for the application of an Array redistribution in default mode. In order to increase performance of the actual redistribution the communication pattern must be precomputed and stored.

```cpp
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```
The `redistHandle` can now be used repeatedly on the `srcArray`, `dstArray` pair to redistribute data from source to destination Array.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)
```

The use of the `redistHandle` is not restricted to `srcArray` and `dstArray`. The `redistHandle` can be applied to redistribute data between any Array pairs that are congruent to the Array pair used during precomputation. Arrays are congruent if they are defined on matching DistGrids and the shape of local array allocations match for all DEs.

The resources held by `redistHandle` need to be deallocated by the user code before the handle becomes inaccessible.

```
call ESMF_ArrayRedistRelease(routehandle=redistHandle, rc=rc)
```

In default mode, i.e. without providing the optional `srcToDstTransposeMap` argument, `ESMF_ArrayRedistStore()` does not require equal number of dimensions in source and destination Array. Only the total number of elements must match.

Specifying `srcToDstTransposeMap` switches `ESMF_ArrayRedistStore()` into transpose mode. In this mode each dimension of `srcArray` is uniquely associated with a dimension in `dstArray`. The sizes of associated dimensions must match for each pair.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/20,10/), rc=rc)
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

This `dstArray` object covers a 20 x 10 index space while the `srcArray`, defined further up, covers a 10 x 20 index space. Setting `srcToDstTransposeMap = (/2,1/)` will associate the first and second dimension of `srcArray` with the second and first dimension of `dstArray`, respectively. This corresponds to a transpose of dimensions. Since the decomposition and distribution of dimensions may be different for source and destination redistribution may occur at the same time.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, srcToDstTransposeMap=(/2,1/), rc=rc)
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)
```

The transpose mode of `ESMF_ArrayRedist()` is not limited to distributed dimensions of Arrays. The `srcToDstTransposeMap` argument can be used to transpose undistributed dimensions in the same manner. Furthermore transposing distributed and undistributed dimensions between Arrays is also supported.

The `srcArray` used in the following examples is of rank 4 with 2 distributed and 2 undistributed dimensions. The distributed dimensions are the two first dimensions of the Array and are distributed according to the `srcDistgrid` which describes a total index space region of 100 x 200 elements. The last two Array dimensions are undistributed dimensions of size 2 and 3, respectively.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)
srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/100,200/), &
rc=rc)
```
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
undistLBound=(/1,1/), undistUBound=(/2,3/), rc=rc)

The first dstArray to consider is defined on a DistGrid that also describes a 100 x 200 index space region. The distribution indicated by dstDistgrid may be different from the source distribution. Again the first two Array dimensions are associated with the DistGrid dimensions in sequence. Furthermore, the last two Array dimensions are undistributed dimensions, however, the sizes are 3 and 2, respectively.

dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/100,200/), &
rc=rc)

dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
undistLBound=(/1,1/), undistUBound=(/3,2/), rc=rc)

The desired mapping between srcArray and dstArray dimensions is expressed by srcToDstTransposeMap = (/1,2,4,3/), transposing only the two undistributed dimensions.

call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, srcToDstTransposeMap=(/1,2,4,3/), rc=rc)

call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)

Next consider a dstArray that is defined on the same dstDistgrid, but with a different order of Array dimensions. The desired order is specified during Array creation using the argument distgridToArrayMap = (/2,3/). This map associates the first and second DistGrid dimensions with the second and third Array dimensions, respectively, leaving Array dimensions one and four undistributed.

dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
distgridToArrayMap=(/2,3/), undistLBound=(/1,1/), undistUBound=(/3,2/), &
rc=rc)

Again the sizes of the undistributed dimensions are chosen in reverse order compared to srcArray. The desired transpose mapping in this case will be srcToDstTransposeMap = (/2,3,4,1/).

call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, srcToDstTransposeMap=(/2,3,4,1/), rc=rc)

call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)

Finally consider the case where dstArray is constructed on a 200 x 3 index space and where the undistributed dimensions are of size 100 and 2.

dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/200,3/), &
rc=rc)

dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
undistLBound=(/1,1/), undistUBound=(/100,2/), rc=rc)
By construction `srcArray` and `dstArray` hold the same number of elements, albeit in a very different layout. Nevertheless, with a `srcToDstTransposeMap` that maps matching dimensions from source to destination an Array redistribution becomes a well defined operation between `srcArray` and `dstArray`.

```call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, srcToDstTransposeMap=(/3,1,4,2/), rc=rc)
```

```call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)
```

The default mode of Array redistribution, i.e. without providing a `srcToDstTransposeMap` to `ESMF_ArrayRedistStore()`, also supports undistributed Array dimensions. The requirement in this case is that the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, be the same for source and destination Array. In this mode the number of undistributed dimensions need not match between source and destination.

```call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)
```

```srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
regDecomp=(/4,1/), rc=rc)
```

```srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
undistLBound=(/1,1/), undistUBound=(/2,4/), rc=rc)
```

```dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
regDecomp=(/1,4/), rc=rc)
```

```dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
distgridToArrayMap=(/2,3/), undistLBound=(/1,1/), undistUBound=(/2,4/), &
rc=rc)
```

Both `srcArray` and `dstArray` have two undistributed dimensions and a total count of undistributed elements of $2 \times 4 = 8$.

The Array redistribution operation is defined in terms of sequentialized undistributed dimensions. In the above case this means that a unique sequence index will be assigned to each of the 8 undistributed elements. The sequence indices will be 1, 2, ..., 8, where sequence index 1 is assigned to the first element in the first (i.e. fastest varying in memory) undistributed dimension. The following undistributed elements are labeled in consecutive order as they are stored in memory.

```call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)
```

The redistribution operation by default applies the identity operation between the elements of undistributed dimensions. This means that source element with sequence index 1 will be mapped against destination element with sequence index 1 and so forth. Because of the way source and destination Arrays in the current example were constructed this corresponds to a mapping of dimensions 3 and 4 on `srcArray` to dimensions 1 and 4 on `dstArray`, respectively.

```call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
routehandle=redistHandle, rc=rc)
```
Array redistribution does not require the same number of undistributed dimensions in source and destination Array, merely the total number of undistributed elements must match.

```fortran
    call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)

dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
    distgridToArrayMap=(/1,3/), undistLBound=(/11/), undistUBound=(/18/), &
    rc=rc)
```

This `dstArray` object only has a single undistributed dimension, while the `srcArray`, defined further back, has two undistributed dimensions. However, the total undistributed element count for both Arrays is 8.

```fortran
    call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
        routehandle=redistHandle, rc=rc)
```

In this case the default identity operation between the elements of undistributed dimensions corresponds to a merging of dimensions 3 and 4 on `srcArray` into dimension 2 on `dstArray`.

```fortran
    call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
        routehandle=redistHandle, rc=rc)
```

### 20.2.15 Communication – SparseMatMul

Sparse matrix multiplication is a fundamental Array communication method. One frequently used application of this method is the interpolation between pairs of Arrays. The principle is this: the value of each element in the exclusive region of the destination Array is expressed as a linear combination of potentially all the exclusive elements of the source Array. Naturally most of the coefficients of these linear combinations will be zero and it is more efficient to store explicit information about the non-zero elements than to keep track of all the coefficients.

There is a choice to be made with respect to the format in which to store the information about the non-zero elements. One option is to store the value of each coefficient together with the corresponding destination element index and source element index. Destination and source indices could be expressed in terms of the corresponding DistGrid patch index together with the coordinate tuple within the patch. While this format may be the most natural way to express elements in the source and destination Array, it has two major drawbacks. First the coordinate tuple is `dimCount` specific and second the format is extremely bulky. For 2D source and destination Arrays it would require 6 integers to store the source and destination element information for each non-zero coefficient and matters get worse for higher dimensions.

Both problems can be circumvented by interpreting source and destination Arrays as sequentialized strings or vectors of elements. This is done by assigning a unique sequence index to each exclusive element in both Arrays. With that the operation of updating the elements in the destination Array as linear combinations of source Array elements takes the form of a sparse matrix multiplication.

The default sequence index rule assigns index 1 to the `minIndex` corner element of the first patch of the DistGrid on which the Array is defined. It then increments the sequence index by 1 for each element running through the DistGrid dimensions by order. The index space position of the DistGrid patches does not affect the sequence labeling of elements. The default sequence indices for

```fortran
    srcDistgrid = ESMF_DistGridCreate(minIndex=(/-1,0/), maxIndex=(/1,3/), rc=rc)
```

for each element are:

```
    -------------------------------------> 2nd dim
    |                                    |
```
The assigned sequence indices are decomposition and distribution invariant by construction. Furthermore, when an Array is created with extra elements per DE on a DistGrid the sequence indices (which only cover the exclusive elements) remain unchanged.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

```
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
rc=rc)
```

The extra padding of 1 element in each direction around the exclusive elements on each DE are "invisible" to the Array spare matrix multiplication method. These extra elements are either updated by the computational kernel or by Array halo operations (not yet implemented!).

An alternative way to assign sequence indices to all the elements in the patches covered by a DistGrid object is to use a special `ESMF_DistGridCreate()` call. This call has been specifically designed for 1D cases with arbitrary, user-supplied sequence indices.

```
seqIndexList(1) = localPet*10
seqIndexList(2) = localPet*10 + 1
dstDistgrid = ESMF_DistGridCreate(arbSeqIndexList=seqIndexList, rc=rc)
```

This call to `ESMF_DistGridCreate()` is collective across the current VM. The `arbSeqIndexList` argument specifies the PET-local arbitrary sequence indices that need to be covered by the local DE. The resulting DistGrid has one local DE per PET which covers the entire PET-local index range. The user supplied sequence indices must be unique, but the sequence may be interrupted. The four DEs of `dstDistgrid` have the following local 1D index space coordinates (given between "()") and sequence indices:

<table>
<thead>
<tr>
<th>covered by DE 0 on PET 0</th>
<th>covered by DE 1 on PET 1</th>
<th>covered by DE 2 on PET 2</th>
<th>covered by DE 3 on PET 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) : 0</td>
<td>(1) : 10</td>
<td>(1) : 20</td>
<td>(1) : 30</td>
</tr>
<tr>
<td>(2) : 1</td>
<td>(2) : 11</td>
<td>(2) : 21</td>
<td>(2) : 31</td>
</tr>
</tbody>
</table>

Again the DistGrid object provides the sequence index labeling for the exclusive elements of an Array created on the DistGrid regardless of extra, non-exclusive elements.
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)

With the definition of sequence indices, either by the default rule or as user provided arbitrary sequence indices, it is now possible to uniquely identify each exclusive element in the source and destination Array by a single integer number. Specifying a pair of source and destination elements takes two integer number regardless of the number of dimensions.

The information required to carry out a sparse matrix multiplication are the pair of source and destination sequence indices and the associated multiplication factor for each pair. ESMF requires this information in form of two Fortran arrays. The factors are stored in a 1D array of the appropriate type and kind, e.g. `real(ESMF_KIND_R8)::factorList(:,).` Array sparse matrix multiplications are only supported between Arrays of the same type and kind using factors of identical type and kind. The sequence index pairs associated with the factors provided by `factorList` are stored in a 2D Fortran array of default integer kind of the shape `integer::factorIndexList(2,:).` The sequence indices of the source Array elements are stored in the first row of `factorIndexList` while the sequence indices of the destination Array elements are stored in the second row.

Each PET in the current VM must call into `ESMF_ArraySMMStore()` to precompute and store the communication pattern for the sparse matrix multiplication. The multiplication factors may be provided in parallel, i.e. multiple PETs may specify `factorList` and `factorIndexList` arguments when calling into `ESMF_ArraySMMStore()`. PETs that do not provide factors either call with `factorList` and `factorIndexList` arguments containing zero elements or issue the call omitting both arguments.

```fortran
if (localPet == 0) then
    allocate(factorList(1)) ! PET 0 specifies 1 factor
    allocate(factorIndexList(2,1))
    factorList = (/0.2/) ! factors
    factorIndexList(1,:) = (/5/) ! seq indices into srcArray
    factorIndexList(2,:) = (/30/) ! seq indices into dstArray
    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
                              routehandle=sparseMatMulHandle, factorList=factorList, &
                              factorIndexList=factorIndexList, rc=rc)
    deallocate(factorList)
    deallocate(factorIndexList)
else if (localPet == 1) then
    allocate(factorList(3)) ! PET 1 specifies 3 factor
    allocate(factorIndexList(2,3))
    factorList = (/0.5, 0.5, 0.8/) ! factors
    factorIndexList(1,:) = (/8, 2, 12/) ! seq indices into srcArray
    factorIndexList(2,:) = (/11, 11, 30/) ! seq indices into dstArray
    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
                              routehandle=sparseMatMulHandle, factorList=factorList, &
                              factorIndexList=factorIndexList, rc=rc)
    deallocate(factorList)
    deallocate(factorIndexList)
else
    ! PETs 2 and 3 do not provide factors
    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
                             routehandle=sparseMatMulHandle, rc=rc)
endif
```

The RouteHandle object `sparseMatMulHandle` produced by `ESMF_ArraySMMStore()` can now be used to...
call ESMF_ArraySMM() collectively across all PETs of the current VM to perform

\[
\text{dstArray} = 0.0
\]
\[
\text{do } n=1, \text{ size(combinedFactorList)}
\]
\[
\text{dstArray(combinedFactorIndexList(2, n))} += \text{combinedFactorList(n)} \times \text{srcArray(combinedFactorIndexList(1, n))}
\]
\[
\text{enddo}
\]

in parallel. Here combinedFactorList and combinedFactorIndexList are the combined lists defined by the respective local lists provided by PETs 0 and 1 in parallel. For this example

\[
\text{call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, & routehandle=sparseMatMulHandle, rc=rc)}
\]

will initialize the entire dstArray to 0.0 and then update two elements:

on DE 1:
\[
dstArray(2) = 0.5 \times \text{srcArray}(0,0) + 0.5 \times \text{srcArray}(0,2)
\]

and

on DE 3:
\[
dstArray(1) = 0.2 \times \text{srcArray}(0,1) + 0.8 \times \text{srcArray}(1,3).
\]

The call to ESMF_ArraySMM() does provide the option to turn the default dstArray initialization off. If argument zeroflag is set to ESMF_REGION EMPTY

\[
\text{call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, & routehandle=sparseMatMulHandle, zeroflag=ESMF_REGION_EMPTY, rc=rc)}
\]

skips the initialization and elements in dstArray are updated according to:

\[
\text{do } n=1, \text{ size(combinedFactorList)}
\]
\[
\text{dstArray(combinedFactorIndexList(2, n))} += \text{combinedFactorList(n)} \times \text{srcArray(combinedFactorIndexList(1, n))}.
\]
\[
\text{enddo}
\]

The resources held by sparseMatMulHandle need to be deallocated by the user code before the handle becomes inaccessible.

\[
\text{call ESMF_ArraySMMRelease(routehandle=sparseMatMulHandle, rc=rc)}
\]

The Array sparse matrix multiplication also applies to Arrays with undistributed dimensions. The undistributed dimensions are interpreted in a sequentialized manner, much like the distributed dimensions, introducing a second sequence index for source and destination elements. Sequence index 1 is assigned to the first element in the first (i.e. fastest varying in memory) undistributed dimension. The following undistributed elements are labeled in consecutive order as they are stored in memory.

In the simplest case the Array sparse matrix multiplication will apply an identity matrix to the vector of sequentialized undistributed Array elements for every non-zero element in the sparse matrix. The requirement in this case is that the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, be the same for source and destination Array.
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
totalLWidth=/(1,1/), totalUWidth=/(1,1/), indexflag=ESMF_INDEX_GLOBAL, &
distgridToArrayMap=/(1,2/), undistLBound=/(1/), undistUBound=/(2/), rc=rc)

call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
distgridToArrayMap=/(2/), undistLBound=/(1/), undistUBound=/(2/), rc=rc)

Setting up factorList and factorIndexList is identical to the case for Arrays without undistributed dimensions. Also the call to ESMF_ArraySMMStore() remains unchanged. Internally, however, the source and destination Arrays are checked to make sure the total undistributed element count matches.

if (localPet == 0) then
allocate(factorList(1)) ! PET 0 specifies 1 factor
allocate(factorIndexList(2,1))
factorList = /(0.2/) ! factors
factorIndexList(1,:) = /(5/) ! seq indices into srcArray
factorIndexList(2,:) = /(30/) ! seq indices into dstArray
call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
routenumber=sparseMatMulHandle, factorList=factorList, &
factorIndexList=factorIndexList, rc=rc)
deallocate(factorList)
deallocate(factorIndexList)
else if (localPet == 1) then
allocate(factorList(3)) ! PET 1 specifies 3 factors
allocate(factorIndexList(2,3))
factorList = /(0.5, 0.5, 0.8/) ! factors
factorIndexList(1,:) = /(8, 2, 12/) ! seq indices into srcArray
factorIndexList(2,:) = /(11, 11, 30/) ! seq indices into dstArray
call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
routenumber=sparseMatMulHandle, factorList=factorList, &
factorIndexList=factorIndexList, rc=rc)
deallocate(factorList)
deallocate(factorIndexList)
else
! PETs 2 and 3 do not provide factors

call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
routenumber=sparseMatMulHandle, rc=rc)
endif

The call into the ESMF_ArraySMM() operation is completely transparent with respect to whether source and/or destination Arrays contain undistributed dimensions.

call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
routenumber=sparseMatMulHandle, rc=rc)

This operation will initialize the entire dstArray to 0.0 and then update four elements:
on DE 1:
\[
\begin{align*}
\text{dstArray}[1](2) &= 0.5 \times \text{srcArray}(0,0)[1] + 0.5 \times \text{srcArray}(0,2)[1], \\
\text{dstArray}[2](2) &= 0.5 \times \text{srcArray}(0,0)[2] + 0.5 \times \text{srcArray}(0,2)[2],
\end{align*}
\]

and

on DE 3:
\[
\begin{align*}
\text{dstArray}[1](1) &= 0.2 \times \text{srcArray}(0,1)[1] + 0.8 \times \text{srcArray}(1,3)[1], \\
\text{dstArray}[2](1) &= 0.2 \times \text{srcArray}(0,1)[2] + 0.8 \times \text{srcArray}(1,3)[2].
\end{align*}
\]

Here indices between "()" refer to distributed dimensions while indices between "[]" correspond to undistributed dimensions.

In a more general version of the Array sparse matrix multiplication the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, need not be the same for source and destination Array. In this formulation each non-zero element of the sparse matrix is identified with a unique element in the source and destination Array. This requires a generalization of the factorIndexList argument which now must contain four integer numbers for each element. These numbers in sequence are the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the source Array, followed by the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the destination Array.

In a more general version of the Array sparse matrix multiplication the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, need not be the same for source and destination Array. In this formulation each non-zero element of the sparse matrix is identified with a unique element in the source and destination Array. This requires a generalization of the factorIndexList argument which now must contain four integer numbers for each element. These numbers in sequence are the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the source Array, followed by the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the destination Array.

Setting up factorList is identical to the previous cases since there is still only one value associated with each non-zero matrix element. However, each entry in factorIndexList now has 4 instead of just 2 components.

```fortran
if (localPet == 0) then
  allocate(factorList(1)) ! PET 0 specifies 1 factor
  allocate(factorIndexList(4,1)) ! factors
  factorList = (/0.2/) ! factors
  factorIndexList(1,:) = (/5/) ! seq indices into srcArray
  factorIndexList(2,:) = (/1/) ! undistr. seq indices into srcArray
  factorIndexList(3,:) = (/30/) ! seq indices into dstArray
  factorIndexList(4,:) = (/2/) ! undistr. seq indices into dstArray
  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
                           routehandle=sparseMatMulHandle, factorList=factorList, &
                           factorIndexList=factorIndexList, rc=rc)
  deallocate(factorList)
  deallocate(factorIndexList)
else if (localPet == 1) then
  allocate(factorList(3)) ! PET 1 specifies 3 factor
  allocate(factorIndexList(4,3)) ! factors
  factorList = (/0.5, 0.5, 0.8/) ! factors
  factorIndexList(1,:) = (/8, 2, 12/) ! seq indices into srcArray
```

178
factorIndexList(2,:) = (/2, 1, 1/)  ! undistr. seq indices into srcArray
factorIndexList(3,:) = (/11, 11, 30/)  ! seq indices into dstArray
factorIndexList(4,:) = (/4, 4, 2/)  ! undistr. seq indices into dstArray

call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=sparseMatMulHandle, factorList=factorList, &
factorIndexList=factorIndexList, rc=rc)
dallocate(factorList)
dallocate(factorIndexList)
else
  ! PETs 2 and 3 do not provide factors
call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
routehandle=sparseMatMulHandle, rc=rc)
endif

The call into the ESMF_ArraySMM() operation remains unchanged.
call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
routehandle=sparseMatMulHandle, rc=rc)

This operation will initialize the entire dstArray to 0.0 and then update two elements:

```
on DE 1:
dstArray[4](2) = 0.5 * srcArray(0,0)[1] + 0.5 * srcArray(0,2)[2],
```

and

```
on DE 3:
dstArray[2](1) = 0.2 * srcArray(0,1)[1] + 0.8 * srcArray(1,3)[1],
```

Here indices in () refer to distributed dimensions while indices in [] correspond to undistributed dimensions.

## 20.3 Restrictions and Future Work

- The Fortran array pointer `farrayPtr` obtainable from `ESMF_ArrayGet()` is not suitable to be used in Fortran `deallocate()` calls. Deallocation of user allocated Fortran arrays must use the appropriate variable returned by the Fortran `allocate()` expression.
- Non-blocking communication, PET-based and DE-based, has not been formulated or implemented.

## 20.4 Design and Implementation Notes

The Array class is part of the ESMF index space layer and is built on top of the DistGrid and DELayout classes. The DELayout class introduces the notion of decomposition elements (DEs) and their layout across the available PETs. The DistGrid describes how index space is decomposed by assigning logically rectangular index space pieces or DE-local tiles to the DEs. The Array finally associates a local memory allocation with each local DE. The following is a list of implementation specific details about the current ESMF Array.

- Implementation language is C++.
- Local memory allocations are internally held in `ESMF_LocalArray` objects.
- All precomputed communication methods are based on sparse matrix multiplication.
20.5 Class API

20.5.1 ESMF_ArrayCreate - Create Array object from Fortran90 array

INTERFACE:

! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAssmdShape<rank><type><kind>(farray, &
distgrid, copyflag, distgridToArrayMap, computationalEdgeLWidth, &
computationalEdgeUWidth, computationalLWidth, &
totalLWidth, &
totalUWidth, indexflag, staggerLoc, vectorDim, &
undistLBound, undistUBound, name, rc)

ARGUMENTS:

<type> (ESMF_KIND_<kind>),dimension(<rank>),intent(in),target :: farray
type(ESMF_DistGrid), intent(in) :: distgrid
type(ESMF_CopyFlag), intent(in), optional :: copyflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: computationalEdgeLWidth(:)
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, intent(in), optional :: staggerLoc
integer, intent(in), optional :: vectorDim
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc

RETURN VALUE:

type(ESMF_Array) :: ESMF_ArrayCreateAssmdShapemrankDmtypekind

DESCRIPTION:

Create an ESMF_Array object from an existing local native Fortran90 array according to distgrid. Besides farray each PET must issue this call with identical arguments in order to create a consistent Array object. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for array dimensions that are not distributed can be chosen freely using the undistLBound and undistUBound arguments.

This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. By default all tensor elements are associated with stagger location 0. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.

The return value is the newly created ESMF_Array object.

The arguments are:
**farray** Valid native Fortran90 array, i.e. memory must be associated with the actual argument. The type/kind/rank information of farray will be used to set Array’s properties accordingly. The shape of farray will be checked against the information contained in the distgrid.

**distgrid** ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of farray.

**copyflag** Specifies whether the Array object will reference the memory allocation provided by farray directly or will copy the data from farray into a new memory allocation. Valid options are ESMF_DATA_REF (default) or ESMF_DATA_COPY. Depending on the specific situation the ESMF_DATA_REF option may be unsafe when specifying an array slice for farray.

**distgridToArrayMap** List that contains as many elements as is indicated by distgrid’s dimCount. The list elements map each dimension of the DistGrid object to a dimension in farray by specifying the appropriate Array dimension index. The default is to map all of distgrid’s dimensions against the lower dimensions of the farray argument in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped farray dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

**computationalEdgeLWidth** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a patch. The default is a zero vector.

**computationalEdgeUWidth** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a patch. The default is a zero vector.

**computationalLWidth** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

**computationalUWidth** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

**totalLWidth** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region exactly.

**totalUWidth** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the computational region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the farray argument.

**indexflag** Indicate how DE-local indices are defined. By default each DE’s exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section 9.1.7 for a list of valid indexflag options.

**staggerLoc** Stagger location is an arbitrary integer index. This information is used to correctly apply connection transformations of the corresponding DistGrid during halo operations.

**vectorDim** If the data stored in this Array object is a component of a vector field then the vectorDim argument may be used to identify the dimension along which the vector component is aligned. This information is used to correctly apply the signChangeVector defined in the connection transformations of the corresponding DistGrid during halo operations.
[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.
[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in farray.

[name] Name of the Array object.

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

20.5.2 ESMF_ArrayCreate - Create Array object from a list of LocalArray objects

INTERFACE:

! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateLocalArray(larrayList, distgrid, copyflag, &
distgridToArrayMap, computationalEdgeLWidth, computationalEdgeUWidth, &
computationalLWidth, computationalUWidth, totalLWidth, totalUWidth, &
indexflag, staggerLoc, vectorDim, undistLBound, undistUBound, name, rc)

ARGUMENTS:

type(ESMF_LocalArray), intent(in) :: larrayList(:)
type(ESMF_DistGrid), intent(in) :: distgrid
type(ESMF_CopyFlag), intent(in), optional :: copyflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: computationalEdgeLWidth(:)
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, intent(in), optional :: staggerLoc
integer, intent(in), optional :: vectorDim
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*)), intent(in), optional :: name
integer, intent(out), optional :: rc

RETURN VALUE:

type(ESMF_Array) :: ESMF_ArrayCreateLocalArray

DESCRIPTION:

Create an ESMF_Array object from existing ESMF_LocalArray objects according to distgrid. Besides larrayList each PET must issue this call with identical arguments in order to create a consistent Array object. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for array dimensions that are not distributed can be chosen freely using the undistLBound and undistUBound arguments.
This interface is able to handle multiple DEs per PET.
The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. By default all tensor elements are associated with stagger location 0. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.
The return value is the newly created ESMF_Array object.
The arguments are:
**larrayList** List of valid ESMF_LocalArray objects, i.e. memory must be associated with the actual arguments. The type/kind/rank information of all larrayList elements must be identical and will be used to set Array’s properties accordingly. The shape of each larrayList element will be checked against the information contained in the distgrid.

**distgrid** ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

**[copyflag]** Specifies whether the Array object will reference the memory allocation provided by farray directly or will copy the data from farray into a new memory allocation. Valid options are ESMF_DATA_REF (default) or ESMF_DATA_COPY. Depending on the specific situation the ESMF_DATA_REF option may be unsafe when specifying an array slice for farray.

**[distgridToArrayMap]** List that contains as many elements as is indicated by distgrid's dimCount. The list elements map each dimension of the DistGrid object to a dimension in the larrayList elements by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the larrayList elements in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped dimensions in the larrayList elements are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

**[computationalEdgeLWidth]** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a patch.

**[computationalEdgeUWidth]** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a patch.

**[computationalLWidth]** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

**[computationalUWidth]** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

**[totalLWidth]** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region exactly.

**[totalUWidth]** This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the larrayList argument.

**[indexflag]** Indicate how DE-local indices are defined. By default each DE’s exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section 9.1.7 for a list of valid indexflag options.

**[staggerLoc]** Stagger location is an arbitrary integer index. This information is used to correctly apply connection transformations of the corresponding DistGrid during halo operations.
[vectorDim] If the data stored in this Array object is a component of a vector field then the vectorDim argument may be used to identify the dimension along which the vector component is aligned. This information is used to correctly apply the signChangeVector defined in the connection transformations of the corresponding DistGrid during halo operations.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in larrayList.

[name] Name of the Array object.

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

### 20.5.3 ESMF_ArrayCreate - Create Array object from specification and allocate memory

**INTERFACE:**

![Private name; call using ESMF_ArrayCreate()](function ESMF_ArrayCreateAllocate(arrayspec, distgrid, distgridToArrayMap, & computationalEdgeLWidth, computationalEdgeUWidth, & computationalLWidth, computationalUWidth, totalLWidth, totalUWidth, & indexflag, staggerLoc, vectorDim, undistLBound, undistUBound, name, rc)

**ARGUMENTS:**

- type(ESMF_ArraySpec), intent(inout) :: arrayspec
- type(ESMF_DistGrid), intent(in) :: distgrid
- integer, intent(in), optional :: distgridToArrayMap(:)
- integer, intent(in), optional :: computationalEdgeLWidth(:)
- integer, intent(in), optional :: computationalEdgeUWidth(:)
- integer, intent(in), optional :: computationalLWidth(:)
- integer, intent(in), optional :: computationalUWidth(:)
- integer, intent(in), optional :: totalLWidth(:)
- integer, intent(in), optional :: totalUWidth(:)
- type(ESMF_IndexFlag), intent(in), optional :: indexflag
- integer, intent(in), optional :: staggerLoc
- integer, intent(in), optional :: vectorDim
- integer, intent(in), optional :: undistLBound(:)
- integer, intent(in), optional :: undistUBound(:)
- character (len=*), intent(in), optional :: name
- integer, intent(out), optional :: rc

**RETURN VALUE:**

- type(ESMF_Array) :: ESMF_ArrayCreateAllocate

**DESCRIPTION:**

Create an ESMF_Array object and allocate uninitialized data space according to arrayspec and distgrid. Each PET must issue this call with identical arguments in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the arguments to this call: distgrid and the optional Width arguments. The return value is the newly created ESMF_Array object.

The arguments are:

arrayspec ESMF_ArraySpec object containing the type/kind/rank information.
**distgrid**  
ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

[distgridToArrayMap]  
List that contains as many elements as is indicated by distgrid's dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the Array object in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth]  
This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a patch.

[computationalEdgeUWidth]  
This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a patch.

[computationalLWidth]  
This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth]  
This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalMemoryLWidth]  
This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region.

[totalMemoryUWidth]  
This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the computational region. The default is to accommodate the union of exclusive and computational region.

[indexflag]  
Indicate how DE-local indices are defined. By default each DE’s exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section 9.1.7 for a list of valid indexflag options.

[staggerLoc]  
Stagger location is an arbitrary integer index. This information is used to correctly apply connection transformations of the corresponding DistGrid during halo operations.

[vectorDim]  
If the data stored in this Array object is a component of a vector field then the vectorDim argument may be used to identify the dimension along which the vector component is aligned. This information is used to correctly apply the signChangeVector defined in the connection transformations of the corresponding DistGrid during halo operations.

[undistLBound]  
Lower bounds for the array dimensions that are not distributed.

[undistUBound]  
Upper bounds for the array dimensions that are not distributed.

[name]  
Name of the Array object.

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

185
20.5.4 ESMF_ArrayCreate - Create Array object as copy of existing Array object

INTERFACE:

! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateCopy(array, rc)

ARGUMENTS:

type(ESMF_Array), intent(in) :: array
integer, intent(out), optional :: rc

RETURN VALUE:

type(ESMF_Array) :: ESMF_ArrayCreateCopy

DESCRIPTION:

Create an ESMF_Array object as the copy of an existing Array. The return value is the newly created ESMF_Array object. The arguments are:

array ESMF_Array object to be copied.

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

---

20.5.5 ESMF_ArrayDestroy - Destroy Array object

INTERFACE:

subroutine ESMF_ArrayDestroy(array, rc)

ARGUMENTS:

type(ESMF_Array), intent(inout) :: array
integer, intent(out), optional :: rc

DESCRIPTION:

Destroy an ESMF_Array object. The arguments are:

array ESMF_Array object to be destroyed.

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

---

20.5.6 ESMF_ArrayGather - Gather a Fortran90 array from an ESMF_Array

INTERFACE:

subroutine ESMF_ArrayGather<rank><type><kind>(array, farray, patch, & rootPet, vm, rc)

ARGUMENTS:
DESCRIPTION:

Gather the data of an ESMF_Array object into the farray located on rootPET. A single DistGrid patch of array must be gathered into farray. The optional patch argument allows selection of the patch. For Arrays defined on a single patch DistGrid the default selection (patch 1) will be correct. The shape of farray must match the shape of the patch in Array.

If the Array contains replicating DistGrid dimensions data will be gathered from the numerically higher DEs. Replicated data elements in numerically lower DEs will be ignored.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for all of its DEs. The call will block until all PET-local data objects are accessible.

The arguments are:

array The ESMF_Array object from which data will be gathered.

[farray] The Fortran90 array into which to gather data. Only root must provide a valid farray.

[patch] The DistGrid patch in array from which to gather farray. By default farray will be gathered from patch 1.

rootPet PET that holds the valid destination array, i.e. farray.

[vm] Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method’s overhead.

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

20.5.7 ESMF_ArrayGet - Get Array internals

INTERFACE:

! Private name; call using ESMF_ArrayGetDefault()
subroutine ESMF_ArrayGetDefault(array, typekind, rank, larrayList, &
indexflag, distgridToArrayMap, distgridToPackedArrayMap, &
arrayToDistGridMap, undistLBound, undistUBound, exclusiveLBound, &
exclusiveUBound, computationalLBound, computationalUBound, totalLBound, &
totalUBound, computationalLWidth, computationalUWidth, totalLWidth, &
totalUWidth, name, distgrid, dimCount, patchCount, minIndexPDimPPatch, &
maxIndexPDimPPatch, patchListPDe, indexCountPDimPDe, delayout, deCount, &
localDeCount, localDeList, rc)

ARGUMENTS:

  type(ESMF_Array), intent(in) :: array
  type(ESMF_TypeKind), intent(out), optional :: typekind
  integer, intent(out), optional :: rank
  type(ESMF_LocalArray), target, intent(out), optional :: larrayList(:)
  type(ESMF_IndexFlag), intent(out), optional :: indexflag

187
DESCRIPTION:

Get internal information.
This interface works for any number of DEs per PET.
The arguments are:

array  Queried ESMF_Array object.
[typekind]  TypeKind of the Array object.
[rank]  Rank of the Array object.
[larrayList]  Upon return this holds a list of the associated ESMC_LocalArray objects. larrayList must be allocated to be of size localDeCount, i.e. the number of DEs associated with the calling PET.
[indexflag]  Upon return this flag indicates how the DE-local indices are defined. See section 9.1.7 for a list of possible return values.
[distgridToArrayMap]  Upon return this list holds the Array dimensions against which the DistGrid dimensions are mapped. distgridToArrayMap must be allocated to be of size dimCount. An entry of zero indicates that the respective DistGrid dimension is replicating the Array across the DEs along this direction.
[distgridToPackedArrayMap]  Upon return this list holds the indices of the Array dimensions in packed format against which the DistGrid dimensions are mapped. distgridToPackedArrayMap must be allocated to be of size dimCount. An entry of zero indicates that the respective DistGrid dimension is replicating the Array across the DEs along this direction.
[arrayToDistGridMap]  Upon return this list holds the DistGrid dimensions against which the Array dimensions are mapped. arrayToDistGridMap must be allocated to be of size rank. An entry of zero indicates that the respective Array dimension is not decomposed, rendering it a tensor dimension.
Upon return this array holds the lower bounds of the undistributed dimensions of the Array. `UndistLBound` must be allocated to be of size `rank-dimCount`.

Upon return this array holds the upper bounds of the undistributed dimensions of the Array. `UndistUBound` must be allocated to be of size `rank-dimCount`.

Upon return this holds the lower bounds of the exclusive regions for all PET-local DEs. `exclusiveLBound` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the upper bounds of the exclusive regions for all PET-local DEs. `exclusiveUBound` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the lower bounds of the computational regions for all PET-local DEs. `computationalLBound` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the upper bounds of the computational regions for all PET-local DEs. `computationalUBound` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the lower bounds of the total regions for all PET-local DEs. `totalLBound` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the upper bounds of the total regions for all PET-local DEs. `totalUBound` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the lower width of the computational regions for all PET-local DEs. `computationalLWidth` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the upper width of the computational regions for all PET-local DEs. `computationalUWidth` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the lower width of the total memory regions for all PET-local DEs. `totalLWidth` must be allocated to be of size `(dimCount, localDeCount)`.

Upon return this holds the upper width of the total memory regions for all PET-local DEs. `totalUWidth` must be allocated to be of size `(dimCount, localDeCount)`.

Name of the Array object.

Upon return this holds the associated ESMF_DistGrid object.

Number of dimensions (rank) of `distgrid`.

Number of patches in `distgrid`.

Lower index space corner per dim, per patch, with size `minIndexPDimPPatch`.

Upper index space corner per dim, per patch, with size `maxIndexPDimPPatch`.

List of patch id numbers, one for each DE, with size `patchListPDe`.

Array of extents per dim, per de, with size `indexCountPDimPDe`.

Upon return this holds the associated ESMF_DELayout object.

Upon return this holds the total number of DEs defined in the DELayout associated with the Array object.

Upon return this holds the number of PET-local DEs defined in the DELayout associated with the Array object.
Upon return this holds the list of DE ids for the PET-local DEs defined in the DELayout associated with the Array object. The provided argument must be of size localDeCount.

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

20.5.8 ESMF_ArrayGet - Get Array internals per dim per local DE

INTERFACE:

! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetPLocalDePDim(array, localDe, dim, indexCount, indexList, rc)

ARGUMENTS:

type(ESMF_Array), intent(in) :: array
integer, intent(in) :: localDe
integer, intent(in) :: dim
integer, intent(out), optional :: indexCount
integer, intent(out), optional :: indexList(:)
integer, intent(out), optional :: rc

DESCRIPTION:

Get internal information per local DE, per dim.
This interface works for any number of DEs per PET.
The arguments are:

array Queried ESMF_Array object.
localDe Local DE for which information is requested. [0,..,localDeCount-1]
dim Dimension for which information is requested. [1,..,dimCount]
[indexCount] DistGrid indexCount associated with localDe, dim.
[indexList] List of DistGrid patch-local indices for localDe along dimension dim.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.5.9 ESMF_ArrayGet - Get access to PET-local Array patch via Fortran90 array pointer

INTERFACE:

! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetFPtr<rank><type><kind>(array, localDe, farrayPtr, rc)

ARGUMENTS:

type(ESMF_Array), intent(in) :: array
integer, intent(in) :: localDe
<type> (ESMF_KIND_<kind>),dimension(<rank>),pointer :: farrayPtr
integer, intent(out), optional :: rc
DESCRIPTION:
Get Fortran90 pointer to the DE-local memory regions in Array object for the specified local DE. The arguments are:

array  Queried ESMF_Array object.
localDe Local DE for which information is requested. [0,..,localDeCount-1]
farrayPtr Upon return farrayPtr points to the DE-local data allocation of array.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.5.10  ESMF_ArrayGet - Get access to PET-local Array patch via LocalArray object.

INTERFACE:
! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetLarray(array, localDe, larray, rc)

ARGUMENTS:
  type(ESMF_Array), intent(in) :: array
  integer, intent(in) :: localDe
  type(ESMF_LocalArray), intent(inout) :: larray
  integer, intent(out), optional :: rc

DESCRIPTION:
Provide access to ESMF_LocalArray object that holds data for the specified local DE. The arguments are:

array  Queried ESMF_Array object.
localDe Local DE for which information is requested. [0,..,localDeCount-1]
larray Upon return larray refers to the DE-local data allocation of array.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

20.5.11  ESMF_ArrayPrint - Print Array internals

INTERFACE:
subroutine ESMF_ArrayPrint(array, options, rc)

ARGUMENTS:
  type(ESMF_Array), intent(in) :: array
  character(len=*) , intent(in), optional :: options
  integer, intent(out), optional :: rc
DESCRIPTION:

Print internal information of the specified ESMF_Array object.
Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.
The arguments are:

array ESMF_Array object.

[options] Print options are not yet supported.

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

20.5.12 ESMF_ArrayRedist - Execute an Array redistribution

INTERFACE:

subroutine ESMF_ArrayRedist(srcArray, dstArray, routehandle, checkflag, rc)

ARGUMENTS:

type(ESMF_Array), intent(in), optional :: srcArray

type(ESMF_Array), intent(inout),optional :: dstArray

type(ESMF_RouteHandle), intent(inout) :: routehandle

type(ESMF_Logical), intent(in), optional :: checkflag

integer, intent(out), optional :: rc

DESCRIPTION:

Execute a precomputed Array redistribution from srcArray to dstArray. Both srcArray and dstArray must be congruent and typekind conform with the respective Arrays used during ESMF_ArrayRedistStore(). Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.
It is erroneous to specify the identical Array object for srcArray and dstArray arguments.
See ESMF_ArrayRedistStore() on how to precompute routehandle.
This call is collective across the current VM.

[srcArray] ESMF_Array with source data.

dstArray] ESMF_Array with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to ESMF_TRUE the input Array pair will be checked for consistency with the precomputed operation provided by routehandle. If set to ESMF_FALSE (default) only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to ESMF_FALSE to achieve highest performance.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
20.5.13  ESMF_ArrayRedistRelease - Release resources associated with Array redistribution

INTERFACE:

    subroutine ESMF_ArrayRedistRelease(routehandle, rc)

ARGUMENTS:

    type(ESMF_RouteHandle), intent(inout) :: routehandle
    integer, intent(out), optional :: rc

DESCRIPTION:

Release resources associated with an Array redistribution. After this call routehandle becomes invalid.

routehandle  Handle to the precomputed Route.

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

20.5.14  ESMF_ArrayRedistStore - Precompute Array redistribution with local factor argument

INTERFACE:

    ! Private name; call using ESMF_ArrayRedistStore()
    subroutine ESMF_ArrayRedistStore&type><kind>(srcArray, dstArray, routehandle, &
    factor, srcToDstTransposeMap, rc)

ARGUMENTS:

    type(ESMF_Array), intent(in) :: srcArray
    type(ESMF_Array), intent(inout) :: dstArray
    type(ESMF_RouteHandle), intent(inout) :: routehandle
    <type>(ESMF_KIND_<kind>), intent(in) :: factor
    integer, intent(in), optional :: srcToDstTransposeMap(:)
    integer, intent(out), optional :: rc

DESCRIPTION:

Store an Array redistribution operation from srcArray to dstArray. PETs that specify a factor argument must use the <type><kind> overloaded interface. Other PETs call into the interface without factor argument. If multiple PETs specify the factor argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a factor argument the default will be a factor of 1.
Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of patches within the DistGrid or by user-supplied arbitrary sequence indices. See section 20.2.15 for details on the definition of sequence indices. Redistribution corresponds to an identity mapping of the source Array vector to the destination Array vector.
Source and destination Arrays may be of different <type><kind>. Further source and destination Arrays may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical Array object for srcArray and dstArray arguments.
The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayRedist() on any pair of Arrays that are congruent and typekind conform with the srcArray, dstArray pair. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.

This call is collective across the current VM.

srcArray  ESMF_Array with source data.
dstArray  ESMF_Array with destination data.
routehandle Handle to the precomputed Route.

[factor]  Factor by which to multiply source data. Default is 1.

[srcToDstTransposeMap]  List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

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

20.5.15  ESMF_ArrayRedistStore - Precompute Array redistribution without local factor argument

INTERFACE:

INTERFACE:

! Private name; call using ESMF_ArrayRedistStore()
subroutine ESMF_ArrayRedistStoreNF(srcArray, dstArray, routehandle, &
srcToDstTransposeMap, rc)

ARGUMENTS:

type(ESMF_Array), intent(in) :: srcArray

type(ESMF_Array), intent(inout) :: dstArray

type(ESMF_RouteHandle), intent(inout) :: routehandle

integer, intent(in), optional :: srcToDstTransposeMap(:)

integer, intent(out), optional :: rc

DESCRIPTION:

Store an Array redistribution operation from srcArray to dstArray. PETs that specify a factor argument must use the <type><kind> overloaded interface. Other PETs call into the interface without factor argument. If multiple PETs specify the factor argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a factor argument the default will be a factor of 1.

Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of patches within the DistGrid or by user-supplied arbitrary sequence indices. See section 20.2.15 for details on the definition of sequence indices. Redistribution corresponds to an identity mapping of the source Array vector to the destination Array vector. Source and destination Arrays may be of different <type><kind>. Further source and destination Arrays may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical Array object for srcArray and dstArray arguments. The routine returns an ESMF_RouteHandle object that can be used to call ESMF_ArrayRedist() on any pair of Arrays that are congruent and typekind conform with the srcArray, dstArray pair. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.

This call is collective across the current VM.

srcArray  ESMF_Array with source data.

dstArray  ESMF_Array with destination data.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap]  List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
20.5.16  ESMF_ArrayScatter - Scatter a Fortran90 array across the ESMF_Array

INTERFACE:

    subroutine ESMF_ArrayScatter<rank><type><kind>(array, farray, patch, &
        rootPet, vm, rc)

ARGUMENTS:

type(ESMF_Array), intent(inout) :: array
mtype (ESMF_KIND_mtypekind),dimension(mdim),intent(in),target :: farray
integer, intent(in), optional :: patch
integer, intent(in) :: rootPet
type(ESMF_VM), intent(in), optional :: vm
integer, intent(out), optional :: rc

DESCRIPTION:

Scatter the data of farray located on rootPET across an ESMF_Array object. A single farray must be scattered across a single DistGrid patch in Array. The optional patch argument allows selection of the patch. For Arrays defined on a single patch DistGrid the default selection (patch 1) will be correct. The shape of farray must match the shape of the patch in Array.

If the Array contains replicating DistGrid dimensions data will be scattered across all of the replicated pieces.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for all of its DEs. The call will block until all PET-local data objects are accessible.

The arguments are:

array  The ESMF_Array object across which data will be scattered.
[farray]  The Fortran90 array that is to be scattered. Only root must provide a valid farray.
[patch]  The DistGrid patch in array into which to scatter farray. By default farray will be scattered into patch 1.
rootPet  PET that holds the valid data in farray.
[vm]  Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method’s overhead.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

20.5.17  ESMF_ArraySet - Set Array properties

INTERFACE:

    subroutine ESMF_ArraySet(array, name, staggerLoc, vectorDim, &
        computationalLWidth, computationalUWidth, rc)

ARGUMENTS:

array  The ESMF_Array object across which data will be scattered.
[name]  The name of the property to set.
[staggerLoc]  The stagger location of the property.
[vectorDim]  The vector dimension of the property.
[computationalLWidth]  The computational lower width of the property.
[computationalUWidth]  The computational upper width of the property.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
type (ESMF_Array), intent (inout) :: array
character (len = *), intent (in), optional :: name
integer, intent (in), optional :: staggerLoc
integer, intent (in), optional :: vectorDim
integer, intent (in), optional :: computationalLWidth(:, :)
integer, intent (in), optional :: computationalUWidth(:, :)
integer, intent (out), optional :: rc

DESCRIPTION:

Sets adjustable settings in an ESMF_Array object. Arrays with tensor dimensions will set values for all tensor components.
The arguments are:

array  ESMF_Array object for which to set properties.

[name] The Array name.

[staggerLoc] User-defined stagger location.

[vectorDim] User-defined vector dimension.

[computationalLWidth] This argument must have of size (dimCount, localDeCount).
computationalLWidth specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for all local DEs.

[computationalUWidth] This argument must have of size (dimCount, localDeCount).
computationalUWidth specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for all local DEs.

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

20.5.18 ESMF_ArraySet - Set Array internals for specific tensor component

INTERFACE:

! Private name; call using ESMF_ArraySet()
subroutine ESMF_ArraySetTensor(array, tensorIndex, staggerLoc, vectorDim, rc)

ARGUMENTS:

type (ESMF_Array), intent (in) :: array
integer, intent (in) :: tensorIndex(:)
integer, intent (in), optional :: staggerLoc
integer, intent (in), optional :: vectorDim
integer, intent (out), optional :: rc

DESCRIPTION:

Sets adjustable settings in an ESMF_Array object for a specific tensor component.
The arguments are:

array  ESMF_Array object for which to set properties.

tensorIndex Specifies the tensor component within the not distributed array dimensions for which properties are to be set.
Stagger location of this tensor element.

Dimension along this vector component of this tensor element is aligned.

Return code; equals ESMF_SUCCESS if there are no errors.

---

**20.5.19 ESMF_ArraySMM - Execute an Array sparse matrix multiplication**

**INTERFACE:**

```
subroutine ESMF_ArraySMM(srcArray, dstArray, routehandle, zeroflag, &
checkflag, rc)
```

**ARGUMENTS:**

- `type(ESMF_Array), intent(in), optional :: srcArray`
- `type(ESMF_Array), intent(inout), optional :: dstArray`
- `type(ESMF_RouteHandle), intent(inout) :: routehandle`
- `type(ESMF_RegionFlag), intent(in), optional :: zeroflag`
- `type(ESMF_Logical), intent(in), optional :: checkflag`
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Execute a precomputed Array sparse matrix multiplication from `srcArray` to `dstArray`. Both `srcArray` and `dstArray` must be congruent and typekind conform with the respective Arrays used during `ESMF_ArraySMMStore()`. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

See `ESMF_ArraySMMStore()` on how to precompute `routehandle`. See section 20.2.15 for details on the operation `ESMF_ArraySMM()` performs.

This call is **collective** across the current VM.

- **[srcArray]** ESMF_Array with source data.
- **[dstArray]** ESMF_Array with destination data.
- **routehandle** Handle to the precomputed Route.
- **[zeroflag]** If set to `ESMF_REGION_TOTAL (default)` the total regions of all DEs in `dstArray` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstArray` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroflag` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section 9.1.11 for a complete list of valid settings.
- **[checkflag]** If set to `ESMF_TRUE` the input Array pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `ESMF_FALSE (default)` only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `ESMF_FALSE` to achieve highest performance.
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.
### 20.5.20 ESMF_ArraySMMRelease - Release resources associated with Array sparse matrix multiplication

**INTERFACE:**

```fortran
subroutine ESMF_ArraySMMRelease(routehandle, rc)
```

**ARGUMENTS:**

- `type(ESMF_RouteHandle), intent(inout) :: routehandle`
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Release resources associated with an Array sparse matrix multiplication. After this call `routehandle` becomes invalid.

- `routehandle` Handle to the precomputed Route.
- `[rc]` Return code; equals `ESMF_SUCCESS` if there are no errors.

---

### 20.5.21 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication with local factors

**INTERFACE:**

```fortran
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStore<type><kind>(srcArray, dstArray, &
    routehandle, factorList, factorIndexList, rc)
```

**ARGUMENTS:**

- `type(ESMF_Array), intent(in) :: srcArray`
- `type(ESMF_Array), intent(inout) :: dstArray`
- `type(ESMF_RouteHandle), intent(inout) :: routehandle`
- `<type>(ESMF_KIND_<kind>), target, intent(in) :: factorList(:)`
- `integer, intent(in) :: factorIndexList(:,:)`
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the `<type><kind>` overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface without `factorList` and `factorIndexList` arguments.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of patches within the DistGrid or by user-supplied arbitrary sequence indices. See section [20.2.15](#) for details on the definition of sequence indices.

Source and destination Arrays, as well as the supplied `factorList` argument, may be of different `<type><kind>`. Further source and destination Arrays may differ in shape and number of elements.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArraySMM()` on any pair of Arrays that are congruent and typekind conform with the `srcArray, dstArray` pair. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE.
This method is overloaded for:
  ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
  ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is collective across the current VM.

**srcArray**  ESMF_Array with source data.

**dstArray**  ESMF_Array with destination data.

**routehandle**  Handle to the precomputed Route.

**factorList**  List of non-zero coefficients.

**factorIndexList**  Pairs of sequence indices for the factors stored in factorList.

  The second dimension of factorIndexList steps through the list of pairs, i.e. size(factorIndexList,2) == size(factorList). The first dimension of factorIndexList is either of size 2 or size 4.

  In the size 2 format factorIndexList(1,:) specifies the sequence index of the source element in the srcArray while factorIndexList(2,:) specifies the sequence index of the destination element in dstArray. For this format to be a valid option source and destination Arrays must have matching number of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

  The size 4 format is more general and does not require a matching tensor element count. Here the factorIndexList(1,:) specifies the sequence index while factorIndexList(2,:) specifies the tensor sequence index of the source element in the srcArray. Further factorIndexList(3,:) specifies the sequence index and factorIndexList(4,:) specifies the tensor sequence index of the destination element in the dstArray.

  See section 20.2.15 for details on the definition of Array sequence indices and tensor sequence indices.

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

---

20.5.22  ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication without local factors

**INTERFACE:**

```fortran
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStoreNF(srcArray, dstArray, routehandle, rc)
```

**ARGUMENTS:**

```fortran
  type (ESMF_Array), intent(in) :: srcArray
  type (ESMF_Array), intent(inout) :: dstArray
  type (ESMF_RouteHandle), intent(inout) :: routehandle
  integer, intent(out), optional :: rc
```

**DESCRIPTION:**

Store an Array sparse matrix multiplication operation from srcArray to dstArray. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the factorList and factorIndexList arguments. Providing factorList and factorIndexList arguments with size(factorList) = (/0/) and size(factorIndexList) = (/2,0/) or (/4,0/) indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface without factorList and factorIndexList arguments.

Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of patches within the DistGrid or by user-supplied arbitrary sequence indices. See section 20.2.15 for details on the definition of sequence indices.
Source and destination Arrays, as well as the supplied factorList argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements. It is erroneous to specify the identical Array object for srcArray and dstArray arguments. The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArraySMM() on any pair of Arrays that are congruent and typekind conform with the srcArray, dstArray pair. Congruent Arrays possess matching DistGrids and the shape of the local array tiles matches between the Arrays for every DE. This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.
This call is collective across the current VM.
srcArray  ESMF_Array with source data.
dstArray  ESMF_Array with destination data.
routenhandle  Handle to the precomputed Route.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

20.5.23 ESMF_ArrayValidate - Validate Array internals

INTERFACE:

subroutine ESMF_ArrayValidate(array, rc)

ARGUMENTS:

    type(ESMF_Array), intent(in) :: array
    integer, intent(out), optional :: rc

DESCRIPTION:

Validates that the Array is internally consistent. The method returns an error code if problems are found. The arguments are:
array  Specified ESMF_Array object.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

21 LocalArray Class

21.1 Description

The ESMF_LocalArray class provides a language independent representation of data in array format. One of the major functions of the LocalArray class is to bridge the Fortran/C/C++ language difference that exists with respect to array representation. All ESMF Field and Array data is internally stored in ESMF LocalArray objects allowing transparent access from Fortran and C/C++.
In the ESMF Fortran API the LocalArray becomes visible in those cases where a local PET may be associated with multiple pieces of an Array, e.g. if there are multiple DEs associated with a single PET. The Fortran language standard does not provide an array of arrays construct, however arrays of derived types holding arrays are possible. ESMF calls use arguments that are of type ESMF_LocalArray with dimension attribute where necessary.
21.2 Restrictions and Future Work

- The TKR (type/kind/rank) overloaded LocalArray interfaces declare the dummy Fortran array arguments with the pointer attribute. The advantage of doing this is that it allows ESMF to inquire information about the provided Fortran array. The disadvantage of this choice is that actual Fortran arrays passed into these interfaces must also be defined with pointer attribute in the user code.

21.3 Class API

21.3.1 ESMF_LocalArrayCreate – Create a LocalArray explicitly specifying TKR arguments.

INTERFACE:

! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateByTKR(rank, typekind, counts, lbounds, &
ubounds, rc)

RETURN VALUE:

type(ESMF_LocalArray) :: ESMF_LocalArrayCreateByTKR

ARGUMENTS:

integer, intent(in) :: rank

type(ESMF_TypeKind), intent(in) :: typekind

integer, intent(in), optional :: counts(:)

integer, intent(in), optional :: lbounds(:)

integer, intent(in), optional :: ubounds(:)

integer, intent(out), optional :: rc

DESCRIPTION:

Create a new ESMF_LocalArray and allocate data space, which remains uninitialized. The return value is a new LocalArray.

The arguments are:

rank Array rank (dimensionality, 1D, 2D, etc). Maximum allowed is 7D.

typekind Array typekind. See section 9.2.1 for valid values.

[counts] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank.

The count argument may be omitted if both lbounds and ubounds arguments are present.

[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.

21.3.2 ESMF_LocalArrayCreate – Create a LocalArray specifying an ArraySpec

INTERFACE:

! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateBySpec(arrayspec, counts, lbounds, ubounds, rc)

RETURN VALUE:
type (ESMF_LocalArray) :: ESMF_LocalArrayCreateBySpec

ARGUMENTS:

  type (ESMF_ArraySpec), intent (inout) :: arrayspec
  integer, intent (in), optional :: counts(:)
  integer, intent (in), optional :: lbounds(:)
  integer, intent (in), optional :: ubounds(:)
  integer, intent (out), optional :: rc

DESCRIPTION:

Create a new ESMF_LocalArray and allocate data space, which remains uninitialized. The return value is a new LocalArray.

The arguments are:

arrayspec ArraySpec object specifying typekind and rank.
[counts] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The count argument may be omitted if both lbounds and ubounds arguments are present.
[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.

21.3.3 ESMF_LocalArrayCreate – Create a LocalArray from existing one

INTERFACE:

! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateCopy(larray, rc)

RETURN VALUE:

  type (ESMF_LocalArray) :: ESMF_LocalArrayCreateCopy

ARGUMENTS:

  type (ESMF_LocalArray), intent (in) :: larray
  integer, intent (out), optional :: rc

DESCRIPTION:

Perform a deep copy of an existing ESMF_LocalArray object. The return value is a new LocalArray.

The arguments are:

larray Existing LocalArray to be copied.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.
21.3.4 ESMF_LocalArrayCreate - Create a LocalArray from a Fortran pointer (associated or unassociated)

INTERFACE:

! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrCreateByPtr<rank><type><kind>(fptr, docopy, counts, &
  lbounds, ubounds, rc)

RETURN VALUE:

type(ESMF_LocalArray) :: ESMF_LocalArrCreateByPtr<rank><type><kind>

ARGUMENTS:

<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: fptr
  type(ESMF_CopyFlag), intent(in), optional :: docopy
  integer, intent(in), optional :: counts(:)
  integer, intent(in), optional :: lbounds(:)
  integer, intent(in), optional :: ubounds(:)
  integer, intent(out), optional :: rc

DESCRIPTION:

Creates an ESMF_LocalArray based on a Fortran array pointer. Two cases must be distinguished.
First, if fptr is associated the optional docopy argument may be used to indicate whether the associated data is to
be copied or referenced. For associated fptr the optional counts, lbounds and ubounds arguments need not
be specified. However, all present arguments will be checked against fptr for consistency.
Second, if fptr is unassociated the optional argument docopy must not be specified. However, in this case a
complete set of counts and bounds information must be provided. Any combination of present counts lbounds
and ubounds arguments that provides a complete specification is valid. All input information will be checked for
consistency.
The arguments are:

fptr  A Fortran array pointer (associated or unassociated).

[docopy] Indicate copy vs. reference behavior in case of associated fptr. This argument must not be present for
unassociated fptr. Default to ESMF_DATA_REF, makes the ESMF_LocalArray reference the associated
data array. If set to ESMF_DATA_COPY this routine allocates new memory and copies the data from the pointer
into the new LocalArray allocation.

[counts] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank.
The count argument may be omitted if both lbounds and ubounds arguments are present.

[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.

21.3.5 ESMF_LocalArrayDestroy - Destroy a LocalArray object

INTERFACE:
subroutine ESMF_LocalArrayDestroy(larray, rc)

ARGUMENTS:

    type(ESMF_LocalArray), intent(inout) :: larray
    integer, intent(out), optional :: rc

DESCRIPTION:

Releases all resources associated with this ESMF_LocalArray object.
The arguments are:

larray  Destroy contents of this ESMF_LocalArray.

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

---

21.3.6  ESMF_LocalArrayGet - Return LocalArray information.

INTERFACE:

    ! Private name; call using ESMF_LocalArrayGet()
    subroutine ESMF_LocalArrayGetDefault(larray, rank, typekind, counts, lbounds, &
                                         ubounds, base, name, rc)

ARGUMENTS:

    type(ESMF_LocalArray), intent(in) :: larray
    integer, intent(out), optional :: rank
    type(ESMF_TypeKind), intent(out), optional :: typekind
    integer, intent(out), optional :: counts(:)
    integer, intent(out), optional :: lbounds(:)
    integer, intent(out), optional :: ubounds(:)
    type(ESMF_Pointer), intent(out), optional :: base
    character(len=ESMF_MAXSTR), intent(out), optional :: name
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns information about the ESMF_LocalArray.
The arguments are:

larray  Queried ESMF_LocalArray object.

[rank]  Rank of the LocalArray object.

[typekind]  TypeKind of the LocalArray object.

[counts]  Count per dimension.

[lbounds]  Lower bound per dimension.

[ubounds]  Upper bound per dimension.

[base]  Base class object.

[name]  Name of the LocalArray object.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
21.3.7 ESMF_LocalArrayGet - Get access to data in LocalArray object

INTERFACE:

! Private name; call using ESMF_LocalArrayGet()
subroutine ESMF_LocalArrayGetData<rank><type><kind>(larray, fptr, docopy, rc)

ARGUMENTS:

  type(ESMF_LocalArray) :: larray
  <type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: fptr
  type(ESMF_CopyFlag), intent(in), optional :: docopy
  integer, intent(out), optional :: rc

DESCRIPTION:

Return a Fortran pointer to the data buffer, or return a Fortran pointer to a new copy of the data.
The arguments are:

larray  The ESMF_LocalArray to get the value from.
fptr  An unassociated or associated Fortran pointer correctly allocated.
[docopy]  An optional copy flag which can be specified. Can either make a new copy of the data or reference existing
  data. See section 9.1.4 for a list of possible values.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

22 ArraySpec Class

22.1 Description

An ArraySpec is a very simple class that contains type, kind, and rank information about an array. This information is
stored in two parameters. TypeKind describes the data type of the elements in the array and their precision. Rank is
the number of dimensions in the array.
The only methods that are associated with the ArraySpec class are those that allow you to set and retrieve this inform-

22.2 Use and Examples

The ArraySpec is passed in as an argument at Field and FieldBundle 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
custom 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_TypeKind) :: mytypekind

! return code
integer:: rc

! initialize ESMF framework
call ESMF_Initialize(rc=rc)

22.2.1 Setting ArraySpec Values
This example shows how to set values in an ESMF_ArraySpec.

call ESMF_ArraySpecSet(arrayDS, rank=2, &
typekind=ESMF_TYPEKIND_R8, rc=rc)

22.2.2 Getting ArraySpec Values
This example shows how to query an ESMF_ArraySpec.

call ESMF_ArraySpecGet(arrayDS, myrank, mytypekind, rc)
print *, "Returned values from ArraySpec:"
print *, "rank =", myrank

! finalize ESMF framework
call ESMF_Finalize(rc=rc)

end program ESMF_ArraySpecEx

22.3 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.

22.4 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.
22.5 Class API

22.5.1 ESMF_ArraySpecGet - Get values from an ArraySpec

INTERFACE:

    subroutine ESMF_ArraySpecGet(arrayspec, rank, typekind, rc)

ARGUMENTS:

    type(ESMF_ArraySpec), intent(inout) :: arrayspec
    integer, intent(out), optional :: rank
    type(ESMF_TypeKind), intent(out), optional :: typekind
    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       Array rank (dimensionality – 1D, 2D, etc). Maximum possible is 7D.
typekind   Array typekind. See section 9.2.1 for valid values.
[rc ]  Return code; equals ESMF_SUCCESS if there are no errors.

22.5.2 ESMF_ArraySpecSet - Set values for an ArraySpec

INTERFACE:

    subroutine ESMF_ArraySpecSet(arrayspec, rank, typekind, rc)

ARGUMENTS:

    type(ESMF_ArraySpec), intent(inout) :: arrayspec
    integer, intent(in) :: rank
    type(ESMF_TypeKind), intent(in) :: typekind
    integer, intent(out), optional :: rc

DESCRIPTION:

Creates a description of the data – the typekind, the rank, and the dimensionality. The arguments are:

arrayspec  The ESMF_ArraySpec to set.
rank       Array rank (dimensionality – 1D, 2D, etc). Maximum allowed is 7D.
typekind   Array typekind. See section 9.2.1 for valid values.
[rc ]  Return code; equals ESMF_SUCCESS if there are no errors.
22.5.3 ESMF_ArraySpecValidate - Validate ArraySpec internals

INTERFACE:

    subroutine ESMF_ArraySpecValidate(arrayspec, rc)

ARGUMENTS:

    type(ESMF_ArraySpec), intent(inout) :: arrayspec
    integer, intent(out), optional :: rc

DESCRIPTION:

Validates that the arrayspec is internally consistent. The method returns an error code if problems are found. The arguments are:

arrayspec Specified ESMF_ArraySpec object.

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

22.5.4 ESMF_ArraySpecPrint - Print information of ArraySpec

INTERFACE:

    subroutine ESMF_ArraySpecPrint(arrayspec, rc)

ARGUMENTS:

    type(ESMF_ArraySpec), intent(in) :: arrayspec
    integer, intent(out), optional :: rc

DESCRIPTION:

Print ArraySpec internals.

Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.

The arguments are:

arrayspec Specified ESMF_ArraySpec object.

23 Grid Class

23.1 Description

The ESMF Grid class is used to describe the geometry and discretization of logically rectangular physical grids. It also contains the description of the grid’s underlying topology and the decomposition of the physical grid across the available computational resources. The most frequent use of the Grid class is to describe physical grids in user code so that sufficient information is available to perform ESMF methods such as regridding.

In the current release (v3.1.0) the functionality in this class is partially implemented. Multi-tile grids are not supported, and edge connectivities are not implemented and default to periodic. Both regular and irregular distributions can be defined, but arbitrary distributions have not yet been implemented. Other constraints of the current implementation are noted in the usage section and in the API descriptions.
Key Features
Representation of grids formed by logically rectangular regions, including uniform and rectilinear grids (e.g. lat-lon grids), curvilinear grids (e.g. displaced pole grids), and grids formed by connected logically rectangular regions (e.g. cubed sphere grids) [CONNECTED REGIONS ARE NOT YET SUPPORTED].
Support for 1D, 2D, 3D, and higher dimension grids.
Distribution of grids across computational resources for parallel operations - users set which grid dimensions are distributed.
Grids can be created already distributed, so that no single resource needs global information during the creation process.
Options to define periodicity and other edge connectivities either explicitly or implicitly via shape shortcuts [EDGE CONNECTIVITIES CURRENTLY DEFAULT TO APERIODIC BOUNDS].
Options for users to define grid coordinates themselves or call prefabricated coordinate generation routines for standard grids [NO GENERATION ROUTINES YET].
Options for incremental construction of grids.
Options for using a set of pre-defined stagger locations or for setting custom stagger locations.

23.1.1 Grid Representation in ESMF
ESMF Grids are based on the concepts described in A Standard Description of Grids Used in Earth System Models [Balaji 2006]. In this document Balaji introduces the mosaic concept as a means of describing a wide variety of Earth system model grids. A mosaic is composed of grid tiles connected at their edges. Mosaic grids includes simple, single tile grids as a special case.
The ESMF Grid class is a representation of a mosaic grid. Each ESMF Grid is constructed of one or more logically rectangular Tiles. A Tile will usually have some physical significance (e.g. the region of the world covered by one face of a cubed sphere grid).
The piece of a Tile that resides on one DE (for simple cases, a DE can be thought of as a processor - see section on the DELayout) is called a LocalTile. For example, the six faces of a cubed sphere grid are each Tiles, and each Tile can be divided into many LocalTiles.
Every ESMF Grid contains a DistGrid object, which defines the Grid’s index space, topology, distribution, and connectivities. It enables the user to define the complex edge relationships of tripole and other grids. The DistGrid can be created explicitly and passed into a Grid creation routine, or it can be created implicitly if the user takes a Grid creation shortcut. Options for grid creation are described in more detail in section 23.1.8.

23.1.2 Supported Grids
The range of supported grids in ESMF can be defined by:
- Types of topologies and shapes supported. ESMF supports one or more logically rectangular grid Tiles with connectivities specified between cells. For more details see section 23.1.3.
- Types of distributions supported. ESMF supports regular, irregular, or arbitrary distributions of data. Note that arbitrary distributions have not been implemented as of v3.1.0. For more details see section 23.1.4.
- Types of coordinates supported. ESMF supports uniform, rectilinear, and curvilinear coordinates. For more details see section 23.1.5.

23.1.3 Grid Topologies and Periodicity
ESMF has shortcuts for the creation of standard Grid topologies or shapes up to 3D. In many cases, these enable the user to bypass the step of creating a DistGrid before creating the Grid. The basic call is ESMF_GridCreateShapeTile(). With this call, the user can specify for each dimension whether there is no connection, it is periodic, it is a pole, or it is a bipole. The assumed connectivities for poles and bipoles are described in section 23.5.1. Connectivities are specified using the ESMF_GridConn parameter, which has values such as ESMF_GRIDCONN_PERIODIC.
ESMF Grids have several options for data distribution (also referred to as decomposition). The user specifies which dimensions are to be distributed through a coordinate dependency (coordDep) argument. The main distribution options are regular, irregular, and arbitrary. A regular distribution is one in which the same number of contiguous grid points are assigned to each DE in the distributed dimension. An irregular distribution is one in which unequal numbers of contiguous gridpoints are assigned to each DE in the distributed dimension. An arbitrary distribution is one in which any gridpoint can be assigned to any DE. Any of these distribution options can be applied to any of the grid shapes (i.e., rectangle) or types (i.e., rectilinear). Arbitrary distributions have not been implemented as of v3.1.0.

Figure 12 illustrates options for distribution. A distribution can also be specified using the DistGrid, by passing object into a Grid create call.

### 23.1.5 Grid Coordinates

Grid Tiles can have uniform, rectilinear, or curvilinear coordinates. The coordinates of uniform grids are equally spaced along their axes, and can be fully specified by the coordinates of the two opposing points that define the grid’s physical span. The coordinates of rectilinear grids are unequally spaced along their axes, and can be fully specified by giving the spacing of grid points along each axis. The coordinates of curvilinear grids must be specified by giving the explicit set of coordinates for each grid point. Curvilinear grids are often uniform or rectilinear grids that have been warped; for example, to place a pole over a land mass so that it does not affect the computations performed on an ocean model grid. Figure 13 shows examples of each type of grid.

Any of these logically rectangular grid types can be combined through edge connections to form a mosaic. Cubed sphere and yin-yang grids are examples of mosaic grids. Note that as of v3.1.0 multi-tile grids have not yet been implemented as of v3.1.0.
Uniform grid
Rectilinear grid
Curvilinear grid

Figure 13: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

implemented.
Each of these coordinate types can be set for each of the standard grid shapes described in section 23.1.3.
The table below shows how examples of common single Tile grids fall into this shape and coordinate taxonomy. Note that any of the grids in the table can have a regular or arbitrary distribution.

<table>
<thead>
<tr>
<th>Shape</th>
<th>Uniform</th>
<th>Rectilinear</th>
<th>Curvilinear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>Global uniform lat-lon grid</td>
<td>Gaussian grid</td>
<td>Displaced pole grid</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Regional uniform lat-lon grid</td>
<td>Gaussian grid section</td>
<td>Polar stereographic grid section</td>
</tr>
</tbody>
</table>

23.1.6 Coordinate Specification and Generation

There are two ways of specifying coordinates in ESMF. The first way is for the user to set the coordinates. The second way is to take a shortcut and have the framework generate the coordinates.
No ESMF generation routines are currently available.
See Section 23.2.7 for more description and examples of setting coordinates.

23.1.7 Staggering

Staggering is a finite difference technique in which the values of different physical quantities are placed at different locations within a grid cell.
The ESMF Grid class supports a variety of stagger locations, including cell centers, corners, and edge centers. Combinations of these are sufficient to specify any of the Arakawa staggers. ESMF also supports staggering in 3D and higher dimensions. There are shortcuts for standard staggers, and interfaces through which users can create custom staggers.
As a default the ESMF Grid class provides symmetric staggering, so that cell centers are enclosed by cell perimeter (e.g. corner) stagger locations. This means the coordinate arrays for stagger locations other than the center will have an additional element of padding in order to enclose the cell center locations. However, to achieve other types of staggering, the user may alter or eliminate this padding by using the appropriate options when adding coordinates to a Grid.
For examples and a full description of the stagger interface see Section 23.1.7.

23.1.8 Options for Building Grids

ESMF Grid objects must represent a wide range of grid types and use cases, some of them quite complex. As a result, multiple ways to build Grid objects are required. This section describes the stages to building Grids, the options for each stage, and typical calling sequences.
In ESMF there are two main stages to building Grids. The ESMF_GridStatus value stored within the Grid object reflects the stage the Grid has attained (see Section 23.5.2). These stages are:

1. Create the Grid topology or shape. At the completion of this stage, the Grid has a specific topology and distribution, but empty coordinate arrays. The Grid can be used as the basis for allocating a Field. Its ESMF_GridStatus parameter has a value of ESMF_GRIDSTATUS_SHAPE_READY.

   The options for specifying the Grid shape are:
   • Use the ESMF_GridCreateShapeTile() shortcut method to specify the Grid size and dimension, and to select from a limited set of edge connectivities.
   • Create a DistGrid using the ESMF_DistGridCreate() method. This enables the user to specify connectivities in greater detail than using ESMF_GridCreateShapeTile(). Then pass the DistGrid into a general ESMF_GridCreate() method.

2. Specify the Grid coordinates and any other information required for regridding (this can vary depending on the particular regridding method). At the completion of this stage, the Grid can be used in a regridding operation (once Grid is connected to regrid; as of v3.1.0, it is not). Its ESMF_GridStatus has a value of ESMF_GRIDSTATUS_REGRID_READY.

When creating the Grid shape and specifying the Grid coordinates, the user can either specify all required information at once, or can provide information incrementally. The call ESMF_GridCreateEmpty() builds a Grid object container that can be filled in with subsequent calls to ESMF_GridSet() methods. A Grid that is empty has the ESMF_GridStatus value ESMF_GRIDSTATUS_NOT_READY.

The following table summarizes possible call sequences for building Grids.

<table>
<thead>
<tr>
<th>Create Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>From shape shortcut</td>
</tr>
<tr>
<td>grid = ESMF_GridCreateShapeTile(...)</td>
</tr>
<tr>
<td>Using DistGrid with general create interface</td>
</tr>
<tr>
<td>distgrid = ESMF_DistGridCreate(...)</td>
</tr>
<tr>
<td>grid = ESMF_GridCreate(distgrid, ...)</td>
</tr>
<tr>
<td>Incremental</td>
</tr>
<tr>
<td>grid = ESMF_GridCreateEmpty(...)</td>
</tr>
<tr>
<td>call ESMF_GridSetCommitShapeTile(grid, ...)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set coordinates by copy or reference</td>
</tr>
<tr>
<td>call ESMF_GridSetCoord(grid, ...)</td>
</tr>
<tr>
<td>Retrieve ESMF Array of coordinates from Grid and set values</td>
</tr>
<tr>
<td>call ESMF_GridGetCoord(grid, esmfArray, ...), set values</td>
</tr>
<tr>
<td>Retrieve local bounds and native array from Grid and set values</td>
</tr>
<tr>
<td>call ESMF_GridGetCoord(grid, lbound, ubound, array), set values</td>
</tr>
</tbody>
</table>

23.2 Use and Examples

This section describes the use of the ESMF Grid class. It first discusses the more user friendly shape specific interface to the Grid. During this discussion it covers creation and options, adding stagger locations, coordinate data access, and other grid functionality. After this initial phase the document discusses the more advanced options which the user can employ should they need more customized interaction with the Grid class.

23.2.1 Shortcut Creation Method for Single-Tile Grids

The method ESMF_GridCreateShapeTile() is a shortcut for building single tile logically rectangular Grids up to three dimensions. It is partially implemented. The user can specify Grid size, dimension and distribution, but cannot specify tile edge connectivities yet. The default is that Grid edges are not connected. Once completed, this method will enable users to create many common grid shapes, including rectangle, bipole sphere, and tripole sphere.
The ESMF_GridCreateShapeTile() method will eventually support the three types of distributions described in Section 23.1.4. It currently supports two of these types: regular and irregular.

To create a Grid with a regular distribution the user specifies the global maximum and minimum ranges of the Grid cell index space (maxIndex and minIndex), and the number of pieces in which to partition each dimension (via a regDecomp argument). ESMF then divides the index space as evenly as possible into the specified number of pieces. If there are cells left over then they are distributed one per DE starting from the first DE until they are gone. If minIndex is not specified, then the bottom of the Grid cell index range is assumed to be (1,1,...,1). If regDecomp is not specified, then by default ESMF creates a distribution that partitions the grid cells in the first dimension (e.g. NPx1x1...) as evenly as possible by the number of processors NP. The remaining dimensions are not partitioned. The dimension of the Grid is the size of maxIndex. The following is an example of creating a 10x20x30 3D grid where the first dimensions is broken into 2 pieces, the second is broken into 4 pieces, and the third is "distributed" across only one processor.

```plaintext
grid3D=ESMF_GridCreateShapeTile(regDecomp=(/2,4,1/), maxIndex=(/10,20,30/), &
rc=rc)
```

Irregular distribution requires the user to specify the exact number of Grid cells per DE in each dimension. In the ESMF_GridCreateShapeTile() call the countsPerDEDim1, countsPerDEDim2, and countsPerDEDim3 arguments are used to specify a rectangular distribution containing size(countsPerDEDim1) by size(countsPerDEDim2) by size(countsPerDEDim3) DEs. The entries in each of these arrays specify the number of grid cells per DE in that dimension. The dimension of the grid is determined by the presence of countsPerDEDim3. If it's present the Grid will be 3D. If just countsPerDEDim1 and countsPerDEDim2 are specified the Grid will be 2D.

The following call illustrates the creation of a 10x20 two dimensional rectangular Grid distributed across six DEs that are arranged 2x3. In the first dimension there are 3 grid cells on the first DE and 7 cells on the second DE. The second dimension has 3 DEs with 11,2, and 7 cells, respectively.

```plaintext
grid2D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/3,7/), &
countsPerDEDim2=(/11,2,7/), rc=rc)
```

To add a distributed third dimension of size 30, broken up into two groups of 15, the above call would be altered as follows.

```plaintext
grid3D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/3,7/), &
countsPerDEDim2=(/11,2,7/), countsPerDEDim3=(/15,15/), rc=rc)
```

To make a third dimension distributed across only 1 DE, then countsPerDEDim3 in the call should only have a single term.

```plaintext
grid3D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/3,7/), &
countsPerDEDim2=(/11,2,7/), countsPerDEDim3=(/30/), rc=rc)
```

The petMap parameter may be used to specify on to which specific PETs the DEs in the Grid are assigned. Note that this parameter is only available for the regular and irregular distribution types. The petMap array is a 3D array, for a 3D Grid each of its dimensions correspond to a Grid dimension. If the Grid is 2D, then the first two dimensions correspond to Grid dimensions and the last dimension should be of size 1. The size of each petMap dimension is the number of DE’s along that dimension in the Grid. For a regular Grid, the size is equal to the number in regDecomp (i.e. size(petMap,d)=regDecomp(d) for all dimensions d in the Grid). For an irregular Grid the size is equal to the number of items in the corresponding countsPerDEDim variable (i.e. size(petMap,d)=size(countsPerDEDimd) for all dimensions d in the Grid).

Each entry in petMap specifies to which PET the corresponding DE should be assigned. For example, petMap(3,2)=4 tells the Grid create call to put the DE located at column 3 row 2 on PET 4.

The following example demonstrates how to specify the PET to DE association for an ESMF_GridCreateShapeTile() call.
allocate memory for petMap
allocate( petMap(2,2,1) )

Set petMap
petMap(:,1,1) = (/3,2/) ! DE (1,1,1) on PET 3 and DE (2,1,1) on PET 2
petMap(:,2,1) = (/1,0/) ! DE (1,2,1) on PET 1 and DE (2,2,1) on PET 0

Let the 3D grid be distributed only in the first two dimensions.
grid2D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/3,7/), &
countsPerDEDim2=(/7,6/), petMap=petMap, rc=rc)

 Arbitrary distribution has not yet been implemented. The design is that the user specifies the global minimum and maximum ranges of the index space with the arguments minIndex and maxIndex, and through a localIndices argument specifies the set of index space locations residing on the local PET. Again, if minIndex is not specified, then the bottom of the index range is assumed to be (1,1,...). The dimension of the Grid is equal to the size of maxIndex.
The following example creates a 2D Grid of dimensions 5x5, and places the diagonal elements (i.e. indices (i,i) where i goes from 1 to 5) on the local PET. The remaining PETs would individually declare the remainder of the Grid locations.

allocate memory for local
allocate( localIndices(2,5) )
Set local indices
localIndices(:,1)=(/1,1/)
localIndices(:,2)=(/2,2/)
localIndices(:,3)=(/3,3/)
localIndices(:,4)=(/4,4/)
localIndices(:,5)=(/5,5/)

Create Grid
! grid2D=ESMF_GridCreateShapeTile(maxIndex=(/5,5/), & ! NOT YET IMPLEMENTED
! localIndices=localIndices, rc=rc)

23.2.2 Creating a 2D Regularly Distributed Rectilinear Grid With Uniformly Spaced Coordinates
The following is an example of creating a simple rectilinear grid and loading in a set of coordinates. It illustrates a straightforward use of the ESMF_GridCreateShapeTile() call described in the previous section. This code creates a 10x20 2D grid with uniformly spaced coordinates varying from (10,10) to (100,200). The grid is partitioned using a regular distribution. The first dimension is divided into two pieces, and the second dimension is divided into 3. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

!-------------------------------------------------- -----------------
! Create the Grid: Allocate space for the Grid object, define the topology and distribution of the Grid, and specify that it will have global indices. Note that aperiodic bounds are specified by default - if periodic bounds were desired they would need to be specified using an additional gridConn argument (which isn’t implemented yet). In this call the minIndex hasn’t been set, so it defaults to (1,1,...). The default is to divide the index range as equally as possible among the DEs

214
! specified in regDecomp. This behavior can be changed by
! specifying decompFlag.
!-------------------------------------------------- -----------------
grid2D=ESMF_GridCreateShapeTile( &
! Define a regular distribution
maxIndex=(/10,20/), & ! define index space
regDecomp=(/2,3/), & ! define how to divide among DEs
! Specify mapping of coords dim to Grid dim
coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
indexflag=ESMF_INDEX_GLOBAL, &
rc=rc)
!-------------------------------------------------- -----------------
! Allocate coordinate storage and associate it with the center
! stagger location. Since no coordinate values are specified in
! this call no coordinate values are set yet.
!-------------------------------------------------- -----------------
call ESMF_GridAddCoord(grid2D, &
  staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)
!-------------------------------------------------- -----------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
  staggerloc=ESMF_STAGGERLOC_CENTER, &
  computationalLBound=lbnd, computationalUBound=ubnd, fptr=coordX, rc=rc)
!-------------------------------------------------- -----------------
! Calculate and set coordinates in the first dimension [10-100].
!-------------------------------------------------- -----------------
do i=lbnd(1),ubnd(1)
  coordX(i) = i*10.0
enddo
!-------------------------------------------------- -----------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
  staggerloc=ESMF_STAGGERLOC_CENTER, &
  computationalLBound=lbnd, computationalUBound=ubnd, fptr=coordY, rc=rc)
!-------------------------------------------------- -----------------
! Calculate and set coordinates in the second dimension [10-200]
!-------------------------------------------------- -----------------
do j=lbnd(1),ubnd(1)
  coordY(j) = j*10.0
enddo

The remaining examples in this section will use the irregular distribution because of its greater generality. To create
code similar to these, but using a regular distribution, replace the countsPerDEDim arguments in the Grid create
with the appropriate maxIndex and regDecomp arguments.
### 23.2.3 Creating a 2D Irregularly Distributed Rectilinear Grid With Uniformly Spaced Coordinates

This example serves as an illustration of the difference between using a regular and irregular distribution. It repeats the previous example except using an irregular distribution to give the user more control over how the cells are divided between the DEs. As before, this code creates a 10x20 2D Grid with uniformly spaced coordinates varying from (10,10) to (100,200). In this example, the Grid is partitioned using an irregular distribution. The first dimension is divided into two pieces, the first with 3 Grid cells per DE and the second with 7 Grid cells per DE. In the second dimension, the Grid is divided into 3 pieces, with 11, 2, and 7 cells per DE respectively. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```fortran
!-------------------------------------------------- -----------------
! Create the Grid: Allocate space for the Grid object, define the  ! topology and distribution of the Grid, and specify that it  ! will have global coordinates. Note that aperiodic bounds are  ! specified by default - if periodic bounds were desired they  ! would need to be specified using an additional gridConn argument  ! (which isn’t implemented yet). In this call the minIndex hasn’t  ! been set, so it defaults to (1,1,...).
!-------------------------------------------------- -----------------
grid2D=ESMF_GridCreateShapeTile( &
  ! Define an irregular distribution
  countsPerDEDim1=(/3,7/), &
  countsPerDEDim2=(/11,2,7/), &
  ! Specify mapping of coords dim to Grid dim
  coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
  coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
  indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)

!-------------------------------------------------- -----------------
! Allocate coordinate storage and associate it with the center  ! stagger location. Since no coordinate values are specified in  ! this call no coordinate values are set yet.
!-------------------------------------------------- -----------------
call ESMF_GridAddCoord(grid2D, &
  staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)

!-------------------------------------------------- -----------------
! Get the pointer to the first coordinate array and the bounds  ! of its global indices on the local DE.
!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
  staggerloc=ESMF_STAGGERLOC_CENTER, &
  computationalLBound=lbnd, computationalUBound=ubnd, fptr=coordX, rc=rc)

!-------------------------------------------------- -----------------
! Calculate and set coordinates in the first dimension [10-100].
!-------------------------------------------------- -----------------
do i=lbnd(1),ubnd(1)
  coordX(i) = i*10.0
enddo
```

216
23.2.4 Creating a 2D Irregularly Distributed Grid With Curvilinear Coordinates

The following is an example of creating a simple curvilinear Grid and loading in a set of coordinates. It creates a 10x20 2D Grid where the coordinates vary along every dimension. The Grid is partitioned using an irregular distribution. The first dimension is divided into two pieces, the first with 3 Grid cells per DE and the second with 7 Grid cells per DE. In the second dimension, the Grid is divided into 3 pieces, with 11, 2, and 7 cells per DE respectively. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```fortran
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
   staggerloc=ESMF_STAGGERLOC_CENTER, &
   computationalLBound=lbnd, computationalUBound=ubnd, fptr=coordY, rc=rc)
!-------------------------------------------------- -----------------
! Calculate and set coordinates in the second dimension [10-200]
!-------------------------------------------------- -----------------
do j=lbnd(1),ubnd(1)
   coordY(j) = j*10.0
endo
```

```fortran
!-------------------------------------------------- -----------------
! Create the Grid: Allocate space for the Grid object, define the
! distribution of the Grid, and specify that it
! will have global coordinates. Note that aperiodic bounds are
! specified by default - if periodic bounds were desired they
! would need to be specified using an additional gridConn argument
! (which isn't implemented yet). In this call the minIndex hasn't
! been set, so it defaults to (1,1,...).
!-------------------------------------------------- -----------------
grid2D=ESMF_GridCreateShapeTile( &
   ! Define an irregular distribution
   countsPerDEDim1=(/3,7/), &
   countsPerDEDim2=(/11,2,7/), &
   ! Specify mapping of coords dim to Grid dim
   coordDep1=(/1,2/), & ! 1st coord is 2D and depends on both Grid dim
   coordDep2=(/1,2/), & ! 2nd coord is 1D and depends on both Grid dim
   indexflag=ESMF_INDEX_GLOBAL, &
   rc=rc)
!-------------------------------------------------- -----------------
! Allocate coordinate storage and associate it with the center
! stagger location. Since no coordinate values are specified in
! this call no coordinate values are set yet.
!-------------------------------------------------- -----------------
call ESMF_GridAddCoord(grid2D, &
   staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)
!-------------------------------------------------- -----------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
```
23.2.5 Creating an Irregularly Distributed Rectilinear Grid with a Non-Distributed Vertical Dimension

This example demonstrates how a user can build a rectilinear horizontal Grid with a non-distributed vertical dimension. The Grid contains both the center and corner stagger locations (i.e. Arakawa B-Grid). In contrast to the previous examples, this example doesn’t assume that the code is being run with a 1-1 mapping between PETs and DEs. It should work when run on any number of PETs.

!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
   staggerloc=ESMF_STAGGERLOC_CENTER, &
   computationalLBound=lbnd, computationalUBound=ubnd, fpdr=coordX2D, rc=rc)
!-------------------------------------------------- -----------------
! Calculate and set coordinates in the first dimension [10-100].
!-------------------------------------------------- -----------------
do j=lbnd(2),ubnd(2)
do i=lbnd(1),ubnd(1)
   coordX2D(i,j) = i+j
enddo
enddo
!
!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
   staggerloc=ESMF_STAGGERLOC_CENTER, &
   computationalLBound=lbnd, computationalUBound=ubnd, fpdr=coordY2D, rc=rc)
!
! Calculate and set coordinates in the second dimension [10-200]
!
!
do j=lbnd(2),ubnd(2)
do i=lbnd(1),ubnd(1)
   coordY2D(i,j) = j-i/100.0
enddo
enddo

grid3D=ESMF_GridCreateShapeTile( &
   ! Define an irregular distribution
   countsPerDEDim1=(/45,75,40,20/), &
   countsPerDEDim2=(/30,40,20/), &
   countsPerDEDim3=(/40/), &
   ! Specify mapping of coords dim to Grid dim
   coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
coordDep3=(/3/), & ! 3rd coord is 1D and depends on 3rd Grid dim
indexflag=ESMF_INDEX_GLOBAL, & ! Use global indices
rc=rc)

!-------------------------------------------------- -----------------
! Allocate coordinate storage for both center and corner stagger
! locations. Since no coordinate values are specified in this
! call no coordinate values are set yet.
!-------------------------------------------------- -----------------
call ESMF_GridAddCoord(grid3D, &
    staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER, rc=rc)
call ESMF_GridAddCoord(grid3D, &
    staggerloc=ESMF_STAGGERLOC_CORNER_VCENTER, rc=rc)

!-------------------------------------------------- -----------------
! Get the number of DEs on this PET, so that the program
! can loop over them when accessing data.
!-------------------------------------------------- -----------------
call ESMF_GridGet(grid3D, localDECount=localDECount, rc=rc)

!-------------------------------------------------- -----------------
! Loop over each localDE when accessing data
!-------------------------------------------------- -----------------
do lDE=0,localDECount-1

!-------------------------------------------------- -----------------
! Fill in the coordinates for the corner stagger location first.
!-------------------------------------------------- -----------------
call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=lDE, &
    staggerLoc=ESMF_STAGGERLOC_CORNER_VCENTER, &
    computationalLBound=lbnd_corner, &
    computationalUBound=ubnd_corner, &
    fptr=cornerX, rc=rc)

!-------------------------------------------------- -----------------
! Calculate and set coordinates in the first dimension.
!-------------------------------------------------- -----------------
do i=lbnd_corner(1),ubnd_corner(1)
    cornerX(i) = (i-1)*(360.0/180.0)
endo
call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=lDE, &
    staggerLoc=ESMF_STAGGERLOC_CORNER_VCENTER, &
    computationalLBound=lbnd_corner, &
    computationalUBound=ubnd_corner, &
    fptr=cornerY, rc=rc)

!-------------------------------------------------- --------------
! Calculate and set coordinates in the second dimension.
!-------------------------------------------------- --------------
do j=lbnd_corner(1),ubnd_corner(1)
    cornerY(j) = (j-1)*(180.0/90.0)
enddo

!-------------------------------------------------- --------------
! Get the local bounds of the global indexing for the third
! coordinate array on the local DE, and the pointer to the array.
!-------------------------------------------------- --------------
call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=lDE, &
    staggerLoc=ESMF_STAGGERLOC_CENTER_VCENTER, &
    computationalLBound=lbnd, computationalUBound=ubnd,&
    fptr=cornerZ, rc=rc)

!-------------------------------------------------- --------------
! Calculate and set the vertical coordinates
!-------------------------------------------------- --------------
do k=lbnd(1),ubnd(1)
    cornerZ(k) = 4000.0*( (1./39.)*(k-1) )**2
  enddo

!-------------------------------------------------- -------------
! Now fill the coordinates for the center stagger location with
! the average of the corner coordinate location values.
!-------------------------------------------------- -------------
!-------------------------------------------------- --------------
! Get the local bounds of the global indexing for the first
! coordinate array on the local DE, and the pointer to the array.
!-------------------------------------------------- --------------
call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=lDE, &
    staggerLoc=ESMF_STAGGERLOC_CENTER_VCENTER, &
    computationalLBound=lbnd, computationalUBound=ubnd, &
    fptr=centerX, rc=rc)

!-------------------------------------------------- --------------
! Calculate and set coordinates in the first dimension.
!-------------------------------------------------- --------------
do i=lbnd(1),ubnd(1)
    centerX(i) = 0.5*(i-1 + i)*(360.0/180.0)
  enddo

!-------------------------------------------------- --------------
! Get the local bounds of the global indexing for the second
! coordinate array on the local DE, and the pointer to the array.
!-------------------------------------------------- --------------
call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=lDE, &
    staggerLoc=ESMF_STAGGERLOC_CENTER_VCENTER, &
    computationalLBound=lbnd, computationalUBound=ubnd, &
    fptr=cornerX, rc=rc)
23.2.6 Creating an Empty Grid in a Parent Component for Completion in a Child Component

ESMF Grids can be created incrementally. To do this, the user first calls ESMF_GirdCreateEmpty to allocate the shell of a Grid. Next, we use the ESMF_GridSetCommitShapeTile() call that fills in the Grid and does an internal commit to make it usable. For consistency's sake the initial ESMF_GirdCreateEmpty call must occur on the same or a superset of the PETs as the ESMF_GridSetCommitShapeTile() calls. The following example uses the incremental technique to create a rectangular 10x20 Grid with coordinates at the center and corner stagger locations.

!-------------------------------------------------- -------------------------
! IN THE PARENT COMPONENT:
! Create an empty Grid in the parent component for use in a child component.  
! The parent may be defined on more PETs than the child component.  
! The child's [vm or pet list] is passed into the create call so that  
! the Grid is defined on the appropriate subset of the parent's PETs.  
!-------------------------------------------------- -------------------------
grid2D=ESMF_GirdCreateEmpty(rc=rc)

!-------------------------------------------------- -------------------------
! IN THE CHILD COMPONENT:
! Set the Grid topology. Here we define an irregularly distributed  
! rectangular Grid.  
!-------------------------------------------------- -------------------------
call ESMF_GridSetCommitShapeTile(grid2D, countsPerDEDim1=(/6,4/), &
   countsPerDEDim2=(/10,3,7/), rc=rc)

!-------------------------------------------------- -------------------------
! Add Grid coordinates at the cell center location.
!-------------------------------------------------- -------------------------
call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CENTER, rc=rc)

!-------------------------------------------------- -------------------------
! Add Grid coordinates at the corner stagger location.
!-------------------------------------------------- -------------------------
call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, rc=rc)

23.2.7 Associating Coordinates with Stagger Locations

A useful finite difference technique is to place different physical quantities at different locations within a grid cell. This staggering of the physical variables on the mesh is introduced so that the difference of a field is naturally defined at the location of another variable. This method was first formalized by Mesinger and Arakawa (1976). To support the staggering of variables, the Grid provides the idea of stagger locations. Stagger locations refer to the places in a Grid cell that contain coordinates and, once a Grid is associated with a Field object, field data. Typically data can be located at the cell center, at the cell corners, or at the cell faces, in 2D, 3D, and higher dimensions. (Note that any Arakawa stagger can be constructed of a set of Grid stagger locations.) Users can put coordinates, which are necessary for operations such as regrid, at multiple stagger locations in a Grid. In addition, the user can put Field data at any of the stagger locations in a Grid.

By default the coordinate array at the center stagger location starts at the bottom index of the Grid (default (1,1...1)) and extends up to the maximum cell index in the Grid (e.g. given by the maxIndex argument). Other stagger locations also start at the bottom index of the Grid, however, they can extend to +1 element beyond the center in some dimensions to allow for the extra space to surround the center elements. See Section 23.2.16 for a description of this extra space and how to adjust if it necessary. The subroutine ESMF_GridGetCoord can be used to retrieve the stagger bounds for the piece of a coordinate array on a particular DE.

The user can allocate coordinate arrays without setting coordinates using the ESMF_AddCoord() call, or allocate and set coordinates in a single call with ESMF_SetCoord(). When adding or accessing coordinate data, the stagger location is specified to tell the Grid method where in the cell to get the data. There are predefined stagger locations (see Section 23.5.3), or, should the user wish to specify their own, there is also a set of methods for generating custom locations (See Section 23.2.16).

The following example adds coordinate storage to the corner stagger location in a grid using one of the predefined stagger locations.

call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, rc=rc)

23.2.8 Specifying the Relationship of Coordinate Arrays to Index Space Dimensions

To specify how the coordinate arrays are mapped to the index dimensions the arguments coordDep1, coordDep2, and coordDep3 are used, each of which is a Fortran array. The values of the elements in a coordDep array specify which index dimension the corresponding coordinate dimension maps to. For example, coordDep1=/(1,2/) means that the first dimension of coordinate 1 maps to index dimension 1 and the second maps to index dimension 2. If the coordDep arrays are not specified, then coordDep1 defaults to /1,2..gridDimCount/. This default thus specifies a curvilinear grid.

The following call demonstrates the creation of a 10x20 2D rectilinear grid where the first coordinate component is mapped to the second index dimension (i.e. is of size 20) and the second coordinate component is mapped to the first index dimension (i.e. is of size 10).
grid2D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/5,5/), &
countsPerDEDim2=(/7,7,6/), &
coordDep1=(/2/), &
coordDep2=(/1/), rc=rc)

The following call demonstrates the creation of a 10x20x30 2D plus 1 curvilinear grid where coordinate component 1 and 2 are still 10x20, but coordinate component 3 is mapped just to the third index dimension.

grid2D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/6,4/), &
countsPerDEDim2=(/10,7,3/), countsPerDEDim3=(/30/), &
coordDep1=(/1,2/), coordDep2=(/1,2/), &
coordDep3=(/3/), rc=rc)

By default the local piece of the array on each PET starts at (1,1,...), however, the indexing for each grid coordinate array on each DE may be shifted to the global indices by using the indexflag. For example, the following call switches the grid to use global indices.

grid2D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/6,4/), &
countsPerDEDim2=(/10,7,3/), indexflag=ESMF_INDEX_GLOBAL, rc=rc)

23.2.9 Accessing Grid Coordinates

Once a Grid has been created, the user has several options to access the Grid coordinate data. The first of these, ESMF_GridSetCoord(), enables the user to set data for one stagger location across the whole Grid, using ESMF Arrays. For example, the following sets the coordinates in the first dimension (e.g. x) for the center stagger location to those in the ESMF Array arrayCoordX.

call ESMF_GridSetCoord(grid2D, &
  staggerLoc=ESMF_STAGGERLOC_CORNER, &
  coordDim=1, array=arrayCoordX, rc=rc)

The method ESMF_GridGetCoord() allows the user to access the Array, as a direct reference, which contains the coordinate data for a stagger location on a Grid. The user can then employ any of the standard ESMF_Array tools to operate on the data. The following copies the coordinates from the second component of the corner and puts it into the ESMF Array arrayCoordY.

call ESMF_GridGetCoord(grid2D, &
  staggerLoc=ESMF_STAGGERLOC_CORNER, &
  coordDim=2, &
  array=arrayCoordY, rc=rc)

Alternatively, the call ESMF_GridGetCoord() gets a Fortran pointer to the coordinate data. The user can then operate on this array in the usual manner. The following call gets a reference to the Fortran array which holds the data for the second coordinate (e.g. y).

call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
  staggerLoc=ESMF_STAGGERLOC_CORNER, fptr=coordY2D, rc=rc)
23.2.10 Grid Regions and Bounds

The index space of the Grid contains three regions with associated bounds. This section describes these regions and bounds in general. The next couple of sections describe getting the bound information for specific types of Grid information.

The exclusive region is the index space defined by the distgrid of the Grid. These correspond to the maximum index space usable by any stagger location in the Grid. The size of the exclusive region is the index space for the Grid cells, plus the stagger padding.

The default stagger padding depends on the topology of the Grid. For an unconnected dimension the stagger padding is a width of 1 on the upper side (i.e. gridEdgeUWidth=(1,1,1,1...)). For a periodic dimension there is no stagger padding. By adjusting gridEdgeLWidth and gridEdgeUWidth, the user can set the stagger padding for the whole Grid and thus the exclusive region can be adjusted at will around the index space corresponding to the cells. The user can also use staggerEdgeLWidth and staggerEdgeUWidth to adjust individual stagger location padding within the Grid’s padding (Please see Section 23.2.17 for further discussion of customizing the stagger padding).

Figure 14 shows an example of a Grid’s exclusive region (with default stagger padding). This exclusive region would be for a Grid generated by either of the following calls:

```cpp
grid2D=ESMF_GridCreateShapeTile(regDecomp=(/2,4/), maxIndex=(/5,15/), &
    indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

```cpp
grid2D=ESMF_GridCreateShapeTile(countsPerDEDim1=(/4,4,4,3/), &
    countsPerDEDim2=(/3,2/), indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

Each rectangle in this diagram represents a DE and the numbers along the sides are the index values of the locations in the DE. Note that the exclusive region has one extra index location in each dimension than the number of cells because of the padding for the larger stagger locations (e.g. corner). The Grid computational region is a subset of the Grid exclusive region. The computational region indicates which index locations in the Grid are active for a particular stagger. The computational region is typically the region that a user would compute over (e.g. would iterate over setting values for).
Figure 15: An example of a Grid’s computational region for the center stagger location.

Figure 15 shows an example of a Grid computational region. This example is for the same Grid used for the exclusive region figure above (Figure 14). The computational region is the interior blue area. This example is for the center stagger location. This would be the computational region for a stagger location allocated by the following call:

```
call ESMF_GridAddCoord(grid2D, staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)
```

Figure 16 shows another example of a Grid computational region. This example is for the same Grid used for the exclusive region figure above (Figure 14). The computational region is the interior blue area. This example is for the edge1 stagger location. This would be the computational region for a stagger location allocated by the following call:

```
call ESMF_GridAddCoord(grid2D, staggerloc=ESMF_STAGGERLOC_EDGE1, rc=rc)
```

Again each rectangle in these diagrams represents a DE and the numbers along the sides are the index values of the locations in the DE.

The total region is the outermost boundary of the memory allocated on each DE to hold the data for the Grid on that DE. This region can be as small as the computational region, but may be larger to include space for halos, memory padding, etc. The total region is what is enlarged to include space for halos, and the total region must be large enough to contain the maximum halo operation on the Grid. The total region is a property of an underlying Array and so information about it is only retrievable when an Array has been set or allocated.

Figure 17 shows an example of a Grid total region. This example is for the same Grid used for the exclusive region figure above (Figure 14). The total region is the outer red area. This region is for the center stagger location. Total width adjustment of a stagger location allocation is not currently implemented, but when it is, this would be the total region for a stagger location allocated by the following call:

```
call ESMF_GridAddCoord(grid2D, staggerloc=ESMF_STAGGERLOC_CENTER, &
                         totalLWidth=(/1,1/), totalUWidth=(/1,1/), &
                         ! NOT YET IMPLEMENTED
                         rc=rc)
```
Figure 16: An example of a Grid’s computational region for the edge1 stagger location.

Figure 17: An example of a Grid’s total region for the center stagger location.
Figure 18: An example of a Grid’s total region for the edge1 stagger location.

Figure 18 shows an example of a Grid total region. This example is for the same Grid used for the exclusive region figure above (Figure 14). The total region is the outer red area. This region is for the edge1 stagger location. Total width adjustment of a stagger location allocation is not currently implemented, but when it is, this would be the total region for a stagger location allocated by the following call:

```latex
\textbf{call} ESMF_GridAddCoord(grid2D, staggerloc=ESMF_STAGGERLOC_CENTER, &
! \text{totalLWidth}=(/1,1/), \text{totalUWidth}=(/1,1/), & ! NOT YET IMPLEMENTED
rc=rc)
```

Again each rectangle in these diagrams represents a DE and the numbers along the sides are the index values of the locations in the DE.

The user can retrieve a set of bounds for each index space region described above: exclusive bounds, computational bounds, and total bounds. Note that although some of these are similar to bounds provided by ESMF_Array subroutines (see Section 20.2.6) the format here is different. The Array bounds are only for distributed dimensions and are ordered to correspond to the dimension order in the associated DistGrid. The bounds provided by the Grid are ordered according to the order of dimensions of the data in question. This means that the bounds provided should be usable "as is" to access the data.

Each of the three types of bounds refers to the maximum and minimum per dimension of the index ranges of a particular region. The parameters referring to the maximums contain a 'U' for upper. The parameters referring to the minimums contain an 'L' for lower. The bounds and associated quantities are almost always given on a per DE basis. The three types of bounds exclusiveBounds, computationalBounds, and totalBounds refer to the ranges of the exclusive region, the computational region, and the total region. Each of these bounds also has a corresponding count parameter which gives the number of items across that region (on a DE) in each dimension. (e.g. totalCount(d)=totalULBound(i)-totalLBound(i)+1). Width parameters give the spacing between two different types of region. The computationalWidth argument gives the spacing between the exclusive region and the computational region. The totalWidth argument gives the spacing between the total region and the computational region. Like the other bound information these are typically on a per DE basis, for example specifying totalLWidth=(1,1) makes the bottom of the total region one lower in each dimension than the computational region on each DE. The exceptions to the per DE rule are computationalEdgeWidth, staggerEdgeWidth, and gridEdgeWidth which give the spacing only on the DEs along the boundary of the Grid.
23.2.11 Getting Grid Coordinate Bounds

When operating on coordinates the user may often wish to retrieve the bounds of the piece of coordinate data on a particular local DE. This is useful for iterating through the data to set coordinates, retrieve coordinates, or do calculations. The method ESMF_GridGetCoord allows the user to retrieve bound information for a particular coordinate array. As described in the previous section there are three types of bounds the user can get: exclusive bounds, computational bounds, and total bounds. The exclusive and computational bounds may be retrieved from an unallocated stagger location, but the total bounds may only be retrieved from a stagger location which contains a valid coordinate Array. The bounds provided by ESMF_GridGetCoord are for both distributed and undistributed dimensions and are ordered according to the order of dimensions in the coordinate. This means that the bounds provided should be usable "as is" to access data in the coordinate array. In the case of factorized coordinate Arrays where a coordinate may have a smaller dimension than its associated Grid, then the dimension of the coordinate’s bounds are the dimension of the coordinate, not the Grid.

The following is an example of retrieving the bounds for localDE 0 for the first coordinate array from the corner stagger location.

```fortran
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
    staggerLoc=ESMF_STAGGERLOC_CORNER, &
    exclusiveLBound=elbnd, exclusiveUBound=eubnd, &
    computationalLBound=clbnd, computationalUBound=cubnd, &
    totalLBound=tlbnd, totalUBound=tubnd, rc=rc)
```

23.2.12 Getting Grid Stagger Location Bounds

When operating on data stored at a particular stagger in a Grid the user may find it useful to be able to retrieve the bounds of the data on a particular local DE. This is useful for iterating through the data for computations or allocating arrays to hold the data. The method ESMF_GridGet allows the user to retrieve bound information for a particular stagger location. As described in Section 23.2.10 there are three types of bounds the user can typically get, however, the Grid doesn’t hold data at a stagger location (that is the job of the Field), and so no Array is contained there and so no total region exists, so the user may only retrieve exclusive and computational bounds from a stagger location. The bounds provided by ESMF_GridGet are ordered according to the order of dimensions in the Grid.

The following is an example of retrieving the bounds for localDE 0 from the corner stagger location.

```fortran
call ESMF_GridGet(grid2D, localDE=0, &
    staggerLoc=ESMF_STAGGERLOC_CORNER, &
    exclusiveLBound=elbnd, exclusiveUBound=eubnd, &
    computationalLBound=clbnd, computationalUBound=cubnd, &
    rc=rc)
```

23.2.13 Getting Grid Stagger Location Information

In addition to the per DE information that can be accessed about a stagger location there is some global information that can be accessed by using ESMF_GridGet without specifying a localDE. One of the main uses of this information is to create an ESMF Array to hold data for a stagger location.

The information currently available from a stagger location is `computationalEdgeLWidth` and `computationalEdgeUWidth`. These give the difference between the lower and upper boundary of the exclusive region and the computational region.

The following is an example of retrieving information for localDE 0 from the corner stagger location.

```fortran
! Get info about staggerloc
call ESMF_GridGet(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, &
    computationalEdgeLWidth=celwidth, computationalEdgeUWidth=ceuwidth, &
    rc=rc)
```
23.2.14 Creating an Array at a Stagger Location

In order to create an Array to correspond to a Grid stagger location several pieces of information need to be obtained from both the Grid and the stagger location in the Grid.

The information that needs to be obtained from the Grid is the distgrid and distgridToGridMap to ensure that the new Array has the correct size and distribution and that its dimensions are mapped correctly to the Grid. These are obtained using the `ESMF_GridGet` method.

The information that needs to be obtained from the stagger location are the offsets from the edges of the exclusive region of the distgrid. These may be obtained from `ESMF_GridGet` by passing in the stagger location and the arguments `computationalEdgeLWidth` and `computationalEdgeUWidth`.

The following is an example of using information from the Grid to create an Array corresponding to a stagger location.

```fortran
! Get info from Grid
call ESMF_GridGet(grid2D, distgrid=distgrid, distgridToGridMap=distgridToGridMap, rc=rc)

! Get info about staggerloc
call ESMF_GridGet(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, &
    computationalEdgeLWidth=celwdth, computationalEdgeUWidth=ceuwdth, &
    rc=rc)

! construct ArraySpec
call ESMF_ArraySpecSet(arrayspec, rank=2, typekind=ESMF_TYPEKIND_R8, rc=rc)

! Create an Array based on the presence of distributed dimensions
array=ESMF_ArrayCreate(arrayspec=arrayspec, &
    distgrid=distgrid, distgridToArrayMap=distgridToGridMap, &
    computationalEdgeLWidth=celwdth, &
    computationalEdgeUWidth=ceuwdth, &
    rc=rc)
```

23.2.15 Creating More Complex Grids Using DistGrid

Besides the shortcut methods for creating a Grid object such as `ESMF_GridCreateShapeTile()`, there is a set of methods which give the user more control over the specifics of the grid. The following describes the more general interface, using DistGrid. The basic idea is to first create an ESMF DistGrid object describing the distribution and shape of the Grid, and then to employ that to either directly create the Grid or first create Arrays and then create the Grid from those. This method gives the user maximum control over the topology and distribution of the Grid. See the DistGrid documentation in Section 24.1 for an in-depth description of its interface and use.

As an example, the following call constructs a 10x20 Grid with a lower bound of (1,2).

```fortran
! Create DistGrid
distgrid2D = ESMF_DistGridCreate(minIndex=(/1,2/), maxIndex=(/11,22/), rc=rc)

! Create Grid
grid3D=ESMF_GridCreate(distGrid=distgrid2D, rc=rc)
```

To alter which dimensions are distributed, the `distgridToGridMap` argument can be used. The `distgridToGridMap` is used to set which dimensions of the Grid are mapped to the dimensions described by `maxIndex`. In other words, it describes how the dimensions of the underlying default DistGrid are mapped to the Grid. Each entry in `distgridToGridMap` contains the Grid dimension to which the corresponding DistGrid dimension should be mapped. The following example illustrates the creation of a Grid where the largest dimension is first. To accomplish this the two dimensions are swapped.
23.2.16 Specifying Custom Stagger Locations

Although ESMF provides a set of predefined stagger locations (See Section 23.5.3), the user may need one outside this set. This section describes the construction of custom stagger locations.

To completely specify stagger for an arbitrary number of dimensions, we define the stagger location in terms of a set of cartesian coordinates. The cell is represented by an n-dimensional cube with sides of length 2, and the coordinate origin located at the center of the cell. The geometry of the cell is for reference purposes only, and does not literally represent the actual shape of the cell. Think of this method instead as an easy way to specify a part (e.g. center, corner, face) of a higher dimensional cell which is extensible to any number of dimensions.

To illustrate this approach, consider a 2D cell. In 2 dimensions the cell is represented by a square. An xy axis is placed at its center, with the positive x-axis oriented East and the positive y-axis oriented North. The resulting coordinate for the lower left corner is at \((-1, -1)\), and upper right corner at \((1, 1)\). However, because our staggers are symmetric they don’t need to distinguish between the \(-1\), and the 1, so we only need concern ourselves with the first quadrant of this cell. We only need to use the 1, and the 0, and many of the cell locations collapse together (e.g. we only need to represent one corner). See figure 19 for an illustration of these concepts.

The cell center is represented by the coordinate pair \((0, 0)\) indicating the origin. The cell corner is \(+1\) in each direction, giving a coordinate pair of \((1, 1)\). The edges are each \(+1\) in one dimension and \(0\) in the other indicating that they’re even with the center in one dimension and offset in the other.

For three dimensions, the vertical component of the stagger location can be added by simply adding an additional coordinate. The three dimensional generalization of the cell center becomes \((0, 0, 0)\) and the cell corner becomes \((1, 1, 1)\). The rest of the 3D stagger locations are combinations of \(+1\) offsets from the center.

To generalize this to \(d\) dimensions, to represent a \(d\) dimensional stagger location. A set of \(d\) 0 and 1 is used to specify for each dimension whether a stagger location is aligned with the cell center in that dimension \((0)\), or offset by \(+1\) in
that dimension (1). Using this scheme we can represent any symmetric stagger location. To construct a custom stagger location in ESMF the subroutine `ESMF_StaggerLocSet()` is used to specify, for each dimension, whether the stagger is located at the interior (0) or on the boundary (1) of the cell. This method allows users to construct stagger locations for which there is no predefined value. In this example, it’s used to set the 4D center and 4D corner locations.

```fortran
! Set Center
call ESMF_StaggerLocSet(staggerLoc, loc=(/0,0,0,0/), rc=rc)
call ESMF_GridAddCoord(grid4D, staggerLoc=staggerLoc, rc=rc)

! Set Corner
call ESMF_StaggerLocSet(staggerLoc, loc=(/1,1,1,1/), rc=rc)
call ESMF_GridAddCoord(grid4D, staggerLoc=staggerLoc, rc=rc)
```

### 23.2.17 Specifying Custom Stagger Padding

There is an added complication with the data (e.g., coordinates) stored at stagger locations in that they can require different amounts of storage depending on the underlying Grid type.

Consider the example 2D grid in figure 20 where the dots represent the cell corners and the “+” represents the cell centers. For the corners to completely enclose the cell centers (symmetric stagger), the number of corners in each dimension needs to be one greater than the number of cell centers. In the above figure, there are two rows and three columns of cell centers. To enclose the cell centers, there must be three rows and four columns of cell corners. This is true in general for Grids without periodicity or other connections. In fact, for a symmetric stagger, given that the center location requires \( n \times m \) storage, the corresponding corner location requires \( n+1 \times m+1 \), and the edges, depending on the side, require \( n+1 \times m \) or \( m+1 \times n \). In order to add the extra storage, but also to allow the different stagger location arrays to remain on the same DistGrid, the capability of the ESMF Array class to have computational bounds different from exclusive bounds is used. By default, when the coordinate arrays are created, one extra layer of padding is added to the arrays to create symmetric staggerers (i.e., the center location is surrounded). The default is to add this padding on the positive side, and to only add this padding where needed (e.g., no padding for the center, padding on both dimensions for the corner, in only one dimension for the edge in 2D.) There are two ways for the user to change these defaults.

One way is to use the `GridEdgeWidth` or `GridAlign` arguments when creating a Grid. These arguments can be used to change the padding around the Grid cell index space. This extra padding is the extra space used by all the stagger locations, and no stagger location can extend outside of it.
The `gridEdgeLWidth` and `gridEdgeUWidth` arguments are both 1D arrays of the same size as the Grid dimension. The entries in the arrays give the extra offset from the outer boundary of the grid cell index space to the exclusive region of the Grid. The following example shows the creation of a Grid with all the extra space to hold stagger padding on the negative side of a Grid. This is the reverse of the default behavior. The resulting Grid will have an exclusive region which extends from (-1, -1) to (10, 10), however, the cell center stagger location will still extend from (1, 1) to (10, 10).

```fortran
grid2D=ESMF_GridCreateShapeTile(minIndex=(/1,1/),maxIndex=(/10,10/), &
    gridEdgeLWidth=(/1,1/), gridEdgeUWidth=(/0,0/), rc=rc)
```

To indicate how the data in a Grid’s stagger locations are aligned with the cell centers, the optional `gridAlign` parameter may be used. This parameter indicates which stagger elements in a cell share the same index values as the cell center. For example, in a 2D cell, it would indicate which of the four corners has the same index value as the center. To set `gridAlign`, the values -1,+1 are used to indicate the alignment in each dimension. This parameter is mostly informational, however, if the `gridEdgeWidth` parameters are not set then its value determines where the default padding is placed. If not specified, then the default is to align all staggerers to the most negative, so the padding is on the positive side. The following code illustrates creating a Grid aligned to the reverse of default (with everything to the positive side). This creates a Grid identical to that created in the previous example.

```fortran
grid2D=ESMF_GridCreateShapeTile(minIndex=(/1,1/),maxIndex=(/10,10/), &
    gridAlign=(/1,1/), rc=rc)
```

The `gridEdgeWidth` and `gridAlign` arguments both allow the user to set the extra padding available to be used by stagger locations in a Grid. By default, stagger locations allocated in a Grid set their stagger padding based on these values. A stagger location’s padding in each dimension is equal to the value of `gridEdgeWidth` (or the value implied by `gridAlign`), unless the stagger location is centered in a dimension in which case the stagger padding is 0. For example, the cell center stagger location has 0 stagger padding in all dimensions, whereas the edge stagger location lower padding is equal to `gridEdgeLWidth` and the upper padding is equal to `gridEdgeUWidth` in one dimension, but both are 0 in the other, centered, dimension. If the user wishes to set the stagger padding individually for each stagger location they may use the `staggerEdgeLWidth` and `staggerAlign` arguments, however, the padding set this way must be within that specified by the `gridEdgeWidth` and `gridAlign` used when the Grid was created (or the defaults if none were used).

The `staggerEdgeLWidth` and `staggerEdgeUWidth` arguments are both 1D arrays of the same size as the Grid dimension. The entries in the arrays give the extra offset from the Grid cell index space for a stagger location. The following example shows the addition of two stagger locations. The corner location has no extra boundary and the center has a single layer of extra padding on the negative side and none on the positive. This is the reverse of the default behavior.

```fortran
grid2D=ESMF_GridCreate(distgrid=distgrid2D, &
    gridEdgeLWidth=(/1,1/), gridEdgeUWidth=(/0,0/), rc=rc)

call ESMF_GridAddCoord(grid2D, &
    staggerLoc=ESMF_STAGGERLOC_CORNER, &
    staggerEdgeLWidth=(/0,0/), staggerEdgeUWidth=(/0,0/) , rc=rc)

call ESMF_GridAddCoord(grid2D, &
    staggerLoc=ESMF_STAGGERLOC_CENTER, &
    staggerEdgeLWidth=(/1,1/), staggerEdgeUWidth=(/0,0/) , rc=rc)
```

To indicate how the data at a particular stagger location is aligned with the cell center, the optional `staggerAlign` parameter may be used. This parameter indicates which stagger elements in a cell share the same index values as the
cell center. For example, in a 2D cell, it would indicate which of the four corners has the same index value as the center. To set `staggerAlign`, the values -1,+1 are used to indicate the alignment in each dimension. If a stagger location is centered in a dimension (e.g. an edge in 2D), then that dimension is ignored in the alignment. This parameter is mostly informational, however, if the `staggerEdgeWidth` parameters are not set then its value determines where the default padding is placed. If not specified, then the default is to align all staggers to the most negative, so the padding is on the positive side. The following code illustrates aligning the positive (northeast in 2D) corner with the center.

```fortran
call ESMF_GridAddCoord(grid2D, &
    staggerLoc=ESMF_STAGGERLOC_CORNER, staggerAlign=(/1,1/), rc=rc)
```

### 23.3 Restrictions and Future Work

- **7D limit.** Only grids up to 7D will be supported.
- **During the first development phase only single tile grids are supported.** In the near future, support for mosaic grids will be added. The initial implementation will be to create mosaics that contain tiles of the same grid type, e.g. rectilinear.
- **Future adaptation.** Currently Grids are created and then remain unchanged. In the future, it would be useful to provide support for the various forms of grid adaptation. This would allow the grids to dynamically change their resolution to more closely match what is needed at a particular time and position during a computation for front tracking or adaptive meshes.
- **Future Grid masks.** Grid masks will be implemented.
- **Future Exchange Grids.** The functionality for creating an exchange grid between two ordinary grids will be implemented to assist with the remapping of data during a regrid operation.
- **Future unstructured Grid.** Currently only grids which can be constructed from a set of logically rectangular tiles are supported. In the future more general unstructured grids will be implemented.
- **Future Grid IO.** In the future it would be useful to have a grid method which can read in a grid specification and distribution from a file and construct the grid. There may need to be a set of these methods corresponding to different group’s file formats.
- **Future Grid generation.** This class for now only contains the basic functionality for operating on the grid. In the future methods will be added to enable the automatic generation of various types of grids.

### 23.4 Design and Implementation Notes

#### 23.4.1 Grid Topology

The `ESMF_Grid` class depends upon the `ESMF_DistGrid` class for the specification of its topology. That is, when creating a Grid, first an `ESMF_DistGrid` is created to describe the appropriate index space topology. This decision was made because it seemed redundant to have a system for doing this in both classes. It also seems most appropriate for the machinery for topology creation to be located at the lowest level possible so that it can be used by other classes (e.g. the `ESMF_Array` class). Because of this, however, the authors recommend that as a natural part of the implementation of subroutines to generate standard grid shapes (e.g. `ESMF_GridGenSphere`) a set of standard topology generation subroutines be implemented (e.g. `ESMF_DistGridGenSphere`) for users who want to create a standard topology, but a custom geometry.

#### 23.4.2 Storage and Distribution of Stagger Locations in Grid

The primarily complication in the storage of multiple stagger locations in a Grid is that different variables in a symmetric stagger can require a different amount of storage depending on the underlying grid type. For example while `h,u, and v` on an `A-grid` all require `n x m` arrays, on a `B-grid` `u` and `v` require `n+1 x m+1`. On a `C` or `D` grid one vector
component requires n+1 x m and the other n x m+1. To handle this complication the natural approach would be to define each stagger’s storage to what is necessary for that grid type. This approach introduces a problem when the arrays are distributed, because they are different sizes. It is non-trivial to guarantee that the (i,j) element of all three of arrays ends up on the same processor. It is simpler to guarantee a consistent distribution of the arrays when using the same distGrid if they are the same size.

This may sound like a contradiction, but to be more precise we choose the exclusive region of each array to be the same size (say n x m), and pad the arrays that need it with additional memory from the computational region of the Array class. Recall that the exclusive region is defined as the cells for which the DE claims exclusive ownership. These are the cells updated by computations local to that DE. The exclusive region is a subset of the computation region.

The computational region contains all the cells kept locally on the DE in addition to the exclusive cells. By using the computational region as padding we are able to guarantee a consistent distribution of the arrays and at the same time impose a symmetric stagger. This approach extends naturally to the connected/periodic cases because the padding can be used to hold the values across the branch cut.

The biased configuration (where each stagger location has the same number of elements) falls out trivially by setting an optional padding argument (staggerWidth) to zero. This argument can also be used to adjust where the stagger padding is located or to add extra for halos.

23.5 Grid Options

23.5.1 ESMF_GridConn

DESCRIPTION:
The ESMF_GridCreateShapeTile command has three specific arguments connDim1, connDim2, and connDim3. These can be used to setup different types of connections at the ends of each dimension of a Tile. Each of these parameters is a two element array. The first element is the connection type at the minimum end of the dimension and the second is the connection type at the maximum end. The default value for all the connections is ESMF_GRIDCONN_NONE, specifying no connection.

ESMF_GRIDCONN_NONE No connection.

ESMF_GRIDCONN_PERIODIC Periodic connection.

ESMF_GRIDCONN_POLE This edge is connected to itself. Given that the edge is n elements long, then element i is connected to element i+n/2.

ESMF_GRIDCONN_BIPOLE This edge is connected to itself. Given that the edge is n elements long, element i is connected to element n-i-1.

23.5.2 ESMF_GridStatus

DESCRIPTION:
The ESMF Grid class can exist in three states. These states are present so that the library code can detect if a Grid has been appropriately setup for the task at hand. The following are the valid values of ESMF_GRIDSTATUS.

ESMF_GRIDSTATUS_NOT_READY: Status after a Grid has been created with ESMF_GridCreateEmpty. A Grid object container is allocated but space for internal objects is not. Topology information and coordinate information is incomplete. This object can be used in ESMF_GridSet() methods in which additional information is added to the Grid.

ESMF_GRIDSTATUS_SHAPE_READY: The Grid has a specific topology and distribution, but incomplete coordinate arrays. The Grid can be used as the basis for allocating a Field.

ESMF_GRIDSTATUS_REGRID_READY: The grid contains valid coordinate values and is ready to be used in regrid.
23.5.3 ESMF_StaggerLoc

DESCRIPTION:
In the ESMF Grid class, data can be located at different positions in a Grid cell. When setting or retrieving coordinate data the stagger location is specified to tell the Grid method from where in the cell to get the data. Although the user may define their own custom stagger locations, ESMF provides a set of predefined locations for ease of use. The following are the valid predefined stagger locations.

The 2D predefined stagger locations (illustrated in figure ??) are:

ESMF_STAGGERLOC_CENTER: The center of the cell.
ESMF_STAGGERLOC_CORNER: The corners of the cell.
ESMF_STAGGERLOC_EDGE1: The edges offset from the center in the 1st dimension.
ESMF_STAGGERLOC_EDGE2: The edges offset from the center in the 2nd dimension.

The 3D predefined stagger locations (illustrated in figure ??) are:

ESMF_STAGGERLOC_CENTER_VCENTER: The center of the 3D cell.
ESMF_STAGGERLOC_CORNER_VCENTER: Half way up the vertical edges of the cell.
ESMF_STAGGERLOC_EDGE1_VCENTER: The center of the face bounded by edge 1 and the vertical dimension.
ESMF_STAGGERLOC_EDGE2_VCENTER: The center of the face bounded by edge 2 and the vertical dimension.
**ESMF_STAGGERLOC_CORNER_VFACE**: The corners of the 3D cell.

**ESMF_STAGGERLOC_EDGE1_VFACE**: The center of the edges of the 3D cell parallel offset from the center in the 1st dimension.

**ESMF_STAGGERLOC_EDGE2_VFACE**: The center of the edges of the 3D cell parallel offset from the center in the 2nd dimension.

**ESMF_STAGGERLOC_CENTER_VFACE**: The center of the top and bottom face. The face bounded by the 1st and 2nd dimensions.

### 23.6 Class API: General Grid Methods

#### 23.6.1 ESMF_GridAddCoord - Allocate coordinate arrays but don’t set their values

**INTERFACE:**

```fortran
! Private name; call using ESMF_GridAddCoord()
subroutine ESMF_GridAddCoordNoValues(grid, staggerloc, &
staggerEdgeLWidth, staggerEdgeUWidth, staggerAlign, &
totalLWidth, totalUWidth, rc)
```

**ARGUMENTS:**

- `type(ESMF_Grid), intent(in) :: grid`
- `type (ESMF_StaggerLoc), intent(in),optional :: staggerloc`
- `integer, intent(in),optional :: staggerEdgeLWidth(:)`
- `integer, intent(in),optional :: staggerEdgeUWidth(:)`
- `integer, intent(in),optional :: staggerAlign`
- `integer, intent(in),optional :: totalLWidth`
- `integer, intent(in),optional :: totalUWidth`
- `integer, intent(in),optional :: rc`
integer, intent(in), optional :: staggerAlign(:) ! N. IMP
integer, intent(out), optional :: totalLWidth(:) ! N. IMP
integer, intent(out), optional :: totalUWidth(:) ! N. IMP
integer, intent(out),optional :: rc

DESCRIPTION:

When a Grid is created all of its potential stagger locations can hold coordinate data, but none of them have storage allocated. This call allocates coordinate storage (creates internal ESMF_Arrays and associated memory) for a particular stagger location. Note that this call doesn’t assign any values to the storage, it only allocates it. The remaining options staggerEdgeLWidth, etc. allow the user to adjust the padding on the coordinate arrays.

The arguments are:

grid  Grid to allocate coordinate storage in.

[staggerloc]  The stagger location to add. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

[staggerEdgeLWidth]  This array should be the same dimCount as the grid. It specifies the lower corner of the stagger region with respect to the lower corner of the exclusive region.

[staggerEdgeUWidth]  This array should be the same dimCount as the grid. It specifies the upper corner of the stagger region with respect to the upper corner of the exclusive region.

[staggerAlign]  This array is of size grid dimCount. For this stagger location, it specifies which element has the same index value as the center. For example, for a 2D cell with corner stagger it specifies which of the 4 corners has the same index as the center. If this is set and either staggerEdgeUWidth or staggerEdgeLWidth is not, this determines the default array padding for a stagger. If not set, then this defaults to all negative. (e.g. The most negative part of the stagger in a cell is aligned with the center and the padding is all on the positive side.)

[totalLWidth]  The lower boundary of the computational region in reference to the computational region. Note, the computational region includes the extra padding specified by ccordLWidth. [CURRENTLY NOT IMPLEMENTED]

[totalUWidth]  The lower boundary of the computational region in reference to the computational region. Note, the computational region includes the extra padding specified by staggerEdgeLWidth. [CURRENTLY NOT IMPLEMENTED]

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

23.6.2  ESMF_GridCreate - Create a Grid from a DistGrid

INTERFACE:

! Private name; call using ESMF_GridCreate()
function ESMF_GridCreateFromDistGrid(name,coordTypeKind,distgrid, &
distgridToGridMap, coordDimCount, coordDimMap, &
gridEdgeLWidth, gridEdgeUWidth, gridAlign, indexflag, &
destroyDistGrid, destroyDELayout, rc)

RETURN VALUE:

type(ESMF_Grid) :: ESMF_GridCreateFromDistGrid

ARGUMENTS:
DESCRIPTION:

This is the most general form of creation for an ESMF_Grid object. It allows the user to fully specify the topology and index space (of the distributed dimensions) using the DistGrid methods and then build a grid out of the resulting distgrid. The distgridToGridMap argument specifies how the Grid dimensions are mapped to the distgrid. The coordDimToGridMap and coordDimMap arguments allow the user to specify how the coordinate arrays should map to the grid dimensions. (Note, though, that creating a grid does not allocate coordinate storage. A method such as ESMF_GridAddCoord() must be called before adding coordinate values.)

The arguments are:

[name] ESMF_Grid name.
[coordTypeKind] The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.
distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the grid dimCount, otherwise a runtime ESMF error will be raised.
distgridToGridMap List that has as many elements as indicated by distgrid’s dimCount value. The elements map each dimension of distgrid to a dimension in the grid. (i.e. the values should range from 1 to griddimCount). If not specified, the default is to map all of distgrid’s dimensions against the lower dimensions of the grid in sequence.
coordDimCount List that has as many elements as the grid dimCount. Gives the dimension of each component (e.g. x) array. This is to allow factorization of the coordinate arrays. If not specified all arrays are the same size as the grid. [NOTE FACTORIZATION HAS NOT CURRENTLY BEEN IMPLEMENTED].
coordDimMap 2D list of size grid dimCount x grid dimCount. This array describes the map of each component array’s dimensions onto the grids dimensions. Each entry coordDimMap(i,j) tells which grid dimension component i’s, jth dimension maps to. Note that if j is bigger than coordDimCount(i) than its ignored.
gridEdgeLWidth The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.
gridEdgeUWidth The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.
gridAlign Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the EdgeWidths.
indexflag Indicates whether the indices in the grid are to be interpreted to form a flat pseudo global index space (ESMF_INDEX_GLOBAL), or are to be taken as patch local (ESMF_INDEX_DELOCAL), which is the default.
rc Return code; equals ESMF_SUCCESS if there are no errors.
23.6.3 ESMF_GridCreateEmpty - Create a Grid that has no contents

**INTERFACE:**

```fortran
function ESMF_GridCreateEmpty(rc)
  type(ESMF_Grid) :: ESMF_GridCreateEmpty
  integer, intent(out), optional :: rc
end function ESMF_GridCreateEmpty
```

**RETURN VALUE:**

```fortran
type(ESMF_Grid) :: ESMF_GridCreateEmpty
```

**ARGUMENTS:**

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

**DESCRIPTION:**

Partially create an ESMF_Grid object. This function allocates an ESMF_Grid object, but doesn’t allocate any coordinate storage or other internal structures. The ESMF_GridSet calls can be used to set the values in the grid object. Before using the grid, ESMF_GridCommit needs to be called to validate the internal state and construct internal structures.

The arguments are:

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

---

23.6.4 ESMF_GridCreateShapeTile - Create a Grid with an irregular distribution

**INTERFACE:**

```fortran
! Private name; call using ESMF_GridCreateShapeTile()
function ESMF_GridCreateShapeTileIrreg(name,coordTypeKind, minIndex, &
  countsPerDEDim1,countsPerDeDim2, countsPerDeDim3, &
  connDim1, connDim2, connDim3, &
  poleStaggerLoc1, poleStaggerLoc2, poleStaggerLoc3, &
  bipolePos1, bipolePos2, bipolePos3, &
  coordDep1, coordDep2, coordDep3, &
  gridEdgeLWidth, gridEdgeUWidth, gridAlign, &
  indexflag, petMap, rc)
  type(ESMF_Grid) :: ESMF_GridCreateShapeTileIrreg
end function ESMF_GridCreateShapeTileIrreg
```

**RETURN VALUE:**

```fortran
type(ESMF_Grid) :: ESMF_GridCreateShapeTileIrreg
```

**ARGUMENTS:**

```fortran
character (len=*), intent(in), optional :: name
  type(ESMF_TypeKind), intent(in), optional :: coordTypeKind
  integer, intent(in), optional :: minIndex(:)
  integer, intent(in) :: countsPerDEDim1(:)
  integer, intent(in) :: countsPerDeDim2(:)
  integer, intent(in) :: countsPerDeDim3(:)
  type(ESMF_GridConn), intent(in), optional :: connDim1(:) ! N. IMP.
  type(ESMF_GridConn), intent(in), optional :: connDim2(:) ! N. IMP.
  type(ESMF_GridConn), intent(in), optional :: connDim3(:) ! N. IMP.
  type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc1(2) ! N. IMP.
  type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc2(2) ! N. IMP.
  type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc3(2) ! N. IMP.
```

239
integer, intent(in), optional :: bipolePos1(2) ! N. IMP.
integer, intent(in), optional :: bipolePos2(2) ! N. IMP.
integer, intent(in), optional :: bipolePos3(2) ! N. IMP.
integer, intent(in), optional :: coordDep1(:)
integer, intent(in), optional :: coordDep2(:)
integer, intent(in), optional :: coordDep3(:)
integer, intent(in), optional :: gridEdgeLWidth(:)
integer, intent(in), optional :: gridEdgeUWidth(:)
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, intent(in), optional :: petMap(:,:,:)
integer, intent(out), optional :: rc

DESCRIPTION:

This method creates a single tile, irregularly distributed grid (see Figure 12). To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim<> arrays that are specified. Section 23.2.3 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates.

The arguments are:

[name] ESMF_Grid name.
[coordTypeKind] The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.
[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.
[countsPerDEDim1] This arrays specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).
[countsPerDEDim2] This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).
[countsPerDEDim3] This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.
[connDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]
[connDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]
[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]
[poleStaggerLoc1] Two element array describing the index dimension 1 connections. The first element represents
the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a
pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of
predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT
IMPLEMENTED]

[poleStaggerLoc2] Two element array describing the index dimension 2 connections. The first element represents
the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a
pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of
predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT
IMPLEMENTED]

[poleStaggerLoc3] Two element array describing the index dimension 3 connections. The first element represents
the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a
pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of
predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT
IMPLEMENTED]

[bipolePos1] Two element array describing the index dimension 1 connections. The first element represents the
minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a bipole,
this gives the index position of one of the poles. The other is half way around. If not present, the default is 1.
[CURRENTLY NOT IMPLEMENTED]

[bipolePos2] Two element array describing the index dimension 2 connections. The first element represents the
minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a bipole,
this gives the index position of one of the poles. The other is half way around. If not present, the default is 1.
[CURRENTLY NOT IMPLEMENTED]

[bipolePos3] Two element array describing the index dimension 3 connections. The first element represents the
minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a bipole,
this gives the index position of one of the poles. The other is half way around. If not present, the default is 1.
[CURRENTLY NOT IMPLEMENTED]

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions
described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the first
coordinate component array. The values specify which of the index dimensions the corresponding coordinate
arrays map to. If not present the default is /1,2,3/.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimen-
sions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the
second coordinate component array. The values specify which of the index dimensions the corresponding
coordinate arrays map to. If not present the default is /1,2,3/.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions
described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the third
coordinate component array. The values specify which of the index dimensions the corresponding coordinate
arrays map to. If not present the default is /1,2,3/.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space cor-
responding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra
padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space cor-
responding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra
padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden
by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the
EdgeWidths.
Flag that indicates how the DE-local indices are to be defined.

Sets the mapping of pets to the created DEs. This 3D should be of size (countsPerDEDim1) x (countsPerDEDim2) x (countsPerDEDim3). If countsPerDEDim3 isn’t present, then the last dimension is of size 1.

Return code; equals ESMF_SUCCESS if there are no errors.

23.6.5 ESMF_GridCreateShapeTile - Create a Grid with a regular distribution

INTERFACE:

    ! Private name; call using ESMF_GridCreateShapeTile()
    function ESMF_GridCreateShapeTileReg(name, coordTypeKind, &
        regDecomp, decompFlag, minIndex, maxIndex, &
        connDim1, connDim2, connDim3, &
        poleStaggerLoc1, poleStaggerLoc2, poleStaggerLoc3, &
        bipolePos1, bipolePos2, bipolePos3, &
        coordDep1, coordDep2, coordDep3, &
        gridEdgeLWidth, gridEdgeUWidth, gridAlign, &
        indexflag, petMap, rc)

RETURN VALUE:

    type(ESMF_Grid) :: ESMF_GridCreateShapeTileReg

ARGUMENTS:

    character (len=*), intent(in), optional :: name
    type(ESMF_TypeKind), intent(in), optional :: coordTypeKind
    integer, intent(in), optional :: regDecomp(:)
    type(ESMF_DecompFlag), intent(in), optional :: decompFlag(:)
    integer, intent(in), optional :: minIndex(:)
    integer, intent(in), optional :: maxIndex(:)
    type(ESMF_GridConn), intent(in), optional :: connDim1(:) ! N. IMP.
    type(ESMF_GridConn), intent(in), optional :: connDim2(:) ! N. IMP.
    type(ESMF_GridConn), intent(in), optional :: connDim3(:) ! N. IMP.
    type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc1(2) ! N. IMP.
    type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc2(2) ! N. IMP.
    type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc3(2) ! N. IMP.
    integer, intent(in), optional :: bipolePos1(2) ! N. IMP.
    integer, intent(in), optional :: bipolePos2(2) ! N. IMP.
    integer, intent(in), optional :: bipolePos3(2) ! N. IMP.
    integer, intent(in), optional :: coordDep1(:)
    integer, intent(in), optional :: coordDep2(:)
    integer, intent(in), optional :: coordDep3(:)
    integer, intent(in), optional :: gridEdgeLWidth(:)
    integer, intent(in), optional :: gridEdgeUWidth(:)
    integer, intent(in), optional :: gridAlign(:)
    type(ESMF_IndexFlag), intent(in), optional :: indexflag
    integer, intent(in), optional :: petMap(:,:,:)
DESCRIPTION:

This method creates a single tile, regularly distributed grid (see Figure 12). To specify the distribution, the user passes in an array (regDecomp) specifying the number of DEs to divide each dimension into. The array decompFlag indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.

The arguments are:

[name] ESMF_Grid name.

[coordTypeKind] The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.

[regDecomp] List that has the same number of elements as maxIndex. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the patch is to be divided between the DEs. The default setting is ESMF_DECOMP_HOMOGEN in all dimensions. Please see Section 9.1.6 for a full description of the possible options.

[minIndex] The bottom extent of the grid array. If not given then the value defaults to /1,1,1,.../.

[maxIndex] The upper extent of the grid array.

[connDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[connDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc1] Two element array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc2] Two element array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc3] Two element array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]
Two element array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1.

[CURRENTLY NOT IMPLEMENTED]

Two element array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1.

[CURRENTLY NOT IMPLEMENTED]

Two element array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1.

[CURRENTLY NOT IMPLEMENTED]

This array specifies the dependence of the first coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

This array specifies the dependence of the second coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

This array specifies the dependence of the third coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the EdgeWidths.

Flag that indicates how the DE-local indices are to be defined.

Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1.

Return code; equals ESMF_SUCCESS if there are no errors.

23.6.6 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 and related internal structures. This call does not destroy the internal coordinate Arrays, or the internally generated DistGrid.
The arguments are:
grid  ESMF_Grid to be destroyed.

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

23.6.7 ESMF_GridGet - Get information about a Grid

INTERFACE:
! Private name; call using ESMF_GridGet()
subroutine ESMF_GridGetDefault(grid, name, coordTypeKind, &
dimCount, tileCount, staggerlayoutsCount, localDECount, distgrid, &
distgridToGridMap, coordDimCount, coordDimMap, &
gridEdgeLWidth, gridEdgeUWidth, gridAlign, &
indexFlag, rc)

ARGUMENTS:
type(ESMF_Grid), intent(in) :: grid
class (len=*) , intent(out), optional :: name
type(ESMF_TypeKind), intent(out), optional :: coordTypeKind
integer, intent(out), optional :: dimCount
integer, intent(out), optional :: tileCount
integer, intent(out), optional :: staggerlayoutsCount
integer, intent(out), optional :: localDECount
type(ESMF_DistGrid), intent(out), optional :: distgrid
integer, intent(out), optional :: distgridToGridMap(:)
integer, intent(out), optional :: coordDimCount(:)
integer, intent(out), optional :: coordDimMap(:,:)
integer, intent(out), optional :: gridEdgeLWidth(:)
integer, intent(out), optional :: gridEdgeUWidth(:)
integer, intent(out), optional :: gridAlign(:)
type(ESMF_IndexFlag), intent(out), optional :: indexFlag
integer, intent(out), optional :: rc

DESCRIPTION:
Gets various types of information about a grid.
The arguments are:
grid  Grid to get the information from.

[name]  ESMF_Grid name.
[coordTypeKind]  The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.
[dimCount]  DimCount of the Grid object.
[tileCount]  The number of logically rectangular tiles in the grid.
[staggerlocCount] The number of stagger locations.

[localDECount] The number of DEs in this grid on this PET.

[distgrid] The structure describing the distribution of the grid.

[distgridToGridMap] List that has as many elements as the distgrid dimCount. This array describes mapping between the grids dimensions and the distgrid.

[coordDimCount] List that has as many elements as the grid dimCount (from arrayspec). Gives the dimension of each component (e.g. x) array. This is to allow factorization of the coordinate arrays. If not specified all arrays are the same size as the grid.

[coordDimMap] 2D list of size grid dimCount x grid dimCount. This array describes the map of each component array’s dimensions onto the grids dimensions.

[gridEdgeLWidth] The padding around the lower edges of the grid. The array should be of size greater or equal to the Grid dimCount.

[gridEdgeUWidth] The padding around the upper edges of the grid. The array should be of size greater or equal to the Grid dimCount.

[gridAlign] Specification of how the stagger locations should align with the cell index space. The array should be of size greater or equal to the Grid dimCount.

[indexflag] Flag that indicates how the DE-local indices are to be defined.

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

23.6.8 ESMF_GridGet - Get information about a particular DE in a stagger location in a Grid

INTERFACE:

! Private name; call using ESMF_GridGet()
subroutine ESMF_GridGetLocalDEPSloc(grid, localDe, staggerloc, &
eclusiveLBound, exclusiveUBound, exclusiveCount, &
computationalLBound, computationalUBound, computationalCount, rc)

ARGUMENTS:

type(ESMF_Grid), intent(in) :: grid
integer, intent(in) :: localDe
integer, intent(in) :: staggerloc
integer, intent(out), optional :: exclusiveLBound(:)
integer, intent(out), optional :: exclusiveUBound(:)
integer, intent(out), optional :: exclusiveCount(:)
ingteger, intent(out), optional :: computationalLBound(:)
ingteger, intent(out), optional :: computationalUBound(:)
ingteger, intent(out), optional :: computationalCount(:)
ingteger, intent(out), optional :: rc

DESCRIPTION:

This method gets information about the range of index space which a particular stagger location occupies. This call differs from the coordinate bound calls (e.g. ESMF_GridGetCoord) in that a given coordinate array may only occupy a subset of the Grid’s dimensions, and so these calls may not give all the bounds of the stagger location. The
bounds from this call are the full bounds, and so for example, give the appropriate bounds for allocating a F90 array
to hold data residing on the stagger location. Note that unlike the output from the Array, these values also include the
undistributed dimensions and are ordered to reflect the order of the indices in the Grid. This call will still give correct
values even if the stagger location does not contain coordinate arrays (e.g. if ESMF_GridAddCoord hasn’t yet been
called on the stagger location).

The arguments are:

grid Grid to get the information from.

[localDe] The local DE from which to get the information.

staggerloc The stagger location to get the information for. Please see Section 23.5.3 for a list of predefined stagger
locations.

exclusiveLBound Upon return this holds the lower bounds of the exclusive region. exclusiveLBound must be
allocated to be of size equal to the Grid dimCount. Please see Section 23.2.10 for a description of the regions
and their associated bounds and counts.

exclusiveUBound Upon return this holds the upper bounds of the exclusive region. exclusiveUBound must be
allocated to be of size equal to the Grid dimCount. Please see Section 23.2.10 for a description of the regions
and their associated bounds and counts.

exclusiveCount Upon return this holds the number of items in the exclusive region per dimension (i.e.
exclusiveUBound-exclusiveLBound+1)
exclusiveCount must be allocated to be of size equal to the Grid dimCount. Please see Section 23.2.10 for
a description of the regions and their associated bounds and counts.

computationalLBound Upon return this holds the lower bounds of the computational region. computationalLBound
must be allocated to be of size equal to the Grid dimCount. Please see Section 23.2.10 for a description of the
regions and their associated bounds and counts.

computationalUBound Upon return this holds the upper bounds of the computational region. computationalUBound
must be allocated to be of size equal to the Grid dimCount. Please see Section 23.2.10 for a description of the
regions and their associated bounds and counts.

computationalCount Upon return this holds the number of items in the computational region per dimension. (i.e.
computationalUBound-computationalLBound+1). computationalCount must be allocated to be of size equal to the Grid dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

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

23.6.9 ESMF_GridGet - Get information about a particular stagger location in a Grid

INTERFACE:

! Private name; call using ESMF_GridGet()
subroutine ESMF_GridGetPSloc(grid, staggerloc, &
    computationalEdgeLWidth, computationalEdgeUWidth, &
    minIndex, maxIndex, rc)

ARGUMENTS:

type(ESMF_Grid), intent(in) :: grid
type (ESMF_StaggerLoc), intent(in) :: staggerloc
integer, intent(out), optional :: computationalEdgeLWidth(:)
integer, intent(out), optional :: computationalEdgeUWidth(:)
DESCRIPTION:

This method gets information about a particular stagger location. This information is useful for creating an ESMF Array to hold the data at the stagger location.

The arguments are:

**grid** Grid to get the information from.

**staggerloc** The stagger location to get the information for. Please see Section 23.5.3 for a list of predefined stagger locations.

[computationalEdgeLWidth] Upon return this holds the global lower width of the stagger region. The width returned is only for the distGrid dimensions and is mapped to correspond to those dimensions. computationalEdgeLWidth must be allocated to be of size equal to the grid distDimCount (i.e., the grid’s distgrid’s dimCount).

[computationalEdgeUWidth] Upon return this holds the global upper width of the stagger region. The width returned is only for the distGrid dimensions and is mapped to correspond to those dimensions. computationalEdgeUWidth must be allocated to be of size equal to the grid distDimCount (i.e., the grid’s distgrid’s dimCount).

[minIndex] Upon return this holds the global lower index of this stagger location. minIndex must be allocated to be of size equal to the grid DimCount. Note that this value is only for the first Grid tile, as multigrid support is added, this interface will likely be changed or moved to adapt.

[maxIndex] Upon return this holds the global upper index of this stagger location. maxIndex must be allocated to be of size equal to the grid DimCount. Note that this value is only for the first Grid tile, as multigrid support is added, this interface will likely be changed or moved to adapt.

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

23.6.10 ESMF_GridGetCoord - Get Grid coordinate bounds and an F90 pointer to coordinate data

INTERFACE:

```fortran
subroutine ESMF_GridGetCoord(grid, localDE, coordDim, staggerloc, &
   exclusiveLBound, exclusiveUBound, exclusiveCount, &
   computationalLBound, computationalUBound, computationalCount, &
   totalLBound, totalUBound, totalCount, &
   <pointer argument>, doCopy, rc)
```

ARGUMENTS:

```fortran
type(ESMF_Grid), intent(in) :: grid
integer, intent(in) :: localDE
integer, intent(in) :: coordDim

type (ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(out), optional :: exclusiveLBound(:)
integer, intent(out), optional :: exclusiveUBound(:)
integer, intent(out), optional :: exclusiveCount(:)
integer, intent(out), optional :: computationalLBound(:)
integer, intent(out), optional :: computationalUBound(:)
integer, intent(out), optional :: computationalCount(:)
integer, intent(out), optional :: computationalCount(:)
```
DESCRIPTION:

This method gets a Fortran pointer to the piece of memory which holds the coordinate data on the local DE for the
given coordinate dimension and stagger locations. This is useful, for example, for setting the coordinate values in a
Grid, or for reading the coordinate values. Currently this method supports up to three coordinate dimensions, of either
R4 or R8 datatype. See below for specific supported values. If the coordinates that you are trying to retrieve are of
higher dimension, use the \texttt{ESMF\_GetCoord()} interface that returns coordinate values in an \texttt{ESMF\_Array} instead.
That interface supports the retrieval of coordinates up to 7D.

Supported values for the \texttt{<pointer argument>} are:

\begin{verbatim}
real(ESMF\_KIND\_R4), pointer :: fptr(:)
real(ESMF\_KIND\_R4), pointer :: fptr(:,:)
real(ESMF\_KIND\_R4), pointer :: fptr(:,:,:)
real(ESMF\_KIND\_R8), pointer :: fptr(:)
real(ESMF\_KIND\_R8), pointer :: fptr(:,:)
real(ESMF\_KIND\_R8), pointer :: fptr(:,:,:)
\end{verbatim}

The arguments are:

\begin{itemize}
\item \texttt{grid} Grid to get the information from.
\item \texttt{localDE} The local DE to get the information for (localDE starts at 0).
\item \texttt{coordDim} The coordinate dimension to get the data from (e.g. 1=x).
\item \texttt{staggerloc} The stagger location to get the information for. Please see Section \ref{section:staggerloc} for a list of predefined stagger
locations. If not present, defaults to \texttt{ESMF\_STAGGERLOC\_CENTER}.
\end{itemize}

\begin{itemize}
\item \texttt{[exclusiveLBound]} Upon return this holds the lower bounds of the exclusive region. \texttt{exclusiveLBound} must be
allocated to be of size equal to the coord dimCount.
\item \texttt{[exclusiveUBound]} Upon return this holds the upper bounds of the exclusive region. \texttt{exclusiveUBound} must be
allocated to be of size equal to the coord dimCount.
\item \texttt{[exclusiveCount]} Upon return this holds the number of items in the exclusive region per dimension (i.e. \texttt{exclusiveUBound−exclusiveLBound+1}).
\texttt{exclusiveCount} must be allocated to be of size equal to the coord dimCount. Please see Section \ref{section:exclusiveCount}
for a description of the regions and their associated bounds and counts.
\item \texttt{[computationalLBound]} Upon return this holds the lower bounds of the stagger region. \texttt{computationalLBound}
must be allocated to be of size equal to the coord dimCount. Please see Section \ref{section:computationalLBound}
for a description of the regions and their associated bounds and counts.
\item \texttt{[computationalUBound]} Upon return this holds the upper bounds of the stagger region. \texttt{exclusiveUBound} must be
allocated to be of size equal to the coord dimCount. Please see Section \ref{section:computationalUBound} for a description of the regions
and their associated bounds and counts.
\item \texttt{[computationalCount]} Upon return this holds the number of items in the computational region per dimension (i.e. \texttt{computationalUBound−computationalLBound+1}).
\texttt{computationalCount} must be allocated to be of size equal to the coord dimCount. Please see Section \ref{section:computationalCount}
for a description of the regions and their associated bounds and counts.
\end{itemize}
Upon return this holds the lower bounds of the total region. **totalLBound** must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

Upon return this holds the upper bounds of the total region. **totalUBound** must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

Upon return this holds the number of items in the total region per dimension (i.e. totalUBound - totalLBound + 1). **totalCount** must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

The pointer to the coordinate data.

If not specified, default to **ESMF_DATA_REF**, in this case fptr is a reference to the data in the Grid coordinate arrays. Please see Section 9.1.4 for further description and a list of valid values.

Return code; equals **ESMF_SUCCESS** if there are no errors.

---

### 23.6.11 ESMF_GridGetCoord - Get Grid coordinate bounds

**INTERFACE:**

```fortran
! Private name; call using ESMF_GridGetCoord()
subroutine ESMF_GridGetCoordBounds(grid, localDE, coordDim, staggerloc, &
    exclusiveLBound, exclusiveUBound, exclusiveCount, &
    computationalLBound, computationalUBound, computationalCount, &
    totalLBound, totalUBound, totalCount, rc)
```

**ARGUMENTS:**

- `type(ESMF_Grid), intent(in) :: grid`: grid
- `integer, intent(in) :: localDE`: localDE
- `integer, intent(in) :: coordDim`: coordDim
- `type (ESMF_StaggerLoc), intent(in), optional :: staggerloc`: staggerloc
- `integer, intent(out), optional :: exclusiveLBound(:)`: exclusiveLBound
- `integer, intent(out), optional :: exclusiveUBound(:)`: exclusiveUBound
- `integer, intent(out), optional :: exclusiveCount(:)`: exclusiveCount
- `integer, intent(out), optional :: computationalLBound(:)`: computationalLBound
- `integer, intent(out), optional :: computationalUBound(:)`: computationalUBound
- `integer, intent(out), optional :: computationalCount(:)`: computationalCount
- `integer, intent(out), optional :: totalLBound(:)`: totalLBound
- `integer, intent(out), optional :: totalUBound(:)`: totalUBound
- `integer, intent(out), optional :: totalCount(:)`: totalCount
- `integer, intent(out), optional :: rc`: rc

**DESCRIPTION:**

This method gets information about the range of index space which a particular piece of coordinate data occupies. In other words, this method returns the bounds of the coordinate arrays. Note that unlike the output from the Array, these values also include the undistributed dimensions and are ordered to reflect the order of the indices in the coordinate. So, for example, **totalLBound** and **totalUBound** should match the bounds of the Fortran array retrieved by ESMF_GridGetCoord.

The arguments are:
grid       Grid to get the information from.
localDE    The local DE to get the information for (localDE starts at 0).
coordDim   The coordinate dimension to get the information for (e.g. 1=x).
staggerloc The stagger location to get the information for. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. exclusiveLBound must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. exclusiveUBound must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

exclusiveCount] Upon return this holds the number of items in the exclusive region per dimension (i.e. exclusiveUBound-exclusiveLBound+1). exclusiveCount must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

computationalLBound] Upon return this holds the lower bounds of the computational region. computationalLBound must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

computationalUBound] Upon return this holds the upper bounds of the computational region. computationalUBound must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. computationalUBound-computationalLBound+1). computationalCount must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

totalLBound] Upon return this holds the lower bounds of the total region. totalLBound must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

totalUBound] Upon return this holds the upper bounds of the total region. totalUBound must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

totalCount] Upon return this holds the number of items in the total region per dimension (i.e. totalUBound-totalLBound+1). totalCount must be allocated to be of size equal to the coord dimCount. Please see Section 23.2.10 for a description of the regions and their associated bounds and counts.

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

23.6.12  ESMF_GridGetCoord - Get coordinates and put in an ESMF Array

INTERFACE:

    ! Private name; call using ESMF_GridGetCoord()
    subroutine ESMF_GridGetCoordIntoArray(grid, staggerloc, coordDim, array, &
        docopy, rc)

ARGUMENTS:
This method allows the user to get access to the ESMF Array holding coordinate data at a particular stagger location. This is useful, for example, to set the coordinate values. To have an Array to access, the coordinate Arrays must have already been allocated, for example by ESMF_GridAddCoord or ESMF_GridSetCoord. The arguments are:

- **staggerloc** The stagger location from which to get the arrays. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.
- **coordDim** The coordinate dimension to get the data from (e.g. 1=x).
- **array** An array into which to put the coordinate information.
- **[doCopy]** If not specified, default to ESMF_DATA_REF, in this case array will contain a reference to the Grid coordinate Arrays. Please see Section 9.1.4 for further description and a list of valid values. [THE ESMF_DATA_COPY OPTION IS CURRENTLY NOT IMPLEMENTED]
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.

---

### 23.6.13 ESMF_GridGetStatus - Return the status of the Grid

**INTERFACE:**

```fortran
function ESMF_GridGetStatus(grid)
    type(ESMF_GridStatus) :: ESMF_GridGetStatus
    type(ESMF_Grid) :: grid

RETURN VALUE:

- type(ESMF_GridStatus) :: ESMF_GridGetStatus

**ARGUMENTS:**

- type(ESMF_Grid) :: grid

**DESCRIPTION:**

Returns the status of the passed in Grid object.

The arguments are:

- **grid** The grid to return the status from.

---

### 23.6.14 ESMF_GridMatch - Check if two Grid objects match

**INTERFACE:**

```fortran
function ESMF_GridMatch(grid1, grid2, rc)
    type(ESMF_GridStatus) :: ESMF_GridMatch
    type(ESMF_Grid) :: grid1, grid2

RETURN VALUE:

- type(ESMF_GridStatus) :: ESMF_GridMatch

**ARGUMENTS:**

- type(ESMF_Grid) :: grid1, grid2

**DESCRIPTION:**

Returns the status of the passed in Grid object.

The arguments are:

- **grid1** The grid to return the status from.
RETURN VALUE:

logical :: ESMF_GridMatch

ARGUMENTS:

  type(ESMF_Grid), intent(in) :: grid1
  type(ESMF_Grid), intent(in) :: grid2
  integer, intent(out), optional :: rc

DESCRIPTION:

Check if grid1 and grid2 match. Returns .true. if Grid objects match, .false. otherwise. This method currently just checks if grid1 and grid2 are the same object, future work will do a more complex check.

The arguments are:

grid1 ESMF_Grid object.

grid2 ESMF_Grid object.

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

23.6.15  ESMF_GridSetCoord - Set coordinates using ESMF Arrays

INTERFACE:

 subroutine ESMF_GridSetCoordFromArray(grid, staggerloc, coordDim, &
    array, doCopy, rc)

ARGUMENTS:

  type(ESMF_Grid), intent(in) :: grid
  type (ESMF_StaggerLoc), intent(in), optional :: staggerloc
  integer, intent(in) :: coordDim
  type(ESMF_Array), intent(in) :: array
  type(ESMF_CopyFlag), intent(in), optional :: doCopy ! NOT IMPLEMENTED
  integer, intent(out), optional :: rc

DESCRIPTION:

This method sets the passed in Array as the holder of the coordinate data for stagger location staggerloc and coordinate coord. If the location already contains an Array, then this one overwrites it.

The arguments are:

staggerloc  The stagger location into which to copy the arrays. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

coordDim  The coordinate dimension to put the data in (e.g. 1=x).

array  An array to set the grid coordinate information from.

[doCopy] If not specified, default to ESMF_DATA_REF, in this case the Grid coordinate Array will be set to a reference to array. Please see Section 9.1.4 for further description and a list of valid values. [THE ESMF_DATA_COPY OPTION IS CURRENTLY NOT IMPLEMENTED]

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
23.6.16  ESMF_GridSetCommitShapeTile - Set and complete a Grid with an irregular distribution

INTERFACE:

! Private name; call using ESMF_GridSetCommitShapeTile()
subroutine ESMF_GridSetCmmitShapeTileIrreg(grid, name, coordTypeKind, minIndex, &
countsPerDEDim1, countsPerDeDim2, countsPerDEDim3, &
connDim1, connDim2, connDim3, &
poleStaggerLoc1, poleStaggerLoc2, poleStaggerLoc3, &
bipolePos1, bipolePos2, bipolePos3, &
coordDep1, coordDep2, coordDep3, &
gridEdgeLWidth, gridEdgeUWidth, gridAlign, &
indexflag, petMap, rc)

ARGUMENTS:

type (ESMF_Grid) :: grid
character (len=*) , intent(in), optional :: name

type(ESMF_TypeKind), intent(in), optional :: coordTypeKind
integer, intent(in), optional :: minIndex(:)
integer, intent(in) :: countsPerDEDim1(:)
integer, intent(in) :: countsPerDEDim2(:)
integer, intent(in) :: countsPerDEDim3(:)
integer, intent(in), optional :: countsPerDEDim3(:)
type(ESMF_GridConn), intent(in), optional :: connDim1(:) ! N. IMP.
type(ESMF_GridConn), intent(in), optional :: connDim2(:) ! N. IMP.
type(ESMF_GridConn), intent(in), optional :: connDim3(:) ! N. IMP.
type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc1(2) ! N. IMP.
type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc2(2) ! N. IMP.
type(ESMF_StaggerLoc), intent(in), optional :: poleStaggerLoc3(2) ! N. IMP.
integer, intent(in), optional :: bipolePos1(2) ! N. IMP.
integer, intent(in), optional :: bipolePos2(2) ! N. IMP.
integer, intent(in), optional :: bipolePos3(2) ! N. IMP.
integer, intent(in), optional :: coordDep1(:)
integer, intent(in), optional :: coordDep2(:)
integer, intent(in), optional :: coordDep3(:)
integer, intent(in) :: gridEdgeLWidth(:)
integer, intent(in) :: gridEdgeUWidth(:)
integer, intent(in), optional :: gridAlign(:)
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, intent(in), optional :: petMap(:, :, :)
integer, intent(out), optional :: rc

DESCRIPTION:

This method sets information into an empty Grid and then commits it to create a single tile, irregularly distributed grid (see Figure 12). To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim<> arrays that are specified. Section 23.2.3 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates. The arguments are:

grid  ESMF_Grid to set information into and then commit.
[name] ESMF_Grid name.

[coordTypeKind] The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,./.

countsPerDEDim1 This array specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).

countsPerDEDim2 This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).

countsPerDEDim3 This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.

[connDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[connDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc1] Two element array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a pole, this describes which stagger location is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc2] Two element array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a pole, this describes which stagger location is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc3] Two element array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a pole, this describes which stagger location is at the pole at each end. If not present, the default is the edge. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[bipolePos1] Two element array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a bipole, this gives the index position of one of the poles. The other is halfway around. If not present, the default is 1. [CURRENTLY NOT IMPLEMENTED]

[bipolePos2] Two element array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a bipole, this gives the index position of one of the poles. The other is halfway around. If not present, the default is 1. [CURRENTLY NOT IMPLEMENTED]
Two element array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1. [CURRENTLY NOT IMPLEMENTED]

This array specifies the dependence of the first coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

This array specifies the dependence of the second coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

This array specifies the dependence of the third coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the EdgeWidths.

Flag that indicates how the DE-local indices are to be defined.

Sets the mapping of pets to the created DEs. This 3D should be of size size(countsPerDEDim1) x size(countsPerDEDim2) x size(countsPerDEDim3). If countsPerDEDim3 isn’t present, then the last dimension is of size 1.

Return code; equals ESMF_SUCCESS if there are no errors.

### 23.6.17 ESMF_GridSetCommitShapeTile - Set and complete a Grid with a regular distribution

**INTERFACE:**

```fortran
subroutine ESMF_GridSetCommitShapeTileReg(grid, name, coordTypeKind, &
   regDecomp, decompFlag, minIndex, maxIndex, &
   connDim1, connDim2, connDim3, &
   poleStaggerLoc1, poleStaggerLoc2, poleStaggerLoc3, &
   bipolePos1, bipolePos2, bipolePos3, &
   coordDep1, coordDep2, coordDep3, &
   gridEdgeLWidth, gridEdgeUWidth, gridAlign, &
   indexflag, petMap, rc)
```

**ARGUMENTS:**

256
DESCRIPTION:

This method sets information into an empty Grid and then commits it to create a single tile, regularly distributed grid (see Figure 12). To specify the distribution, the user passes in an array \((\text{regDecomp})\) specifying the number of DEs to divide each dimension into. The array \((\text{decompFlag})\) indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.

The arguments are:

\(\text{grid}\) ESMF_Grid to set information into in preparation for commit.

\([\text{name}]\) ESMF_Grid name.

\([\text{coordTypeKind}]\) The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.

\([\text{regDecomp}]\) List that has the same number of elements as \((\text{maxIndex})\). Each entry is the number of decougs for that dimension. If not specified, the default decomposition will be \(\text{petCount} \times 1 \times 1 \times \ldots\).

\([\text{decompFlag}]\) List of decomposition flags indicating how each dimension of the patch is to be divided between the DEs. The default setting is \(\text{ESMF_DECOMP_HOMOGEN} \) in all dimensions. Please see Section 9.1.6 for a full description of the possible options.

\([\text{minIndex}]\) The bottom setting of the grid array. If not given then the value defaults to \(/1,1,1,\ldots/\).

\([\text{maxIndex}]\) The upper extent of the grid array.

\([\text{connDim1}]\) Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to \(\text{ESMF_GRIDCONN_NONE} \) \(\text{[CURRENTLY NOT IMPLEMENTED]}\)

\([\text{connDim2}]\) Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one
element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 23.5.1 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc1] Two element array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc2] Two element array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[poleStaggerLoc3] Two element array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a pole, this describes which staggerlocation is at the pole at each end. Please see Section 23.5.3 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER. [CURRENTLY NOT IMPLEMENTED]

[bipolePos1] Two element array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1. [CURRENTLY NOT IMPLEMENTED]

[bipolePos2] Two element array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1. [CURRENTLY NOT IMPLEMENTED]

[bipolePos3] Two element array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If a bipole, this gives the index position of one of the poles. The other is half way around. If not present, the default is 1. [CURRENTLY NOT IMPLEMENTED]

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is /1,2,3/.
The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the EdgeWidths.

Flag that indicates how the DE-local indices are to be defined.

Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1. If the Grid contains undistributed dimensions then these should also be of size 1.

Return code; equals ESMF_SUCCESS if there are no errors.

23.6.18 ESMF_GridValidate - Validate Grid internals

INTERFACE:

subroutine ESMF_GridValidate(grid, rc)

ARGUMENTS:

type(ESMF_Grid), intent(in) :: grid
integer, intent(out), optional :: rc

DESCRIPTION:

Validates that the Grid is internally consistent. Note that one of the checks that the Grid validate does is the Grid status. Currently, the validate will return an error if the grid is not at least ESMF_GRIDSTATUS_SHAPE_READY. This means if a Grid was created with ESMF_GridCreateEmpty it must also have been committed with ESMF_GridCommit to be valid. If a Grid was created with another create call it should automatically have the correct status level to pass the status part of the validate. The Grid validate at this time doesn’t check for the presence or consistency of the Grid coordinates. The method returns an error code if problems are found.

The arguments are:

grid Specified ESMF_Grid object.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

23.7 Class API: StaggerLoc Methods

23.7.1 ESMF_StaggerLocSet - Set a StaggerLoc to a particular position in the cell

INTERFACE:

! Private name; call using ESMF_StaggerLocSet()
subroutine ESMF_StaggerLocSetAllDim(staggerloc,loc,rc)
ARGUMENTS:

```fortran
  type (ESMF_StaggerLoc), intent(inout) :: staggerloc
  integer, intent(in) :: loc(:)
  integer, optional :: rc
```

DESCRIPTION:

Sets a custom staggerloc to a position in a cell by using the array loc. The values in the array should only be 0, 1. If loc(i) is 0 it means the position should be in the center in that dimension. If loc(i) is 1 then for dimension i, the position should be on the side of the cell. Please see Section 23.2.16 for diagrams and further discussion of custom stagger locations.

The arguments are:

- **staggerloc** Grid location to be initialized
- **loc** Array holding position data. Each entry in loc should only be 0 or 1. note that dimensions beyond those specified are set to 0.
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.

23.7.2 ESMF_StaggerLocSet - Set one dimension of a StaggerLoc to a particular position

INTERFACE:

```fortran
  ! Private name; call using ESMF_StaggerLocSet()
  subroutine ESMF_StaggerLocSetDim(staggerloc,dim,loc,rc)
  type (ESMF_StaggerLoc), intent(inout) :: staggerloc
  integer, intent(in) :: dim,loc
  integer, optional :: rc
```

ARGUMENTS:

```fortran
  type (ESMF_StaggerLoc), intent(inout) :: staggerloc
  integer, intent(in) :: dim,loc
  integer, optional :: rc
```

DESCRIPTION:

Sets a particular dimension of a custom staggerloc to a position in a cell by using the variable loc. The variable loc should only be 0, 1. If loc is 0 it means the position should be in the center in that dimension. If loc is +1 then for the dimension, the position should be on the positive side of the cell. Please see Section 23.2.16 for diagrams and further discussion of custom stagger locations.

The arguments are:

- **staggerloc** Stagger location to be initialized
- **dim** Dimension to be changed (1-7).
- **loc** Position data should be either 0,1.
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.
23.7.3  ESMF_StaggerLocString - Return a StaggerLoc as a string

INTERFACE:

    subroutine ESMF_StaggerLocString(staggerloc, string, rc)

ARGUMENTS:

    type(ESMF_StaggerLoc), intent(in) :: staggerloc
    character (len = *), intent(out) :: string
    integer, intent(out), optional :: rc

DESCRIPTION:

    Return an ESMF_StaggerLoc as a printable string.

    The arguments are:
    staggerloc  The ESMF_StaggerLoc to be turned into a string.
    string  Return string.
    [rc]  Return code; equals ESMF_SUCCESS if there are no errors.

23.7.4  ESMF_StaggerLocPrint - Print information of a ESMF_StaggerLoc object

INTERFACE:

    subroutine ESMF_StaggerLocPrint(staggerloc, rc)

ARGUMENTS:

    type (ESMF_StaggerLoc), intent(in) :: staggerloc
    integer, intent(out), optional :: rc

DESCRIPTION:

    Print the internal data members of a ESMF_StaggerLoc object
    Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a
type with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order.
If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed
in a future release.

    The arguments are:
    staggerloc  ESMF_StaggerLoc object as the method input
    [rc]  Return code; equals ESMF_SUCCESS if there are no errors.

24  DistGrid Class

24.1  Description

    The ESMF_DistGrid class sits on top of the DELayout class and holds domain information in index space. A
DistGrid object captures the index space topology and describes its decomposition in terms of DEs. Combined with
DELLayout and VM the DistGrid defines the data distribution of a domain decomposition across the computational
resources of an ESMF component.
The global domain is defined as the union or “patchwork” of logically rectangular (LR) sub-domains or patches. The DistGrid create methods allow the specification of such a patchwork global domain and its decomposition into exclusive, DE-local LR regions according to various degrees of user specified constraints. Complex index space topologies can be constructed by specifying connection relationships between patches during creation.

The DistGrid class holds domain information for all DEs. Each DE is associated with a local LR region. No overlap of the regions is allowed. The DistGrid offers query methods that allow DE-local topology information to be extracted, e.g. for the construction of halos by higher classes.

A DistGrid object only contains decomposable dimensions. The minimum rank for a DistGrid object is 1. A maximum rank does not exist for DistGrid objects, however, ranks greater than 7 may lead to difficulties with respect to the Fortran API of higher classes based on DistGrid. The rank of a DELayout object contained within a DistGrid object must be equal to the DistGrid rank. Higher class objects that use the DistGrid, such as an Array object, may be of different rank than the associated DistGrid object. The higher class object will hold the mapping information between its dimensions and the DistGrid dimensions.

24.2 Use and Examples

The following examples demonstrate how to create, use and destroy DistGrid objects. In order to produce complete and valid DistGrid objects all of the ESMF_DistGridCreate() calls require to be called in unison i.e. on all PETs of a component with a complete set of valid arguments.

24.2.1 Single patch DistGrid with regular decomposition

The minimum information required to create an ESMF_DistGrid object for a single patch with default decomposition are the corners of the patch in index space. The following call will create a 1D DistGrid for a 1D index space patch with elements from 1 through 1000.

\[
\text{distgrid} = \text{ESMF_DistGridCreate}(\text{minIndex}=(/1/), \text{maxIndex}=(/1000/), \text{rc}=\text{rc})
\]

A default DELayout with 1 DE per PET will be created during ESMF_DistGridCreate(). The 1000 elements of the specified 1D patch will then be block decomposed across the available DEs, i.e. across all PETs. Hence, for 4 PETs the (min) ~ (max) corners of the DE-local LR regions will be:

- DE 0 - (1) ~ (250)
- DE 1 - (251) ~ (500)
- DE 2 - (501) ~ (750)
- DE 3 - (751) ~ (1000)

DistGrids with rank > 1 can also be created with default decompositions, specifying only the corners of the patch. The following will create a 2D DistGrid for a 5x5 patch with default decomposition.

\[
\text{distgrid} = \text{ESMF_DistGridCreate}(\text{minIndex}=(/1,1/), \text{maxIndex}=(/5,5/), \text{rc}=\text{rc})
\]

The default decomposition for a DistGrid of rank $N$ will be $(nDEs \times 1 \times \ldots \times 1)$, where $nDEs$ is the number of DEs in the DELayout and there are $N-1$ factors of 1. For the 2D example above this means a $4 \times 1$ regular decomposition if executed on 4 PETs and will result in the following DE-local LR regions:

- DE 0 - (1,1) ~ (2,5)
- DE 1 - (3,1) ~ (3,5)
- DE 2 - (4,1) ~ (4,5)
- DE 3 - (5,1) ~ (5,5)

In many cases the default decomposition will not suffice for higher rank DistGrids (rank > 1). For this reason a decomposition descriptor regDecomp argument is available during ESMF_DistGridCreate(). The following call creates a DistGrid on the same 2D patch as before, but now with a user specified regular decomposition of $2 \times 3 = 6$ DEs.
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
regDecomp=(/2,3/), rc=rc)

The default DE labeling sequence follows column major order for the regDecomp argument:

```
0 2 4
1 3 5
```

1st dimension

2nd dimension

By default grid points along all dimensions are homogeneously divided between the DEs. The maximum element count difference between DEs along any dimension is 1. The (min) ~ (max) corners of the DE-local LR domains of the above example are as follows:

DE 0 - (1,1) ~ (3,2)
DE 1 - (4,1) ~ (5,2)
DE 2 - (1,3) ~ (3,4)
DE 3 - (4,3) ~ (5,4)
DE 4 - (1,5) ~ (3,5)
DE 5 - (4,5) ~ (5,5)

The specifics of the patch decomposition into DE-local LR domains can be modified by the optional decompflag argument. The following line shows how this argument is used to keep ESMF's default decomposition in the first dimension but move extra grid points of the second dimension to the last DEs in that direction. Extra elements occur if the number of DEs for a certain dimension does not evenly divide its extent. In this example there are 2 extra grid points for the second dimension because its extent is 5 but there are 3 DEs along this index space axis.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
regDecomp=(/2,3/), decompflag=(/ESMF_DECOMP_DEFAULT, ESMF_DECOMP_RESTLAST/),&
rc=rc)
```

Now DE 4 and DE 5 will hold the extra elements along the 2nd dimension.

DE 0 - (1,1) ~ (3,1)
DE 1 - (4,1) ~ (5,1)
DE 2 - (1,2) ~ (3,2)
DE 3 - (4,2) ~ (5,2)
DE 4 - (1,3) ~ (3,5)
DE 5 - (4,3) ~ (5,5)

An alternative way of indicating the DE-local LR regions is to list the index space coordinate as given by the associated DistGrid patch for each dimension. For this 2D example there are two lists (dim 1) / (dim 2) for each DE:

```
DE 0 - (1,2,3) / (1)
DE 1 - (4,5) / (1)
DE 2 - (1,2,3) / (2)
DE 3 - (4,5) / (2)
DE 4 - (1,2,3) / (3,4,5)
DE 5 - (4,5) / (3,4,5)
```

Information about DE-local LR regions in the latter format can be obtained from the DistGrid object by use of ESMF_DistGridGet() methods:
allocate(dimExtent(2, 0:5)) ! (dimCount, deCount)
call ESMF_DistGridGet(distgrid, delayout=delayout, &
    indexCountPDimPDe=dimExtent, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
call ESMF_DELayoutGet(delayout, localDeCount=localDeCount, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
allocate(localDeList(0:localDeCount-1))
call ESMF_DELayoutGet(delayout, localDeList=localDeList, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
do localDe=0, localDeCount-1
    de = localDeList(localDe)
do dim=1, 2
    allocate(localIndexList(dimExtent(dim, de))) ! allocate list to hold indices
    call ESMF_DistGridGet(distgrid, localDe=localDe, dim=dim, &
        indexList=localIndexList, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(terminationflag=ESMF_ABORT)
        dim, ":: ", localIndexList
    deallocate(localIndexList)
endo
dendo
deallocate(localDeList)
deallocate(dimExtent)

The advantage of the localIndexList format over the min-/max-corner format is that it can be used directly for DE-local to patch index dereferencing. Furthermore the localIndexList allows to express very general decompositions such as the cyclic decompositions in the first dimension generated by the following call:

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), decompflag=(/ESMF_DECOMP_CYCLIC,ESMF_DECOMP_RESTLAST/), &
    rc=rc)

with decomposition:

DE 0 - (1,3,5) / (1)
DE 1 - (2,4) / (1)
DE 2 - (1,3,5) / (2)
DE 3 - (2,4) / (2)
DE 4 - (1,3,5) / (3,4,5)
DE 5 - (2,4) / (3,4,5)

Finally, a DistGrid object is destroyed by calling

call ESMF_DistGridDestroy(distgrid, rc=rc)

24.2.2 DistGrid and DELayout

The examples of this section use the 2D DistGrid of the previous section to show the interplay between DistGrid and DELayout. By default, i.e. without specifying the delayout argument, a DELayout will be created during DistGrid creation that provides as many DEs as the DistGrid object requires. The implicit call to ESMF_DELayoutCreate() is issued with a fixed number of DEs and default settings in all other aspects. The resulting DE to PET mapping depends on the number of PETs of the current VM context. Assuming 6 PETs in the VM
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), rc=rc)

will result in the following domain decomposition in terms of DEs

```
0 2 4
1 3 5
```

and their layout or distribution over the available PETs:

```
DE 0 -> PET 0
DE 1 -> PET 1
DE 2 -> PET 2
DE 3 -> PET 3
DE 4 -> PET 4
DE 5 -> PET 5
```

Running the same example on a 4 PET VM will not change the domain decomposition into 6 DEs as specified by

```
0 2 4
1 3 5
```

but the layout across PETs will now contain multiple DE-to-PET mapping with default cyclic distribution:

```
DE 0 -> PET 0
DE 1 -> PET 1
DE 2 -> PET 2
DE 3 -> PET 3
DE 4 -> PET 0
DE 5 -> PET 1
```

Sometimes it may be desirable for performance tuning to construct a DELayout with specific characteristics. For instance, if the 6 PETs of the above example are running on 3 nodes of a dual-SMP node cluster and there is a higher communication load along the first dimension of the model than along the second dimension it would be sensible to place DEs according to this knowledge.

The following example first creates a DELayout with 6 DEs where groups of 2 DEs are to be in fast connection. This DELayout is then used to create a DistGrid.

```
delayout = ESMF_DELayoutCreate(deCount=6, deGrouping=(/(i/2,i=0,5)/), rc=rc)
```

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), delayout=delayout, rc=rc)
```

This will ensure a distribution of DEs across the cluster resource in the following way:

```
0 2 4
1 3 5
SMP SMP SMP
```

The interplay between DistGrid and DELayout may at first seem complicated. The simple but important rule to understand is that DistGrid describes a domain decomposition and each domain is labeled with a DE number. The DELayout describes how these DEs are laid out over the compute resources of the VM, i.e. PETs. The DEs are purely logical elements of decomposition and may be relabeled to fit the algorithm or legacy code better. The following example demonstrates this by describing the exact same distribution of the domain data across the fictitious cluster of SMP-nodes with a different choice of DE labeling:
delayout = ESMF_DELayoutCreate(deCount=6, deGrouping=(/(mod(i,3),i=0,5)/), &
    rc=rc)

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), deLabelList=(/0,3,1,4,2,5/), delayout=delayout, rc=rc)

Here the deLabelList argument changes the default DE label sequence from column major to row major. The DELayout compensates for this change in DE labeling by changing the deGrouping argument to map the first dimension to SMP nodes as before. The decomposition and layout now looks as follows:

```
 0 1 2
3 4 5
SMP SMP SMP
```

Finally, in order to achieve a completely user-defined distribution of the domain data across the PETs of the VM a DELayout may be created from a petMap before using it in the creation of a DistGrid. If for instance the desired distribution of a 2 x 3 decomposition puts the DEs of the first row onto 3 separate PETs (PET 0, 1, 2) and groups the DEs of the second row onto PET 3 a petMap must first be setup that takes the DE labeling of the DistGrid into account. The following lines of code result in the desired distribution using column major DE labeling by first create a DELayout and then using it in the DistGrid creation.

delayout = ESMF_DELayoutCreate(petMap=(/0,3,1,3,2,3/), rc=rc)

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), delayout=delayout, rc=rc)

This decomposes the global domain into

```
 0 2 4
1 3 5
```

and associates the DEs to the following PETs:

- DE 0 -> PET 0
- DE 1 -> PET 3
- DE 2 -> PET 1
- DE 3 -> PET 3
- DE 4 -> PET 2
- DE 5 -> PET 3

### 24.2.3 Single patch DistGrid with decomposition by DE blocks

The examples of the previous sections showed how DistGrid objects with regular decompositions are created. However, in some cases a regular decomposition may not be specific enough. The following example shows how the deBlockList argument is used to create a DistGrid object with completely user-defined decomposition. A single 5x5 LR domain is to be decomposed into 6 DEs. To this end a list is constructed that holds the min and max corners of all six DE LR blocks. The DE-local LR blocks are arranged as to cover the whole patch domain without overlap.

```latex
allocate(deBlockList(2, 2, 6)) ! (dimCount, 2, deCount)
deBlockList(:,1,1) = (/1,1/) ! minIndex 1st deBlock
```
deBlockList(:,2,1) = (/3,2/) ! maxIndex  1st deBlock
deBlockList(:,1,2) = (/4,1/) ! minIndex  2nd deBlock
deBlockList(:,2,2) = (/5,2/) ! maxIndex  2nd deBlock
deBlockList(:,1,3) = (/1,3/) 
deBlockList(:,2,3) = (/2,4/) 
deBlockList(:,1,4) = (/3,3/) 
deBlockList(:,2,4) = (/5,4/) 
deBlockList(:,1,5) = (/1,5/) 
deBlockList(:,2,5) = (/3,5/) 
deBlockList(:,1,6) = (/4,5/) ! minIndex  6th deBlock
deBlockList(:,2,6) = (/5,5/) ! maxIndex  6th deBlock

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
deBlockList=deBlockList, rc=rc)

24.2.4 Single patch DistGrid with periodic boundaries

By default the edges of all patches have solid wall boundary conditions. Periodic boundary conditions can be imposed by specifying connections between patches. For the single LR domain of the last section periodic boundaries along the first dimension are imposed by adding a connectionList argument with only one element to the create call. Each connectionList element is a vector of (3 * dimCount + 2) integer numbers:

allocate(connectionList(3*2+2, 1)) ! (3*dimCount+2, number of connections)

and has the following format:
(/patchIndex_A, patchIndex_B, positionVector, orientationVector, repetitionVector/).
The following constructor call can be used to construct a suitable connectionList element.

call ESMF_DistGridConnection(connection=connectionList(:,1), &
patchIndexA=1, patchIndexB=1, &
positionVector=(/5, 0/), &
orientationVector=(/1, 2/), &
repetitionVector=(/1, 0/), rc=rc)

The patchIndexA and patchIndexB arguments specify that this is a connection within patch 1. The positionVector indicates that there is no offset between patchB and patchA along the second dimension, but there is an offset of 5 along the first dimension (which in this case is the length of dimension 1). This aligns patchB (which is patch 1) right next to patchA (which is also patch 1).
The orientationVector fixes the orientation of the patchB index space to be the same as the orientation of patchA (it maps index 1 of patchA to index 1 of patchB and the same for index 2). The orientationVector could have been omitted in this case which corresponds to the default orientation.
Finally, the repetitionVector indicates that this connection element will be periodically repeated along dimension 1.
The connectionList can now be used to create a DistGrid object with the desired boundary conditions.

distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
deBlockList=deBlockList, connectionList=connectionList, rc=rc)

deallocate(connectionList)

This closes the patch along the first dimension on itself, thus imposing periodic boundaries along this direction.
### 24.2.5 2D patchwork DistGrid with regular decomposition

Creating a DistGrid from a list of LR domains is a straightforward extension of the case with a single LR domain. The first four arguments of `ESMF_DistGridCreate()` are promoted to rank 2, the second dimension being the patch count index.

The following 2D patchwork domain consisting of 3 LR patches will be used in the examples of this section:

```
----------------------------------------> 2nd dim
|                  (1,11)-----(1,20)
|                  |       |
|                  |       |
|                  (10,11)---(10,20) (11,11)---(11,20)
|                  |       |
|                (11,1)----(11,10)  |
|                  |       |
|                  |       |
|                  |       |
|                  (20,1)----(20,10)  (20,11)---(20,20) v
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
1st dim
```

The first step in creating a patchwork global domain is to construct the `minIndex` and `maxIndex` arrays.

```
allocate(minIndex(2,3)) ! (dimCount, number of patches)
allocation(maxIndex(2,3)) ! (dimCount, number of patches)
minIndex(:,1) = (/11,1/) maxIndex(:,1) = (/20,10/)
minIndex(:,2) = (/11,11/) maxIndex(:,2) = (/20,20/)
minIndex(:,3) = (/1,11/) maxIndex(:,3) = (/10,20/)
```

Next the regular decomposition for each patch is set up in the `regDecomp` array. In this example each patch is associated with a single DE.

```
allocate(regDecomp(2,3)) ! (dimCount, number of patches)
regDecomp(:,1) = (/1,1/) ! one DE
regDecomp(:,2) = (/1,1/) ! one DE
regDecomp(:,3) = (/1,1/) ! one DE
```

Finally the DistGrid can be created by calling

```
distgrid = ESMF_DistGridCreate(minIndex=minIndex, maxIndex=maxIndex, &
regDecomp=regDecomp, rc=rc)
```

The default DE labeling sequence is identical to the patch labeling sequence and follows the sequence in which the patches are defined during the create call. However, DE labels start at 0 whereas patch labels start at 1. In this case the DE labels look as:
Each patch can be decomposed differently into DEs. The default DE labeling follows the column major order for each patch. This is demonstrated in the following case where the patchwork global domain is decomposed into 9 DEs,

\[
\text{regDecomp(:,1)} = (/2,2/) \quad \text{! 4 DEs} \\
\text{regDecomp(:,2)} = (/1,3/) \quad \text{! 3 DEs} \\
\text{regDecomp(:,3)} = (/2,1/) \quad \text{! 2 DEs}
\]

\[
distgrid = \text{ESMF_DistGridCreate}(	ext{minIndex}=\text{minIndex}, \text{maxIndex}=\text{maxIndex}, \& \text{regDecomp}=\text{regDecomp}, \text{rc}=\text{rc})
\]

resulting in the following decomposition:

```
+-------+     +-------+
| 7 |     | 4 5 6 |
|     | 8     |       |
+-------+     +-------+
| 0 2 |     | 1 3   |
+-------+     +-------+
```

DE 0 - (11,1) ~ (15,5) 
DE 1 - (16,1) ~ (20,5) 
DE 2 - (11,6) ~ (15,10) 
DE 3 - (16,6) ~ (20,10) 
DE 4 - (11,11) ~ (20,14) 
DE 5 - (11,15) ~ (20,17) 
DE 6 - (11,18) ~ (20,20) 
DE 7 - (1,11) ~ (5,20) 
DE 8 - (6,11) ~ (10,20)

The \text{decompflag} and \text{deLabelList} arguments can be used much like in the single LR domain case to overwrite the default grid decomposition (per patch) and to change the overall DE labeling sequence, respectively.

### 24.3 Restrictions and Future Work
- Topologies that require Connections and ConnectionTransforms are not supported yet.
- DEs that participate in cyclic decompositions are excluded from GetDeLinks queries.

### 24.4 Design and Implementation Notes
This section will be updated as the implementation of the DistGrid class nears completion.

### 24.5 Class API
#### 24.5.1 ESMF_DistGridCreate - Create DistGrid object with regular decomposition

**INTERFACE:**
ARGUMENTS:

integer, intent(in) :: minIndex(:)
integer, intent(in) :: maxIndex(:)
integer, target, intent(in), optional :: regDecomp(:)
type(ESMF_DecompFlag), target,intent(in), optional :: decompFlag(:)
integer, target, intent(in), optional :: regDecompFirstExtra(:)
integer, target, intent(in), optional :: regDecompLastExtra(:)
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, target, intent(in), optional :: connectionList(:,:)
integer, target, intent(in), optional :: connectionTransList(:,:)
type(ESMF_DELayout), intent(in), optional :: delayout
integer, target, intent(in), optional :: deLabelList(:)
type(ESMF_VM), intent(in), optional :: vm
integer, intent(out),optional :: rc

RETURN VALUE:

  type(ESMF_DistGrid) :: ESMF_DistGridCreateRD

DESCRIPTION:

Create an ESMF_DistGrid from a single logically rectangular (LR) patch with regular decomposition. A regular decomposition is of the same rank as the patch and decomposes each dimension into a fixed number of DEs. A regular decomposition of a single patch is expressed by a single regDecomp list of DE counts in each dimension.

The arguments are:

minIndex  Global coordinate tuple of the lower corner of the patch.
maxIndex  Global coordinate tuple of the upper corner of the patch.
regDecomp List of DE counts for each dimension. The default decomposition will be $deCount \times 1 \times ... \times 1$. The value of $deCount$ for a default DELayout equals petCount, i.e. the default decomposition will be into as many DEs as there are PETs and the distribution will be 1 DE per PET.
decomFlag List of decomposition flags indicating how each dimension of the patch is to be divided between the DEs. The default setting is ESMF_DECOMP_HOMOGEN in all dimensions. See section 9.1.6 for a list of valid decomposition flag options.
regDecompFirstExtra Extra elements on the first DEs along each dimension in a regular decomposition. The default is a zero vector.
regDecompLastExtra Extra elements on the last DEs along each dimension in a regular decomposition. The default is a zero vector.
deLabelList List assigning DE labels to the default sequence of DEs. The default sequence is given by the column major order of the regDecomp argument.
indexFlag Indicates whether the indices provided by the minIndex and maxIndex arguments are to be interpreted to form a flat pseudo global index space (ESMF_INDEX_GLOBAL) or are to be taken as patch local (ESMF_INDEX_DELOCAL), which is the default.
**[connectionList]** List of connections between patches in index space. The second dimension of `connectionList` steps through the connection interface elements, defined by the first index. The first index must be of size $2 \times \text{dimCount} + 2$, where `dimCount` is the rank of the decomposed index space. Each `connectionList` element specifies the connection interface in the format

\[(/\text{patchIndex}_A, \text{patchIndex}_B, \text{positionVector}, \text{orientationVector}/)\]

where:

- `patchIndex_A` and `patchIndex_B` are the patch index of the two connected patches respectively,
- `positionVector` is the vector that points from patch A’s minIndex to patch B’s minIndex.
- `orientationVector` associates each dimension of patch A with a dimension in patch B’s index space. Negative index values may be used to indicate a reversal in index orientation.

**[connectionTransList]** List of transforms associated with patch connections defined in `connectionList`. The second dimension of `connectionTransList` steps through the connection transforms, defined by the first index. The first index must be of size $5 + \text{dimCount}$, where `dimCount` is the rank of the decomposed index space. Each `connectionTransList` element specifies a connection transform by a list of integer values in the format \((/\text{connectionIndex}, \text{direction}, \text{staggerSrc}, \text{staggerDst}, \text{offsetDst}, \text{signVector}/)\), where

- `connectionIndex` corresponds to the index of the connection in `connectionList`,
- `direction` can be +1 to specify forward direction, i.e. source patch of the transform is patch A and destination patch is patch B of the corresponding connection, or -1 to indicate reverse direction through the connection. The only other valid `direction` value is 0 which indicates a bidirectional connection with source and destination definitions as in the forward case.
- `staggerSrc` and `staggerDst` indicate staggering location in the source and destination patch interface, respectively,
- `offsetDst` is a vector of size `dimCount` that specifies the index offset on the destination side of this connection,
- `signVector` is of size `dimCount` with elements either +1 or -1 to indicate optional sign change of vector components along the respective directions.

**[delayout]** Optional `ESMF_DELayout` object to be used. By default a new DELayout object will be created with the correct number of DEs. If a DELayout object is specified its number of DEs must match the number indicated by `regDecomp`.

**[vm]** Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method’s overhead.

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

### 24.5.2 ESMF_DistGridCreate - Create DistGrid object with DE blocks

**INTERFACE:**

```fortran
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateDB(minIndex, maxIndex, deBlockList, &
   deLabelList, indexflag, connectionList, connectionTransList, delayout, &
   vm, rc)
ARGUMENTS:
```
integer, intent(in) :: minIndex(:)
integer, intent(in) :: maxIndex(:)
integer, intent(in) :: deBlockList(:,:,:)
integer, intent(in), optional :: deLabelList(:)
type(ESMF_IndexFlag), intent(in), optional :: indexflag
integer, intent(in), optional :: connectionList(:,:)
type(ESMF_DELayout), intent(in), optional :: delayout
type(ESMF_VM), intent(in), optional :: vm
integer, intent(out),optional :: rc

RETURN VALUE:

type(ESMF_DistGrid) :: ESMF_DistGridCreateDB

DESCRIPTION:

Create an ESMF_DistGrid from a single logically rectangular (LR) patch with decomposition specified by deBlockList. The arguments are:

minIndex  Global coordinate tuple of the lower corner of the patch.
maxIndex  Global coordinate tuple of the upper corner of the patch.
deBlockList List of DE-local LR blocks. The third index of deBlockList steps through the deBlock elements, which are defined by the first two indices. The first index must be of size dimCount and the second index must be of size 2. Each 2D element of deBlockList defined by the first two indices hold the following information.

<table>
<thead>
<tr>
<th>1st index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 minIndex(1)</td>
</tr>
<tr>
<td>2 minIndex(2)</td>
</tr>
<tr>
<td>. minIndex(.)</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>2nd index</td>
</tr>
</tbody>
</table>

It is required that there be no overlap between the LR segments defined by deBlockList.

deLabelList  List assigning DE labels to the default sequence of DEs. The default sequence is given by the column major order of the regDecomp argument.

indexflag  Indicates whether the indices provided by the minIndex and maxIndex arguments are to be interpreted to form a flat pseudo global index space (ESMF_INDEX_GLOBAL) or are to be taken as patch local (ESMF_INDEX_DELOCAL), which is the default.

connectionList List of connections between patches in index space. The second dimension of connectionList steps through the connection interface elements, defined by the first index. The first index must be of size 2 x dimCount + 2, where dimCount is the rank of the decomposed index space. Each connectionList element specifies the connection interface in the format

(/patchIndex_A, patchIndex_B, positionVector, orientationVector/) where:

- patchIndex_A and patchIndex_B are the patch index of the two connected patches respectively,
- positionVector is the vector that points from patch A’s minIndex to patch B’s minIndex.
• orientationVector associates each dimension of patch A with a dimension in patch B’s index space. Negative index values may be used to indicate a reversal in index orientation.

[connectionTransList] List of transforms associated with patch connections defined in connectionList. The second dimension of connectionTransList steps through the connection transforms, defined by the first index. The first index must be of size $5 + \text{dimCount}$, where \text{dimCount} is the rank of the decomposed index space. Each connectionTransList element specifies a connection transform by a list of integer values in the format $(/\text{connectionIndex}, \text{direction}, \text{staggerSrc}, \text{staggerDst}, \text{offsetDst}, \text{signVector}/)$, where

- connectionIndex corresponds to the index of the connection in connectionList.
- direction can be $+1$ to specify forward direction, i.e. source patch of the transform is patch A and destination patch is patch B of the corresponding connection, or $-1$ to indicate reverse direction through the connection. The only other valid direction value is $0$ which indicates a bidirectional connection with source and destination definitions as in the forward case.
- staggerSrc and staggerDst indicate staggering location in the source and destination patch interface, respectively.
- offsetDst is a vector of size \text{dimCount} that specifies the index offset on the destination side of this connection.
- signVector is of size \text{dimCount} with elements either $+1$ or $-1$ to indicate optional sign change of vector components along the respective directions.

[delayout] Optional ESMF_DELayout object to be used. By default a new DELayout object will be created with the correct number of DEs. If a DELayout object is specified its number of DEs must match the number indicated by regDecomp.

[vm] Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method’s overhead.

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

### 24.5.3 ESMF_DistGridCreate - Create DistGrid object from patchwork with regular decomposition

**INTERFACE:**

```fortran
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateRDP(minIndex, maxIndex, regDecomp,&
    decompflag, regDecompFirstExtra, regDecompLastExtra, deLabelList, &
    indexflag, connectionList, connectionTransList, delayoutput, vm, rc)
ARGUMENTS:

    integer, intent(in) :: minIndex(:,:)
    integer, intent(in) :: maxIndex(:,:)
    integer, intent(in), optional :: regDecomp(:,:)
    type(ESMF_DecompFlag),target, intent(in), optional :: decompflag(:,:)
    integer, target, intent(in), optional :: regDecompFirstExtra(:,:)
    integer, target, intent(in), optional :: regDecompLastExtra(:,:)
    integer, intent(in), optional :: deLabelList(:)
    type(ESMF_IndexFlag), intent(in), optional :: indexflag
    integer, intent(in), optional :: connectionList(:,:)
    integer, intent(in), optional :: connectionTransList(:,:)
    type(ESMF_DELayout), intent(in), optional :: delayoutput
    type(ESMF_VM), intent(in), optional :: vm
    integer, intent(out),optional :: rc
```

273
**RETURN VALUE:**

\[ \text{type(ESMF\_DistGrid) :: ESMF\_DistGridCreateRDP} \]

**DESCRIPTION:**

Create an ESMF\_DistGrid from a patchwork of logically rectangular (LR) patches with regular decomposition. A regular decomposition is of the same rank as the patch and decomposes each dimension into a fixed number of DEs. A regular decomposition of a patchwork of patches is expressed by a list of DE count vectors, one vector for each patch. Each vector contained in the \text{regDecomp} argument ascribes DE counts for each dimension. It is erroneous to provide more patches than there are DEs.

The arguments are:

- **minIndex** The first index provides the global coordinate tuple of the lower corner of a patch. The second index indicates the patch number.

- **maxIndex** The first index provides the global coordinate tuple of the upper corner of a patch. The second index indicates the patch number.

- **[regDecomp]** List of DE counts for each dimension. The second index indicates the patch number. The default decomposition will be \text{deCount} \times 1 \times \ldots \times 1. The value of deCount for a default DELayout equals petCount, i.e. the default decomposition will be into as many DEs as there are PETs and the distribution will be 1 DE per PET.

- **[decompflag]** List of decomposition flags indicating how each dimension of each patch is to be divided between the DEs. The default setting is \text{ESMF\_DECOMP\_HOMOGEN} in all dimensions for all patches. See section 9.1.6 for a list of valid decomposition flag options. The second index indicates the patch number.

- **[regDecompFirstExtra]** Extra elements on the first DEs along each dimension in a regular decomposition. The default is a zero vector. The second index indicates the patch number.

- **[regDecompLastExtra]** Extra elements on the last DEs along each dimension in a regular decomposition. The default is a zero vector. The second index indicates the patch number.

- **[deLabelList]** List assigning DE labels to the default sequence of DEs. The default sequence is given by the column major order of the \text{regDecomp} elements in the sequence as they appear following the patch index.

- **[indexflag]** Indicates whether the indices provided by the minIndex and maxIndex arguments are to be interpreted to form a flat pseudo global index space (ESMF\_INDEX\_GLOBAL) or are to be taken as patch local (ESMF\_INDEX\_DELOCAL), which is the default.

- **[connectionList]** List of connections between patches in index space. The second dimension of connectionList steps through the connection interface elements, defined by the first index. The first index must be of size 2 \times dimCount + 2, where dimCount is the rank of the decomposed index space. Each connectionList element specifies the connection interface in the format

  \[ (/\text{patchIndex\_A, patchIndex\_B, positionVector, orientationVector/}) \]

  - **patchIndex\_A** and **patchIndex\_B** are the patch index of the two connected patches respectively,
  - **positionVector** is the vector that points from patch A's minIndex to patch B's minIndex.
  - **orientationVector** associates each dimension of patch A with a dimension in patch B's index space. Negative index values may be used to indicate a reversal in index orientation.

- **[connectionTransList]** List of transforms associated with patch connections defined in connectionList. The second dimension of connectionTransList steps through the connection transforms, defined by the first index. The first index must be of size 5 + dimCount, where dimCount is the rank of the decomposed index space. Each connectionTransList element specifies a connection transform by a list of integer values in the format

  \[ (/\text{connectionIndex, direction, staggerSrc, staggerDst, offsetDst, signVector/}) \]

  where
- `connectionIndex` refers to the index of the connection in `connectionList`.
- `direction` can be `+1` to specify forward direction, i.e., source patch of the transform is `patch_A` and destination patch is `patch_B` of the corresponding connection, or `-1` to indicate reverse direction through the connection. The only other valid `direction` value is `0` which indicates a bidirectional connection with source and destination definitions as in the forward case.
- `staggerSrc` and `staggerDst` indicate staggering location in the source and destination patch interface, respectively.
- `offsetDst` is a vector of size `dimCount` that specifies the index offset on the destination side of this connection.
- `signVector` is of size `dimCount` with elements either `+1` or `-1` to indicate optional sign change of vector components along the respective directions.

**[delayout]** Optional `ESMF_DELlayout` object to be used. By default a new DELlayout object will be created with the correct number of DEs. If a DELlayout object is specified its number of DEs must match the number indicated by `regDecomp`.

**[vm]** Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

### 24.5.4 ESMF_DistGridCreate - Create 1D DistGrid object from user's arbitrary index list

**INTERFACE:**

```fortran
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateDBAI(arbSeqIndexList, rc)

ARGUMENTS:

integer, intent(in) :: arbSeqIndexList(:)
integer, intent(out),optional :: rc

RETURN VALUE:

  type(ESMF_DistGrid) :: ESMF_DistGridCreateDBAI

DESCRIPTION:

Create an `ESMF_DistGrid` of `dimCount` 1 from a PET-local list of sequence indices. The PET-local size of the `arbSeqIndexList` argument determines the number of local elements in the created DistGrid. The sequence indices must be unique across all PETs and are meant to be used in combination with `ESMF_ArraySparseMatMulStore()`. A default 1-D DELlayout with 1 DE per PET across all PETs of the current VM is automatically created. The arguments are:

**[arbSeqIndexList]** List of arbitrary sequence indices that reside on the local PET.

**[rc]** Return code; equals `ESMF_SUCCESS` if there are no errors.
24.5.5 ESMF_DistGridDestroy - Destroy DistGrid object

INTERFACE:

    subroutine ESMF_DistGridDestroy(distgrid, rc)

ARGUMENTS:

    type(ESMF_DistGrid), intent(inout) :: distgrid
    integer, intent(out), optional :: rc

DESCRIPTION:

Destroy an ESMF_DistGrid object.
The arguments are:

distgrid  ESMF_DistGrid object to be destroyed.

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

24.5.6 ESMF_DistGridGet - Get information about DistGrid object

INTERFACE:

    subroutine ESMF_DistGridGet(distgrid, delayout, dimCount, patchCount, &
                                minIndexPDimPPatch, maxIndexPDimPPatch, elementCountPPatch, &
                                minIndexPDimPDe, maxIndexPDimPDe, elementCountPDe, patchListPDe, &
                                indexCountPDimPDe, regDecompFlag, rc)

ARGUMENTS:

    type(ESMF_DistGrid), intent(in) :: distgrid
    type(ESMF_DELayout), intent(out), optional :: delayout
    integer, intent(out), optional :: dimCount
    integer, intent(out), optional :: patchCount
    integer, intent(out), optional :: minIndexPDimPPatch(:,:)
    integer, intent(out), optional :: maxIndexPDimPPatch(:, :)
    integer, intent(out), optional :: elementCountPPatch(:)
    integer, intent(out), optional :: minIndexPDimPDe(:, :)
    integer, intent(out), optional :: maxIndexPDimPDe(:, :)
    integer, intent(out), optional :: elementCountPDe(:)
    integer, intent(out), optional :: patchListPDe(:)
    integer, intent(out), optional :: indexCountPDimPDe(:, :)
    type(ESMF_Logical), intent(out), optional :: regDecompFlag
    integer, intent(out), optional :: rc

DESCRIPTION:

Get internal DistGrid information.
The arguments are:

distgrid  Queried ESMF_DistGrid object.

delayout  ESMF_DELayout object associated with distgrid.
24.5.7 ESMF_DistGridGet - Get DE local information about DistGrid

INTERFACE:

! Private name; call using ESMF_DistGridGet()
subroutine ESMF_DistGridGetPLocalDe(distgrid, localDe, seqIndexList, rc)

ARGUMENTS:

type(ESMF_DistGrid), intent(in) :: distgrid
integer, intent(in) :: localDe
integer, intent(out), optional :: seqIndexList(:)
integer, intent(out), optional :: rc

DESCRIPTION:

Get internal DistGrid information.
The arguments are:

**distgrid**  Queried ESMF_DistGrid object.

**localDe**  Local DE for which information is requested. [0,..,localDeCount-1]

**seqIndexList**  List of DistGrid patch-local sequence indices for localDe, with size(seqIndexList) == (/elementCountPDe(localDe)/).

**rc**  Return code; equals ESMF_SUCCESS if there are no errors.
24.5.8 ESMF_DistGridGet - Get DE local information for dimension about DistGrid

**INTERFACE:**

```fortran
! Private name; call using ESMF_DistGridGet()
subroutine ESMF_DistGridGetPLocalDePDim(distgrid, localDe, dim, indexList, rc)
```

**ARGUMENTS:**

- `type(ESMF_DistGrid), intent(in) :: distgrid` : distgrid
- `integer, intent(in) :: localDe` : localDe
- `integer, intent(in) :: dim` : dim
- `integer, intent(out), optional :: indexList(:)` : indexList
- `integer, intent(out), optional :: rc` : rc

**DESCRIPTION:**

Get internal DistGrid information.
The arguments are:

- **distgrid**  Queried ESMF_DistGrid object.
- **localDe**  Local DE for which information is requested. [0,..,localDeCount-1]
- **dim**  Dimension for which information is requested. [1,..,dimCount]
- **indexList**  Upon return this holds the list of DistGrid patch-local indices for localDe along dimension dim. The supplied variable must be at least of size indexCountPDimPDe(dim, de(localDe)).
- **rc**  Return code; equals ESMF_SUCCESS if there are no errors.

24.5.9 ESMF_DistGridPrint - Print DistGrid internals

**INTERFACE:**

```fortran
subroutine ESMF_DistGridPrint(distgrid, options, rc)
```

**ARGUMENTS:**

- `type(ESMF_DistGrid), intent(in) :: distgrid` : distgrid
- `character(len=*) , intent(in), optional :: options` : options
- `integer, intent(out), optional :: rc` : rc

**DESCRIPTION:**

Prints internal information about the specified ESMF_DistGrid object to stdout.
Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.
The arguments are:

- **distgrid**  Specified ESMF_DistGrid object.
- **options**  Print options are not yet supported.
- **rc**  Return code; equals ESMF_SUCCESS if there are no errors.
24.5.10  ESMF_DistGridMatch - Check if two DistGrid objects match

INTERFACE:

    function ESMF_DistGridMatch(distgrid1, distgrid2, rc)

RETURN VALUE:

    type(ESMF_Logical) :: ESMF_DistGridMatch

ARGUMENTS:

    type(ESMF_DistGrid), intent(in) :: distgrid1
    type(ESMF_DistGrid), intent(in) :: distgrid2
    integer, intent(out), optional :: rc

DESCRIPTION:

    Check if distgrid1 and distgrid2 match. Returns ESMF_TRUE if DistGrid objects match, ESMF_FALSE otherwise.
    The arguments are:

distgrid1  ESMF_DistGrid object.

distgrid2  ESMF_DistGrid object.

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

24.5.11  ESMF_DistGridValidate - Validate DistGrid internals

INTERFACE:

    subroutine ESMF_DistGridValidate(distgrid, rc)

ARGUMENTS:

    type(ESMF_DistGrid), intent(in) :: distgrid
    integer, intent(out), optional :: rc

DESCRIPTION:

    Validates that the distgrid is internally consistent. The method returns an error code if problems are found.
    The arguments are:

distgrid  Specified ESMF_DistGrid object.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
24.5.12 ESMF_DistGridConnection - Construct a DistGrid connection element

INTERFACE:

    subroutine ESMF_DistGridConnection(connection, patchIndexA, patchIndexB, &
    positionVector, orientationVector, repetitionVector, rc)

ARGUMENTS:

    integer, intent(out) :: connection(:)
    integer, intent(in) :: patchIndexA
    integer, intent(in) :: patchIndexB
    integer, intent(in) :: positionVector(:)
    integer, intent(in), optional :: orientationVector(:)
    integer, intent(in), optional :: repetitionVector(:)
    integer, intent(out), optional :: rc

DESCRIPTION:

This call helps to construct a DistGrid connection, which is a simple vector of integers, out of its components.

The arguments are:

connection Element to be constructed. The provided connection must be dimensioned to hold exactly the number of integers that result from the input information.

patchIndexA Index of one of the two patches that are to be connected.

patchIndexB Index of one of the two patches that are to be connected.

positionVector Position of patch B’s minIndex with respect to patch A’s minIndex.

[orientationVector] Associates each dimension of patch A with a dimension in patch B’s index space. Negative index values may be used to indicate a reversal in index orientation. It is erroneous to associate multiple dimensions of patch A with the same index in patch B. By default orientationVector = (/1,2,3,.../), i.e. same orientation as patch A.

[repetitionVector] The allowed values for each direction are 0 and 1. An entry of 1 indicates that this connection element will be repeated along the respective dimension. A value of 0 indicates no repetition along this dimension. By default repetitionVector = (/0,0,0,.../), i.e. no repetition along any direction.

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

25 IOSpec Class

25.1 Description

The IOSpec is 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.

25.2 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 FieldBundle. 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.

25.3 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.

25.4 Class API

25.4.1 ESMF_IOSpecGet - Get values in an IOSpec

**INTERFACE:**

```fortran
subroutine ESMF_IOSpecGet(iospec, filename, iofileformat, &
                          iorwtype, asyncIO, rc)
```

**PARAMETERS:**

```fortran
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:

25.4.2 ESMF_IOSpecSet - Set values in an IOSpec

**INTERFACE:**

```fortran
subroutine ESMF_IOSpecSet(iospec, filename, iofileformat, &
                          iorwtype, asyncIO, rc)
```

**PARAMETERS:**

```fortran
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:
26 Overview of Distributed Data Methods

FieldBundles, Fields, and Arrays all have versions of the following data communication methods. In these objects, 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. The ESMF goal of providing performance portability means the framework will in the future attempt to select the fastest communication strategy on each hardware platform transparently to the user code. (The current implementation uses MPI for communication.)

Communication patterns, meaning exactly which bytes need to be copied or sent from one PET to another to perform the requested operation, can be precomputed during an initialization phase and then later executed repeatedly. There is a common object handle, an ESMF_RouteHandle, which identifies these stored communication patterns. Only the ESMF_RouteHandle and the source and destination data pointers must be supplied at runtime to minimize execution overhead.

26.1 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** Update ghost-cell or halo regions at the boundaries of a local data decomposition.
- **Regrid** Transform data from one Grid to another, performing any necessary data interpolation.
- **Redist** Copy data associated with a single Grid from one decomposition to another. No data interpolation is necessary.

26.2 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.

26.3 Common Options

ESMF will select an appropriate default for the internal communication strategy for executing the communications. However, additional control is available to the user by specifying the following route options. (For more details on exactly what changes with the various options, see Section 26.4)
26.3.1 ESMF_RouteOptions

DESCRIPTION:
Specifies control options when executing the communication represented by a Route object. Normally these do not need to be set by the user, but can be specified if the best communication strategy is known in advance. The synchronous and asynchronous options are mutually exclusive; and the other packing options are also mutually exclusive. Setting the Route options is "sticky"; it maintains the last value set until explicitly changed.

Note that these options control the internal execution of a single set of communications represented by a Route object and do not affect the user level behavior at all. For example, the asynchronous option does not cause the user level entry point to return sooner; it means the route will queue all communication requests first and then go back and check for completion in an internal loop.

Valid values are:

ESMF_ROUTE_OPTION_ASYNC Use an internal asynchronous strategy to execute the Route.

ESMF_ROUTE_OPTION_SYNC Use an internal synchronous strategy to execute the Route.

ESMF_ROUTE_OPTION_PACK_PET Pack all data from or to another PET into a single buffer when sending or receiving.

ESMF_ROUTE_OPTION_PACK_XP Pack all data from each non-contiguous exchange packet into a single buffer when sending or receiving.

ESMF_ROUTE_OPTION_PACK_NOPACK Do no buffering; send each contiguous run of data as a distinct communications operation.

ESMF_ROUTE_OPTION_PACK_VECTOR Use the MPI type vector interfaces to send non-contiguous data which has regular strides when sending or receiving.

ESMF_ROUTE_OPTION_PACK_BUFFER When multiple data addresses are sent to the Route routines (for example, identical ESMF_Fields from an ESMF_FieldBundle), this flag controls whether to pack the buffers together or send them separately.

ESMF_ROUTE_OPTION_DEFAULT Use the system default for communication, which is the combination of ESMF_ROUTE_OPTION_PACK_BUFFER, ESMF_ROUTE_OPTION_PACK_PET, and ESMF_ROUTE_OPTION_SYNC.

26.4 Design and Implementation Notes

1. There is an internal ESMC_Route class which supports the distributed communication methods. There are 4 additional internal-only classes which support ESMC_Route: ESMC_AxisIndex, ESMC_XPacket, ESMC_CommTable, and ESMC_RTable; and a public ESMF_RouteHandle class which is what the user sets and gets. The implementation is in C++, with interfaces in Fortran 90.

The general communication strategy is that each DE computes its own communication information independently, in parallel, and adds entries to a per-PET route table which contains all needed sends and receives (or gets and puts) stored in terms relative to itself. (Implementation note: this code will need to be made thread-safe if multiple threads are trying to add information to the same route table.)

AxisIndex is a small helper class which contains an index minimum and maximum for each dimension and is used to describe an n-dimensional hypercube of information in index space. These are associated with logically rectangular grids and local data arrays. There are usually multiple instances of them, for example the local data chunk, and the overall global index-space grid this data is a subset of. Within each of the local or global categories, there are also multiple instances to describe the allocated space, the total area, the computational area, and the exclusive area. See Figure ?? for the definitions of each of these regions. (Implementation note: the allocated space is only partially implemented internally and has no external user API yet.)

An Exchange Packet (XPacket) describes groups of memory addresses which constitute an n-dimensional hypercube of data. Each XPacket has an offset from a base address, a contiguous run length, a stride (or number of items to skip) per dimension, and a repeat count per dimension. See Figure ?? for a diagram of how the XPacket describes memory. The actual unit size stored in an XPacket is an item count, so before using an XPacket to
address bytes of memory the item size must be known and the counts multiplied by the number of bytes per item. This allows the same XPacket to describe different data types which have the same memory layout, for example 4 byte integers and 8 byte reals/doubles. The XPacket methods include basic set/get, how to turn a list of AxisIndex objects into an XPacket, compute a local XPacket from one in global (undecomposed grid) space, and a method to compute the intersection of 2 XPackets and produce a 3rd XPacket describing that region.

The Communication Table (CommTable) class encapsulates which other PETs this PET needs to talk to, and in what order. There are create and destroy methods, methods to set that a PET has data either to send or receive, and query routines that return an answer to the question ‘which PET should I exchange data with next’.

The Route Table (RTable) class contains a list of XPackets to be sent and received from other PETs. It has create/destroy methods, methods to add XPackets to the list for each PET, and methods to retrieve the XPackets from any list.

The top level class is a Route. A Route object contains a send RTable, a recv RTable, a CommTable, and a pointer to a Virtual Machine. The VM must include all PETs which are participating in this communication. The Route methods include create/destroy, setting a send or recv XPacket for a particular PET, and some higher level functions specific to each type of communication, for example RoutePrecomputeHalo or RoutePrecomputeRedist. These latter functions are where the XPackets are actually computed and added to the Route table. Each DE computes its own set of intersections, either source or destination, and fills its own corresponding PET.
The Route methods also include a RouteRun method which executes the code which actually traverses the table and sends the information between PETs.

A RouteHandle class is a small helper class which is returned through the public API to the user when a Route is created, and passed back in through the API to select which precomputed Route is to be executed. A RouteHandle contains a handle type and a pointer to a Route object. In addition, for use only by the Regrid code, there is an additional Route pointer and a TransformValues pointer. (TransformValues is an internal class only used by the Regridding code.) If the RouteHandle describes the Route for a FieldBundle, then the RouteHandle can contain a list of Routes, one for each Field in the FieldBundle, and for Regrid use, a list of additional Routes instead of a single Route. There is also a flag to indicate whether a single Route is applicable to all Fields in a FieldBundle or whether there are multiple Routes. The RouteHandle methods are fairly basic; mostly accessor methods for getting and setting values.

2. While intended for any distributed data communication method, the current implementation only builds a Route object for the halo, redist, and regrid methods. Scatter, Gather, AllGather, and AlltoAll should have the option of building a Route for operations which are executed repeatedly. This should only require writing a Precompute method for each one; the existing RouteRun can be invoked for these operations. (This is a lack-of-implementation-time issue, not a design or architecture issue.)

3. The original design included automatic detection of different Routes and internal caching, so the user API did not have to include a RouteHandle object to identify which Route was being invoked. However, users requested that the framework not cache and that explicit RouteHandle arguments be created and required to invoke the distributed data methods. Nothing prevents this code from being revived from the CVS repository and reinstated in the system, should automatic caching be desired by future users.

4. The current distributed methods have 2 related but distinct interfaces which differ in what information they require and whether they use RouteHandles:

Precompute/Run/Release  This is the most frequently used interface set. It contains 3 distinct phases: precomputing which bytes must be moved, actually executing the communications operation, and releasing the stored information. This is intended for any communication pattern which will be executed more than once.

All-in-One   For a communication which will only be executed once, or in any situation in which the user does not want to save a RouteHandle, there are interfaces which do not have RouteHandles as part of the argument list. Internally the code computes a Route, executes it, and releases the resources before returning.

5. The current CommTable code executes one very specific communication strategy based on input from a user who did extensive timing measurements on several different hardware platforms. Rather than broadcasting all data at once asynchronously, it selects combinations of pairs of processors and has them execute a SendRecv operation, which does both a data send and a data receive in a single call. At each step in the execution, different pairs of processors exchange data until all pair combinations have been selected.

The table itself must be a power of 2 in size; the number of PETs is rounded up to the next power of 2 and then all entries for PETs larger than the actual number are marked as no-ops.

There are many alternative execution strategies, including a completely asynchronous execution, in numeric PET order, without computing processor pairs. Also single-direction communications are possible (only the Send XPackets are processed, or only the Receive XPackets) in either a synchronous or asynchronous mode. This would not require any changes to the XPacket or RTable classes, but would require writing a set of alternative RouteRun methods.

6. The current RouteRun routine has many possible performance options for how to make the tradeoff between time spent packing disjoint memory blocks into a single buffer to minimize the number of sends, verses simply sending the contiguous blocks without the pack overhead. The tradeoffs are not expected to be the same on all systems; hardware latency verses bandwidth characteristics will differ, plus the underlying communication software (MPI, shared memory, etc) will change the performance. Also the size of the data blocks to be sent, the amount of contiguity, and limits on the number of outstanding communication buffers all affect what options are best.
The ESMF_RouteOptions are listed in 26.3; the following description contains more implementation detail about what each of the options controls inside the execution of a Route. Note that the options do not affect the creation of a Route, nor any of the Precompute code, and can optionally be changed each time the Route is run.

Packing options:

By Buffer If multiple memory addresses are provided to RouteRun (from bundle-level communications, for example), then this option packs data across all buffers/blocks as specified by the other packing flags before sending or receiving. Note: unlike the other packing flags, this is handled in the code at a higher level by either passing down multiple addresses into the route run routine or not. If multiple addresses are passed into the run routine, they will be packed. The "no-packing" option at this level would be identical to looping at the outermost level in the RouteRun code and therefore there is no disadvantage to calling this routine once per address (and the advantage is not adding yet another coding loop inside the already complex RouteRun code). The higher level list-of-address code can be disabled by clearing this flag (which is on by default).

By PET All data from a single block intended for a remote PET is packed into a single send buffer, and sent in a single VM communications call. A buffer large enough to receive all data coming from that remote PET is allocated, the data is received, and then the data is copied into the final location. See 28.

By XP All data described by a single XPacket (which is an n-dimensional hyperslab of memory) is packed into a single buffer for sending, and a single buffer large enough to receive an XPacket is allocated for receiving the data. See 27.

No Packing A VM communication call is made for each single contiguous strip of memory, regardless of how long or short.

MPI Vector MPI implements a set of interfaces for sending and receiving which allows certain strided memory patterns to be sent in a single call. The actual implementation is up to the MPI library itself. But no user-level data copy is needed in this case. (Not implemented yet.)

Note that in all packing options, if the XPacket describes a chunk of memory which is completely contiguous, then the code does not allocate a packing or unpacking buffer but supplies the actual data address to the communications call so the data is read or written in place.

The following options refer to the internal strategy for executing the route and not to whether the user-level API call returns before the route has finished executing. The current system only implements user-synchronous calls; asynchronous calls are on the to-be-written list.

Sync Each pair of processors exchanges data with the VM equivalent of an MPI_SendRecv() call, which does not return until both the send and receive have completed.

Async Each processor executes both an asynchronous send and asynchronous receive to the other processor and does not wait for completion before moving on to the next communication in the CommTable. Then in a separate loop through the RTables, each call is waited for in turn and when all outstanding communication calls have completed, then the API call returns to the user.

(Note that in the Async case it makes much more sense to iterate through the Route table in PET order instead of the complication of computing communication pairs and iterating in a non-sequential order. The code is as it is now for reasons of implementation speed and not for any other design reason. This would require a slightly simpler, but separate, version of the RouteRun() subroutine.)
Figure 24: A common XPacket pattern which generally benefits from packing; the overlap region between 2 DEs during a halo update are often short in the contiguous dimension and have a high repeat count.
Figure 25: When there are multiple XPackets destined for the same remote PET there are more options for how to order the contiguous pieces into a packed buffer.
Figure 26: When the XPacket describes memory which is physically a single contiguous region, there is no need to copy the data into another buffer; it can be communicated in place. There is a flag in the XPacket which marks how many of the dimensions are contiguous.
Figure 27: Often the overhead of making multiple communication calls outweighs the cost of copying non-contiguous data into a contiguous buffer, sending it in a single operation, and then copying it to the final memory locations on the receiving side.
Figure 28: Once there is more than a single XPacket to pack, there are many more interleave options. For example, packing in the order: 1, 4, 2, 5, 3, 6 would also be possible here. However, the code becomes more complicated when the XPackets have different repeat counts, and has no real performance advantage over the straightforward packing of each XPacket in sequence. Note that this packing is the same whether it refers to multiple XPackets from the same memory buffer or from multiple buffers.
7. FieldBundle-level communication calls have additional packing options under certain circumstances. FieldBundles are groups of Fields which share the same Grid, but they are not required to share the same data types, data ranks, nor relative data locations. FieldBundles in which these things are the same in all Fields are marked inside the bundle code as being congruent. At communication store time FieldBundles which have congruent data in all the Fields have the option of packing all Field data together into fewer communication calls which generally is expected to give better performance. Fields where the data is not of the same type or perhaps not the same number of items (e.g. different rank, vertex-centered data vs. cell centered data) can in theory also be packed but in fact the code becomes more complicated, and in the case of differing data types may cause system errors because of accessing data on non-standard byte offsets or putting mixing integer data with floating data and causing NaN (not a number) exceptions. In this case, the conservative implementation strategy is to construct a separate Route object for each Field, all enclosed in the same RouteHandle. Inside the FieldBundle communication code the execution for both types of FieldBundles is identical for the caller, but inside the congruent FieldBundle code calls the ESMF_RouteRun() code once and all communication for all Fields in the FieldBundle is done when it returns. The non-congruent FieldBundles execute a separate ESMF_RouteRun() call for each Field and return to the user when all Field data have been sent/received.

There are comments in the code for an intermediate level of optimization in which the FieldBundle code determines the smallest number of unique types of Fields in the FieldBundle, and all same types share the same Route object, but this has not been implemented at this time. Once the existing code has been in use for a while, whether this is useful or needed may become more clear.

8. The precompute code for all operations must have enough information to compute which parts of the data arrays are expected to be sent to remote PETs and also what remote data is expected to be received by this PET. These computations depend heavily on what type of distributed method is being executed. The regridding methods are described in detail separately in the Regrid Design and Implementation Notes section. The halo and redistribution operations are described here.

Halo The total array area, which includes any halo regions, are intersected with the computational area of other DEs. The overlap regions are converted from index space into memory space and stored as XPackets in the RTables. This code must be aware of: whether the grid was defined as periodic in any or all of the dimensions since that affects which halo regions overlap at the grid edges; if the data is only decomposed into a single block in any dimension (which means it halos with itself); and if the halo region is large enough that a halo operation may require intersection with the N+1 neighbor in any dimension.

Redistribute Each DE computes the overlap between its own computational region and all DEs in the remote Grid, again only working in computational area. The overlap regions are converted from index space into memory space and stored as XPackets in the RTables. After execution a redistribution, a halo operation may be required to populate any halo regions with consistent data.

(Note: the Redistribution code has been reimplemented to intersect the DEs in index space and then convert the overlap region to an XPacket representation. Halo still converts the regions from AxisIndex to XPackets and then intersects the XPackets, but this code needs to be changed to intersect in AxisIndex space and once the overlap is computed then convert to XPackets. Intersecting AxisIndex objects is very much simpler, both to understand and to execute, and more easily extensible to multiple dimensions than intersecting XPackets.)

26.5 Object Model

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

Infrastructure: Utilities
27 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.
28 Attribute Class

28.1 Description

The ESMF Attribute class is used to store metadata describing other objects in the framework. 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. 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.
Currently Attributes can be defined for the following ESMF objects:

- FieldBundle
- Field
- State

28.2 Restrictions and Future Work

Subsequent releases of ESMF will include the concept of an Attribute hierarchy. This will enable, for example, the Attributes of the Fields in a Bundle to be associated with the Bundle. Future work will also include the association of a purpose, convention (e.g., CF), and I/O format (e.g., plain text, XML) with sets of Attributes.

28.3 Class API

28.3.1 ESMF_AttributeGet - Retrieve the value of an Attribute

INTERFACE:

    subroutine ESMF_AttributeGet(<object>, name, <value argument>, rc)

ARGUMENTS:

- <object>, see below for supported values
- character (len = *), intent(in) :: name
- <value argument>, see below for supported values
- integer, intent(out), optional :: rc

DESCRIPTION:

Returns the value of an Attribute given its name.
Supported values for <object> are:

- type(ESMF_Array), intent(inout) :: array
- type(ESMF_Grid), intent(inout) :: grid
- type(ESMF_Field), intent(inout) :: field
- type(ESMF_FieldBundle), intent(inout) :: fieldbundle
- type(ESMF_State), intent(inout) :: state

Supported values for <value argument> are:
integer(ESMF_KIND_I4), intent(out) :: value
integer(ESMF_KIND_I8), intent(out) :: value
real (ESMF_KIND_R4), intent(out) :: value
real (ESMF_KIND_R8), intent(out) :: value
type(ESMF_Logical), intent(out) :: value
character (len = *), intent(out), value

The arguments are:
<object> The object containing the Attribute.
name The name of the Attribute to retrieve.
<value argument> The value of the named Attribute.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

28.3.2 ESMF_AttributeGet - Retrieve the values of a list Attribute

INTERFACE:
    subroutine ESMF_AttributeGet(<object>, name, count, <value argument>, rc)

ARGUMENTS:
    <object>, see below for supported values
    character (len = *), intent(in) :: name
    integer, intent(in) :: count
    <value argument>, see below for supported values
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns the values of a list Attribute given its name.
Supported values for <object> are:

    type(ESMF_Array), intent(inout) :: array
    type(ESMF_Grid), intent(inout) :: grid
    type(ESMF_Field), intent(inout) :: field
    type(ESMF_FieldBundle), intent(inout) :: fieldbundle
    type(ESMF_State), intent(inout) :: state

Supported values for <value argument> are:

    integer(ESMF_KIND_I4), dimension(:), intent(out) :: valueList
    integer(ESMF_KIND_I8), dimension(:), intent(out) :: valueList
    real (ESMF_KIND_R4), dimension(:), intent(out) :: valueList
    real (ESMF_KIND_R8), dimension(:), intent(out) :: valueList
type(ESMF_Logical), dimension(:), intent(out) :: valueList

The arguments are:

<object> The object containing the Attribute.

name The name of the Attribute to retrieve.

count The number of values in the Attribute.

<value argument> The value of the named Attribute.

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

28.3.3 ESMF_AttributeGet - Query the number of Attributes

INTERFACE:

    subroutine ESMF_AttributeGet(<object>, count, rc)

ARGUMENTS:

    <object>, see below for supported values
    integer, intent(out) :: count
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns the number of Attributes associated with the given object in the argument count. Supported values for <object> are:

    type(ESMF_Array), intent(inout) :: array
    type(ESMF_Grid), intent(inout) :: grid
    type(ESMF_Field), intent(inout) :: field
    type(ESMF_FieldBundle), intent(inout) :: fieldbundle
    type(ESMF_State), intent(inout) :: state

The arguments are:

<object> The object to be queried.

count The number of Attributes associated with this object.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
28.3.4 ESMF_AttributeGet - Query an Attribute by name

**INTERFACE:**

```
subroutine ESMF_AttributeGet(<object>, name, typekind, count, rc)
```

**ARGUMENTS:**

- `<object>`, see below for supported values
- `character(len=*), intent(in) :: name`
- `type(ESMF_TypeKind), intent(out), optional :: typekind`
- `integer, intent(out), optional :: count`
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Returns information associated with the named Attribute, including `typekind` and `count`. Supported values for `<object>` are:

- `type(ESMF_Array), intent(inout) :: array`
- `type(ESMF_Grid), intent(inout) :: grid`
- `type(ESMF_Field), intent(inout) :: field`
- `type(ESMF_FieldBundle), intent(inout) :: fieldbundle`
- `type(ESMF_State), intent(inout) :: state`

The arguments are:

- `<object>` The object containing the Attribute.
- `name` The name of the Attribute to query.
- `[typekind]` The typekind of the Attribute.
- `[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.

28.3.5 ESMF_AttributeGet - Query an Attribute by index number

**INTERFACE:**

```
subroutine ESMF_AttributeGet(<object>, attributeIndex, name, &
   typekind, itemcount, rc)
```

**ARGUMENTS:**

- `<object>`, see below for supported values
- `integer, intent(in) :: attributeIndex`
- `character(len=*) , intent(out) :: name`
- `type(ESMF_TypeKind), intent(out), optional :: typekind`
- `integer, intent(out), optional :: itemcount`
- `integer, intent(out), optional :: rc`
DESCRIPTION:

Returns information associated with the indexed Attribute, including name, typekind and itemcount. Supported values for <object> are:

- type(ESMF_Array), intent(inout) :: array
- type(ESMF_Grid), intent(inout) :: grid
- type(ESMF_Field), intent(inout) :: field
- type(ESMF_FieldBundle), intent(inout) :: fieldbundle
- type(ESMF_State), intent(inout) :: state

The arguments are:

<object> The object to be queried.
attributeIndex The index number of the Attribute to query.
name Returns the name of the Attribute.
[typekind] The typekind of the Attribute.
[itemcount] 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.

28.3.6 ESMF_AttributeSet - Set the value of an Attribute

INTERFACE:

subroutine ESMF_AttributeSet(<object>, name, <value argument>, rc)

ARGUMENTS:

<object>, see below for supported values
character (len = *), intent(in) :: name
<value argument>, see below for supported values
integer, intent(out), optional :: rc

DESCRIPTION:

Sets the value of an Attribute.
Supported values for <object> are:

- type(ESMF_Array), intent(inout) :: array
- type(ESMF_Grid), intent(inout) :: grid
- type(ESMF_Field), intent(inout) :: field
- type(ESMF_FieldBundle), intent(inout) :: fieldbundle
- type(ESMF_State), intent(inout) :: state

Supported values for the <value argument> are:
The arguments are:

<object> The object containing the Attribute to set.

name The name of the Attribute to set.

/value argument> The value of the Attribute to set.

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

### 28.3.7 ESMF_AttributeSet - Set the values of a list Attribute

**INTERFACE:**

```fortran
subroutine ESMF_AttributeSet(<object>, name, count, <value argument>, rc)
```

**ARGUMENTS:**

```fortran
<object>, see below for supported values
character (len = *), intent(in) :: name
integer, intent(in) :: count
/value argument>, see below for supported values
integer, intent(out), optional :: rc
```

**DESCRIPTION:**

Sets the value of an Attribute.

Supported values for <object> are:

```fortran
type(ESMF_Array), intent(inout) :: array
type(ESMF_Grid), intent(inout) :: grid
type(ESMF_Field), intent(inout) :: field
type(ESMF_FieldBundle), intent(inout) :: fieldbundle
type(ESMF_State), intent(inout) :: state
```

Supported values for the <value argument> are:

```fortran
integer(ESMF_KIND_I4), dimension(:), intent(in) :: valueList
integer(ESMF_KIND_I8), dimension(:), intent(in) :: valueList
real (ESMF_KIND_R4), dimension(:), intent(in) :: valueList
real (ESMF_KIND_R8), dimension(:), intent(in) :: valueList
```
type(ESMF_Logical), dimension(:), intent(in) :: valueList

The arguments are:

<object>  The object containing the Attribute to set.

name   The name of the Attribute to set.

count  The number of values in the Attribute.

/value argument>  The value of the Attribute to set.

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

## 29 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.

<table>
<thead>
<tr>
<th>Key Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drift-free timekeeping through an integer-based internal time representation.</td>
</tr>
<tr>
<td>The ability to represent time as a rational fraction, to support exact timekeeping in applications that involve grid refinement.</td>
</tr>
<tr>
<td>Support for many calendar types, including user-customized calendars.</td>
</tr>
<tr>
<td>Support for both concurrent and sequential modes of component execution.</td>
</tr>
<tr>
<td>Support for varying and negative time steps.</td>
</tr>
</tbody>
</table>

### 29.1 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.

### 29.2 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**: The standard Julian date calendar.
- **Julian Day**: The standard Julian days calendar.
- **360-day**: A 30-day-per-month, 12-month-per-year calendar.
- **no calendar**: Tracks only elapsed model time in hours, minutes, seconds.

See Section 30.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

### 29.3 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 31.1 and 32.1 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.

### 29.4 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 33.1 and 34.1 describe the use of Clocks and Alarms in detail.
Table 1: Specifiers for Times and TimeIntervals

<table>
<thead>
<tr>
<th>Unit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;yy</td>
<td>yy_i8&gt;</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>&lt;d</td>
<td>d_i8</td>
</tr>
<tr>
<td>&lt;h</td>
<td>h_r8&gt;</td>
</tr>
<tr>
<td>&lt;m</td>
<td>m_r8&gt;</td>
</tr>
<tr>
<td>&lt;s</td>
<td>s_i8</td>
</tr>
<tr>
<td>&lt;ms</td>
<td>ms_r8&gt;</td>
</tr>
<tr>
<td>&lt;us</td>
<td>us_r8&gt;</td>
</tr>
<tr>
<td>&lt;ns</td>
<td>ns_r8&gt;</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>
</tbody>
</table>

29.5 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 positive 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.
29.6 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.
30  Calendar Class

30.1  Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, Julian Day, 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 [9] and [6].

30.2  Calendar Options

30.2.1  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: 3/1/4801 BC to 10/29/292,277,019,914
The Gregorian calendar is the calendar currently in use throughout Western countries. Named after Pope Gregory XIII, 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  Valid range: 3/1/4713 BC to 4/24/292,271,018,333
The Julian calendar was introduced by Julius Caesar in 46 B.C., and reached its final form in 4 A.D. 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 year is a leap year if divisible by 4. Days are considered to begin at midnight.

ESMF_CAL_JULIANDAY  Valid range: +/- 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.

30.3  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.

```
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)
```

### 30.3.1 Calendar Creation
This example shows how to create two `ESMF_Calendar`s.

```
! 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)
```

### 30.3.2 Calendar Comparison
This example shows how to compare an `ESMF_Calendar` with a known calendar 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
```

### 30.3.3 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."

30.3.4 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=rc)
end program ESMF_CalendarEx

30.4 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.

30.5 Class API
30.5.1 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.
30.5.2   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.


30.5.3   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.
30.5.4 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.

30.5.5 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.
30.5.6  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.

30.5.7  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.
30.5.8 ESMF_CalendarOperator(=/=) - Test if Calendar Type is not equal to Calendar

INTERFACE:

interface operator(=/=)
  if (calendartype /= calendar) then ... endif
  OR
  result = (calendartype /= calendar)
endif

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.

30.5.9 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_JULIAN, ESMF_CAL_JULIANDAY, ESMF_CAL_NOCALENDAR, and ESMF_CAL_NOLEAP. See Section [30.2] for a description of each calendar type.
  [rc]  Return code; equals ESMF_SUCCESS if there are no errors.
30.5.10  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.

30.5.11  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.

30.5.12 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.

30.5.13 ESMF_CalendarGet - Get Calendar properties

INTERFACE:

    subroutine ESMF_CalendarGet(calendar, name, calendarType, &
        daysPerMonth, monthsPerYear, &
        secondsPerDay, secondsPerYear, &
        daysPerYear, &
        daysPerYearDn, daysPerYearDd, rc)

ARGUMENTS:
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_JULIAN, 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/daysPerYearDd 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.

30.5.14 ESMF_CalendarIsLeapYear - Determine if given year is a leap year

INTERFACE:

    ! Private name; call using ESMF_CalendarIsLeapYear() 
    function ESMF_CalendarIsLeapYearI4(calendar, yy, rc)

RETURN VALUE:

    logical :: ESMF_CalendarIsLeapYearI4
ARGUMENTS:

\[
\begin{align*}
type(ESMF\_Calendar), & \quad \text{intent(inout)} \quad \text{:: calendar} \\
integer(ESMF\_KIND\_I4), & \quad \text{intent(in)} \quad \text{:: yy} \\
\text{integer,} & \quad \text{intent(out), optional} \quad \text{:: rc}
\end{align*}
\]

DESCRIPTION:

Returns true if the given year is a leap year within the given calendar, and false otherwise. See also ESMF_TimeIsLeapYear(). This is a private method; invoke via the public overloaded entry point ESMF_CalendarIsLeapYear(). The arguments are:

calendar ESMF_Calendar to determine leap year within.

yy Year to check for leap year.

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

30.5.15 ESMF_CalendarIsLeapYear - Determine if given year is a leap year

INTERFACE:

\[
\begin{align*}
& \text{function ESMF_CalendarIsLeapYearI8(calendar, yy_i8, rc)} \\
\end{align*}
\]

RETURN VALUE:

\[
\text{logical :: ESMF_CalendarIsLeapYearI8}
\]

ARGUMENTS:

\[
\begin{align*}
type(ESMF\_Calendar), & \quad \text{intent(inout)} \quad \text{:: calendar} \\
integer(ESMF\_KIND\_I8), & \quad \text{intent(in)} \quad \text{:: yy_i8} \\
\text{integer,} & \quad \text{intent(out), optional} \quad \text{:: rc}
\end{align*}
\]

DESCRIPTION:

Returns true if the given year is a leap year within the given calendar, and false otherwise. See also ESMF_TimeIsLeapYear(). This is a private method; invoke via the public overloaded entry point ESMF_CalendarIsLeapYear(). The arguments are:

calendar ESMF_Calendar to determine leap year within.

yy_i8 Year to check for leap year.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
30.5.16 ESMF_CalendarPrint - Print the contents of a Calendar

INTERFACE:

    subroutine ESMF_CalendarPrint(calendar, options, rc)

ARGUMENTS:

    type(ESMF_Calendar), intent(inout) :: 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.
Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.
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.

30.5.17 ESMF_CalendarSet - Set a Calendar to a built-in type

INTERFACE:

    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_JULIAN, ESMF_CAL_JULIANDAY, ESMF_CAL_NOCALENDAR, and ESMF_CAL_NOLEAP. See Section [30.2](#) for a description of each calendar type.

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

### 30.5.18  ESMF_CalendarSet - Set properties of a custom Calendar

#### INTERFACE:

```fortran
! 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.

30.5.19 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.

30.5.20 ESMF_CalendarSetDefault - Set the default Calendar

INTERFACE:

    ! Private name; call using ESMF_CalendarSetDefault()
    subroutine ESMF_CalendarSetDefaultCal(calendar, rc)

ARGUMENTS:

    type(ESMF_Calendar), intent(inout) :: 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.

30.5.21 ESMF_CalendarValidate - Validate a Calendar’s properties

INTERFACE:

    subroutine ESMF_CalendarValidate(calendar, options, rc)

ARGUMENTS:

    type(ESMF_Calendar), intent(inout) :: calendar
    character (len=*), intent(in), optional :: options
    integer, intent(out), optional :: rc

DESCRIPTION:

Checks whether a calendar is valid. Must be one of the defined calendar types. daysPerMonth, daysPerYear, secondsPerDay must all be greater than or equal to zero.

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.
31 Time Class

31.1 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 can 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.[5]

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 30.1. 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.

31.2 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.

```fortran
! 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)
```

31.2.1 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 *, "Timel = "
call ESMF_TimePrint(time1, "string", rc)

31.2.2 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 = timel + timeinterval1 = ", YY, "/", MM, "/", DD, " ", &
H, " ", M, ":", S

31.2.3 Time Comparison
This example shows how to compare two ESMF_Times.

if (time2 > timel) then
print *, "time2 is larger than timel"
else
print *, "timel is smaller than or equal to time2"
endif

! finalize ESMF framework
call ESMF_Finalize(rc=rc)

end program ESMF_TimeEx

31.3 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.
31.4 Class API

31.4.1 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.

31.4.2 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.
31.4.3  ESMF_TimeOperator(-) - Return the difference between two Times

INTERFACE:

interface operator(-)
  time3 = time1 - time2
end interface

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.

31.4.4  ESMF_TimeOperator(==) - Test if Time 1 is equal to Time 2

INTERFACE:

interface operator(==)
  if (time1 == time2) then ... endif
  result = (time1 == time2)
end interface

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.
31.4.5 ESMF_TimeOperator(/=) - Test if Time 1 is not equal to Time 2

INTERFACE:

```fortran
interface operator(/=)
  if (time1 /= time2) then ... endif
  result = (time1 /= time2)
end interface
```

RETURN VALUE:

```fortran
logical :: result
```

ARGUMENTS:

```fortran
  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.

31.4.6 ESMF_TimeOperator(<) - Test if Time 1 is less than Time 2

INTERFACE:

```fortran
interface operator(<)
  if (time1 < time2) then ... endif
  result = (time1 < time2)
end interface
```

RETURN VALUE:

```fortran
logical :: result
```

ARGUMENTS:

```fortran
  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.
31.4.7 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.

---

31.4.8 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.
31.4.9 ESMF_TimeOperator(>=) - Test if Time 1 is greater than or equal to Time 2

INTERFACE:

```fortran
interface operator(>=)
if (time1 >= time2) then ... endif
OR
result = (time1 >= time2)
end interface
```

RETURN VALUE:

```fortran
logical :: result
```

ARGUMENTS:

```fortran
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.

---

31.4.10 ESMF_TimeGet - Get a Time value

INTERFACE:

```fortran
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, timeStringISO, &
    dayOfWeek, midMonth, &
    dayOfYear, dayOfYear_r8, &
    dayOfYear_intvl, rc)
```

ARGUMENTS:

```fortran
type(ESMF_Time), intent(inout) :: 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
```

---

329
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). (Reals not implemented yet).

Units are bound (normalized) by the next larger unit specified. For example, if a time is defined to be 2:00 am on February 2, 2004, then ESMF_TimeGet(dd=day, h=hours, s=seconds) would return day = 2, hours = 2, seconds = 0, whereas ESMF_TimeGet(dd = day, s=seconds) would return day = 2, seconds = 7200. Note that hours and seconds are bound by a day. If bound by a month, ESMF_TimeGet(mm=month, h=hours, s=seconds) would return month = 2, hours = 26, seconds = 0, and ESMF_TimeGet(mm = month, s=seconds) would return month = 2, seconds = 93600 (26 * 3600). Similarly, if bound to a year, ESMF_TimeGet(yy=year, h=hours, s=seconds) would return year = 2004, hours = 770 (32*24+2), seconds = 0, and ESMF_TimeGet(yy = year, s=seconds) would return year = 2004, seconds = 2772000 (770 * 3600).

For timeString, timeStringISOFrac, dayOfWeek, midMonth, dayOfYear, dayOfYear_r8, and dayOfYear_intvl described below, valid calendars are Gregorian, Julian, No Leap, 360 Day and Custom calendars. Not valid for Julian Day or No Calendar.

For timeString and timeStringISOFrac, YYYY format returns at least 4 digits; years <= 999 are padded on the left with zeroes and years >= 10000 return the number of digits required.

For timeString, convert ESMF_Time's value into partial ISO 8601 format YYYY-MM-DDThh:mm:ss[:n/d]. See [5] and [2]. See also method ESMF_TimePrint().

For timeStringISOFrac, convert ESMF_Time's value into full ISO 8601 format YYYY-MM-DDThh:mm:ss.[f]. See [5] and [2]. See also method ESMF_TimePrint().
For dayOfWeek, gets the day of the week the given ESMF_Time instant falls on. ISO 8601 standard: Monday = 1 through Sunday = 7. See [5] and [2].

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 discussion 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 discussion in argument list below. Return as floating point value; fractional part represents the time of day. (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.
- **[us]** Integer microseconds.
- **[ns]** Integer nanoseconds.
- **[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).
- **[sD]** Integer denominator portion of fractional seconds (sN/sD).
- **[calendar]** Associated Calendar.
- **[calendarType]** Associated CalendarType.
- **[timeZone]** Associated timezone (hours offset from UCT, e.g. EST = -5). (Not implemented yet).
 Convert time value to format string YYYY-MM-DDThh:mm:ss[: n/d], where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [5] and [2]. See also method ESMF_TimePrint().

Convert time value to strict ISO 8601 format string YYYY-MM-DDThh:mm:ss[.f], where f is decimal form of any fractional seconds. See [5] and [2]. See also method ESMF_TimePrint().

The time instant’s day of the week [1-7].

The given time instant’s middle-of-the-month time instant.


The ESMF_Time instant’s day of the year as an ESMF_TimeInterval.

Return code; equals ESMF_SUCCESS if there are no errors.

31.4.11 ESMF_TimeIsLeapYear - Determine if a Time is in a leap year

INTERFACE:

function ESMF_TimeIsLeapYear(time, rc)

RETURN VALUE:

logical :: ESMF_TimeIsLeapYear

ARGUMENTS:

type(ESMF_Time), intent(inout) :: time
integer, intent(out), optional :: rc

DESCRIPTION:

Returns true if given time is in a leap year, and false otherwise. See also ESMF_CalendarIsLeapYear().

The arguments are:

time The ESMF_Time to check for leap year.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

31.4.12 ESMF_TimeIsSameCalendar - Compare Calendars of two Times

INTERFACE:

function ESMF_TimeIsSameCalendar(time1, time2, rc)
**RETURN VALUE:**

```plaintext
logical :: ESMF_TimeIsSameCalendar
```

**ARGUMENTS:**

```plaintext
type(ESMF_Time), intent(inout) :: time1
type(ESMF_Time), intent(inout) :: 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.

---

### 31.4.13 ESMF_TimePrint - Print the contents of a Time

**INTERFACE:**

```plaintext
subroutine ESMF_TimePrint(time, options, rc)
```

**ARGUMENTS:**

```plaintext
type(ESMF_Time), intent(inout) :: 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. For options "string" and "string isofrac", YYYY format returns at least 4 digits; years <= 999 are padded on the left with zeroes and years >= 10000 return the number of digits required.

Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call `flush(6)` as a workaround until this issue is fixed in a future release.

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 for all units through seconds. For any non-zero fractional seconds, prints in integer rational fraction form n/d. Format is YYYY-MM-DDThh:mm:ss[.f][n/d], where [n/d] is the integer numerator and denominator of the fractional seconds value, if present. See [5] and [2]. See also method ESMF_TimeGet(..., timeString= , ...)"string isofrac" - prints time’s value in strict ISO 8601 format for all units, including any fractional seconds part. Format is YYYY-MM-DDThh:mm:ss[.f] where [f] represents fractional seconds in decimal form, if present. See [5] and [2]. See also method ESMF_TimeGet(..., timeStringISOFrac= , ...)
Return code; equals ESMF_SUCCESS if there are no errors.

31.4.14  ESMF_TimeSet - Initialize or set a Time

INTERFACE:

```fortran
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:

```fortran
  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_I8), intent(in), optional :: h_r8
  integer(ESMF_KIND_I4), intent(in), optional :: m
  integer(ESMF_KIND_I8), intent(in), optional :: m_r8
  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
  integer(ESMF_KIND_I8), intent(in), optional :: ms_r8
  integer(ESMF_KIND_I4), intent(in), optional :: us
  integer(ESMF_KIND_I8), intent(in), optional :: us_r8
  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
  integer(ESMF_KIND_I4), intent(in), optional :: sN
  integer(ESMF_KIND_I4), intent(in), optional :: sD
  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 and Julian calendars, where the lowest date limits are 3/1/-4800 and 3/1/-4712, respectively. This is a limitation of the Gregorian date-to-Julian day and Julian date-to-Julian day conversion algorithms used to convert Gregorian and Julian dates to the internal representation of seconds. See [3] for a description of the Gregorian date-to-Julian day algorithm and [4] for a description of the Julian date-to-Julian day algorithm. The Custom calendar will have user-defined values for yy, yy_i8, mm, and dd.

The Julian day specifier, d or d_i8, can only be used with the Julian-day calendar, and has a valid range depending on the word size. For a signed 32-bit d, the range is [+/- 24855]. For a signed 64-bit d or d_i8, the valid range is [+/- 106,751,991,167,300]. The Julian day number system adheres to the conventional standard where the reference day of d=0 corresponds to 11/24/-4713 in the Gregorian calendar and 1/1/-4712 in the Julian calendar. See [7] and [1]. Note that d and d_i8 are not valid for the No-Calendar. To remain consistent with non-Earth calendars added to ESMF in the future, ESMF requires a calendar to be planet-specific. Hence the No-Calendar does not know what a day is; it cannot assume an Earth day of 86400 seconds.

Hours, minutes, seconds, and sub-seconds can be used with any calendar, since they are standardized units that are the same for any planet.

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 are represented internally with an integer numerator and denominator fraction (sN/sD). The smallest resolution is 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. (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
- **[us]** Integer microseconds. Default = 0
- **[ns]** Integer nanoseconds. Default = 0
- **[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.
[sD] Integer denominator portion of fractional seconds (sN/sD). Default = 1.
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.

31.4.15 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.

31.4.16 ESMF_TimeValidate - Validate a Time

INTERFACE:

    subroutine ESMF_TimeValidate(time, options, rc)

ARGUMENTS:

    type(ESMF_Time), intent(inout) :: time
    character (len=*) , intent(in), optional :: options
    integer, intent(out), optional :: rc
DESCRIPTION:

Checks whether an ESMF_Time is valid. Must be a valid date/time on a valid calendar. 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.
32 TimeInterval Class

32.1 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 31.1 Table 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 33.1) and Alarms (Section 34.1).

32.2 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 33.1.

The following brief example shows how to create, initialize and manipulate TimeInterval.

```plaintext
! 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)
```

338
32.2.1 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)

32.2.2 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."

32.2.3 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."

32.2.4 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."
32.2.5 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=rc)

end program ESMF_TimeIntervalEx

32.3 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.

32.4 Class API

32.4.1 ESMF_TimeIntervalOperator(+) - Add two TimeIntervals

INTERFACE:

    interface operator(+)
    sum = timeinterval1 + timeinterval2
    end interface

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.
32.4.2 ESMF_TimeIntervalOperator(-) - Subtract one TimeInterval from another

INTERFACE:

    interface operator(-)
    difference = timeinterval1 - timeinterval2
    end interface

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.

32.4.3 ESMF_TimeIntervalOperator(-) - Perform unary negation on a TimeInterval

INTERFACE:

    interface operator(-)
    timeinterval = -timeinterval
    end interface

RETURN VALUE:

    type(ESMF_TimeInterval) :: -timeInterval

ARGUMENTS:

    type(ESMF_TimeInterval), intent(in) :: timeinterval

DESCRIPTION:

Overloads the (-) operator for the ESMF_TimeInterval class to perform unary negation on timeinterval and return the result.

The arguments are:

  timeinterval  The time interval to be negated.

32.4.4 ESMF_TimeIntervalOperator(/) - Divide two TimeIntervals, return double precision quotient

INTERFACE:

    interface operator(/)
    quotient = timeinterval1 / timeinterval2
    end interface
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.

---

32.4.5 ESMF_TimeIntervalOperator(/) - Divide a TimeInterval by an integer, return TimeInterval quotient

INTERFACE:

```
interface operator(/
    quotient = timeinterval / divisor
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: quotient
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval

integer(ESMF_KIND_I4), intent(in) :: divisor
```

DESCRIPTION:

Overloads the (/) operator for the ESMF_TimeInterval class to divide a timeinterval by an integer divisor, and return the quotient as an ESMF_TimeInterval.

The arguments are:

- **timeinterval** The dividend.
- **divisor** Integer divisor.

---

32.4.6 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.

32.4.7 ESMF_TimeIntervalOperator(x) - Multiply a TimeInterval by an integer

INTERFACE:

```
interface operator(*)
  product = timeinterval * multiplier
end interface
```

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.
- multiplier The integer multiplier.

32.4.8 ESMF_TimeIntervalOperator(x) - Multiply a TimeInterval by an integer

INTERFACE:

```
interface operator(*)
  product = multiplier * timeinterval
end interface
```

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:

multiplier  The integer multiplier.
timeinterval  The multiplicand.

32.4.9  ESMF_TimeIntervalOperator(==) - Test if TimeInterval 1 is equal to TimeInterval 2

INTERFACE:

interface operator(==)
  if (timeintvall1 == timeinterval2) then ... endif
  OR
  result = (timeintvall1 == timeinterval2)
end interface

RETURN VALUE:

logical :: result

ARGUMENTS:

  type(ESMF_TimeInterval), intent(in) :: timeintvall1
  type(ESMF_TimeInterval), intent(in) :: timeinterval2

DESCRIPTION:

Overloads the (==) operator for the ESMF_TimeInterval class to return true if timeintvall1 and timeinterval2 are equal, and false otherwise.
The arguments are:

timeintvall1  First ESMF_TimeInterval in comparison.
timeinterval2  Second ESMF_TimeInterval in comparison.

32.4.10  ESMF_TimeIntervalOperator(/=) - Test if TimeInterval 1 is not equal to TimeInterval 2

INTERFACE:

interface operator(/=)
  if (timeintvall1 /= timeinterval2) then ... endif
  OR
  result = (timeintvall1 /= timeinterval2)
end interface

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.

32.4.11  ESMF_TimeIntervalOperator(<) - Test if TimeInterval 1 is less than TimeInterval 2

INTERFACE:

    interface operator(<)
        if (timeinterval1 < timeinterval2) then ... endif
    OR
        result = (timeinterval1 < timeinterval2)
    end interface

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.

32.4.12  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)
    end interface

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.

32.4.13 ESMF_TimeIntervalOperator(>) - Test if TimeInterval 1 is greater than TimeInterval 2

INTERFACE:

```
interface operator(>)
  if (timeinterval1 > timeinterval2) then ... endif
  result = (timeinterval1 > timeinterval2)
end interface
```

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.

32.4.14 ESMF_TimeIntervalOperator(>=) - Test if TimeInterval 1 is greater than or equal to TimeInterval 2

INTERFACE:

```
interface operator(>=)
  if (timeinterval1 >= timeinterval2) then ... endif
  result = (timeinterval1 >= timeinterval2)
end interface
```

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.

### 32.4.15 ESMF_TimeIntervalAbsValue - Get the absolute value of a TimeInterval

**INTERFACE:**

```plaintext
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.

### 32.4.16 ESMF_TimeIntervalGet - Get a TimeInterval value

**INTERFACE:**

```plaintext
subroutine ESMF_TimeIntervalGetDur(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, calendar, calendarType, &
  timeString, timeStringISOFrac, rc)
```
ARGUMENTS:

- \texttt{timeinterval} (type(ESMF TimeInterval), intent(inout))
- \texttt{yy} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{yy\_i8} (integer(ESMF_KIND_I8), intent(out), optional)
- \texttt{mm} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{mm\_i8} (integer(ESMF_KIND_I8), intent(out), optional)
- \texttt{d} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{d\_i8} (integer(ESMF_KIND_I8), intent(out), optional)
- \texttt{h} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{m} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{s} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{s\_i8} (integer(ESMF_KIND_I8), intent(out), optional)
- \texttt{ms} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{us} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{ns} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{d\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{h\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{m\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{s\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{ms\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{us\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{ns\_r8} (real(ESMF_KIND_R8), intent(out), optional)
- \texttt{sN} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{sD} (integer(ESMF_KIND_I4), intent(out), optional)
- \texttt{startTime} (type(ESMF\_Time), intent(out), optional)
- \texttt{calendar} (type(ESMF\_Calendar), intent(out), optional)
- \texttt{calendarType} (type(ESMF\_Calendar\_Type), intent(out), optional)
- \texttt{timeString} (character(len=*), intent(out), optional)
- \texttt{timeStringISOFrac} (character(len=*), intent(out), optional)
- \texttt{rc} (integer, intent(out), optional)

DESCRIPTION:

Gets the value of \texttt{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. (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 \texttt{ESMF\_TimeIntervalGet(d = days, s = seconds)} would return \texttt{days = 1, seconds = 0}, whereas \texttt{ESMF\_TimeIntervalGet(s = seconds)} would return \texttt{seconds = 86400}. See ../include/ESMC\_BaseTime.h and ../include/ESMC\_TimeInterval.h for complete description.

For \texttt{timeString}, converts \texttt{ESMF\_TimeInterval}'s value into partial ISO 8601 format PyYmMdDThhMmSs\,[n/d]S. See § and ¶. See also method \texttt{ESMF\_TimeIntervalPrint()}.

For \texttt{timeStringISOFrac}, converts \texttt{ESMF\_TimeInterval}'s value into full ISO 8601 format PyYmMdDThhMmSs\,[.]fS. See § and ¶. See also method \texttt{ESMF\_TimeIntervalPrint()}.

The arguments are:

- \texttt{timeinterval} The object instance to query.
- \texttt{yy} Integer years (\texttt{>= 32-bit}).
- \texttt{yy\_i8} Integer years (large, \texttt{>= 64-bit}).
- \texttt{mm} Integer months (\texttt{>= 32-bit}).
- \texttt{mm\_i8} Integer months (large, \texttt{>= 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.
[us] Integer microseconds.
[ns] Integer nanoseconds.
[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).
[sD] Integer denominator portion of fractional seconds (sN/sD).
[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).
[calendar] Associated Calendar, if any.
[calendarType] Associated CalendarType, if any.
[timeString] Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [5] and [2]. See also method ESMF_TimeIntervalPrint().
[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[.f], where f is decimal form of any fractional seconds. See [5] and [2]. See also method ESMF_TimeIntervalPrint().
[rc] Return code; equals ESMF_SUCCESS if there are no errors.
32.4.17  ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

! Private name; call using ESMF_TimeIntervalGet()
subroutine ESMF_TimeIntervalGetDurStart (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, &
  calendar, calendarType, &
  startTimeIn, &
timeString, timeStringISOFrac, rc)

ARGUMENTS:

type (ESMF_TimeInterval), intent (inout) :: 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_I8), intent (out), optional :: h_r8  ! not implemented  
integer (ESMF_KIND_I4), intent (out), optional :: m  
integer (ESMF_KIND_I8), intent (out), optional :: m_r8  ! not implemented  
integer (ESMF_KIND_I4), intent (out), optional :: s  
integer (ESMF_KIND_I8), intent (out), optional :: s_r8  ! not implemented  
integer (ESMF_KIND_I4), intent (out), optional :: ms  
integer (ESMF_KIND_I8), intent (out), optional :: ms_r8  ! not implemented  
integer (ESMF_KIND_I4), intent (out), optional :: us  
integer (ESMF_KIND_I8), intent (out), optional :: us_r8  ! not implemented  
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  ! 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  
integer (ESMF_KIND_I4), intent (out), optional :: sD  
type (ESMF_Time), intent (out), optional :: startTime  
type (ESMF_Time), intent (inout) :: startTimeIn  ! Input  
type (ESMF_Calendar), intent (out), optional :: calendar  
type (ESMF_CalendarType), intent (out), optional :: calendarType  
type (ESMF_Time), intent (out), optional :: timeString  
character (len=*), intent (out), optional :: timeStringISOFrac  
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. (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 partial ISO 8601 format PyYmMdDTmMs[:n/d]S. See [5] and [2]. See also method ESMF_TimeIntervalPrint(). For timeStringISOFrac, converts ESMF_TimeInterval’s value into full ISO 8601 format PyYmMdDTmMm[f]S. See [5] and [2]. See also method ESMF_TimeIntervalPrint().

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.

[us] Integer microseconds.

[ns] Integer nanoseconds.

[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).

[sD] Integer denominator portion of fractional seconds (sN/sD).

[startTime] Starting 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.

[timeString] Convert time interval value to format string PyYmMdDThHmMs[n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [5] and [2]. See also method ESMF_TimeIntervalPrint().

[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs.f, where f is decimal form of any fractional seconds. See [5] and [2]. See also method ESMF_TimeIntervalPrint().

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

32.4.18  ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

! Private name; call using ESMF_TimeIntervalGet() subroutine ESMF_TimeIntervalGetDurCal(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, &
calendar, calendarType, &
calendarIn, &
timeString, timeStringISOFrac, rc)

ARGUMENTS:

type(ESMF_TimeInterval), intent(inout) :: 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_I4), intent(out), optional :: s_i8
integer(ESMF_KIND_I4), intent(out), optional :: ms
integer(ESMF_KIND_I4), intent(out), optional :: ms_r8
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 ! not implemented
real(ESMF_KIND_R8), intent(out), optional :: m_r8 ! not implemented
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. (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`.

For `timeString`, converts ESMF_TimeInterval's value into partial ISO 8601 format `PyYmMdDThHmMs[:n/d]S`. See [5] and [2]. See also method `ESMF_TimeIntervalPrint()`.

For `timeStringISOFrac`, converts ESMF_TimeInterval's value into full ISO 8601 format `PyYmMdDThHmMs[f]S`. See [5] and [2]. See also method `ESMF_TimeIntervalPrint()`.

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.
- `[us]` Integer microseconds.
- `[ns]` Integer nanoseconds.
- `[d_r8]` Double precision days. (Not implemented yet).
Double precision hours. (Not implemented yet).

Double precision minutes. (Not implemented yet).

Double precision seconds. (Not implemented yet).

Double precision milliseconds. (Not implemented yet).

Double precision microseconds. (Not implemented yet).

Double precision nanoseconds. (Not implemented yet).

Integer numerator portion of fractional seconds (sN/sD).

Integer denominator portion of fractional seconds (sN/sD).

Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).

Associated Calendar, if any.

Associated CalendarType, if any.

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.

Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See \[5\] and \[2\]. See also method ESMF_TimeIntervalPrint().

Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[,f], where f is decimal form of any fractional seconds. See \[5\] and \[2\]. See also method ESMF_TimeIntervalPrint().

Return code; equals ESMF_SUCCESS if there are no errors.

32.4.19 ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

! Private name; call using ESMF_TimeIntervalGet()
subroutine ESMF_TimeIntervalGetDurCalTyp(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, &
    calendar, calendarType, &
    calendarTypeIn, &
    timeString, &
    timeStringISOFrac, rc)
ARGUMENTS:

<table>
<thead>
<tr>
<th>Type</th>
<th>Intent</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type(ESMF_TimeInterval), intent(inout)</td>
<td>:: timeinterval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: yy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I8), intent(out), optional</td>
<td>:: yy_i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: mm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I8), intent(out), optional</td>
<td>:: mm_i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I8), intent(out), optional</td>
<td>:: d_i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: s_i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: us</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(ESMF_KIND_R8), intent(out), optional</td>
<td>:: d_r8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(ESMF_KIND_R8), intent(out), optional</td>
<td>:: h_r8 ! not implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(ESMF_KIND_R8), intent(out), optional</td>
<td>:: m_r8 ! not implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(ESMF_KIND_R8), intent(out), optional</td>
<td>:: s_r8 ! not implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(ESMF_KIND_R8), intent(out), optional</td>
<td>:: ms_r8 ! not implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(ESMF_KIND_R8), intent(out), optional</td>
<td>:: us_r8 ! not implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: sN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer(ESMF_KIND_I4), intent(out), optional</td>
<td>:: sD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ESMF_Time), intent(out), optional</td>
<td>:: startTime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ESMF_Calendar), intent(out), optional</td>
<td>:: calendar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ESMF_CalendarType), intent(out), optional</td>
<td>:: calendarType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ESMF_CalendarType), intent(in)</td>
<td>:: calendarTypeIn ! Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>character (len=*)</td>
<td>intent(out), optional</td>
<td>:: timeString</td>
<td></td>
</tr>
<tr>
<td>character (len=*)</td>
<td>intent(out), optional</td>
<td>:: timeStringISOFrac</td>
<td></td>
</tr>
<tr>
<td>integer,</td>
<td>intent(out), optional</td>
<td>:: rc</td>
<td></td>
</tr>
</tbody>
</table>

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. (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 partial ISO 8601 format PyYmMdDThHmMs[n/d]S. See \[5\] and \[2\]. See also method ESMF_TimeIntervalPrint() .

For timeStringISOFrac, converts ESMF_TimeInterval's value into full ISO 8601 format PyYmMdDThHmMs[f]S. See \[5\] and \[2\]. See also method ESMF_TimeIntervalPrint() .

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).
[dd]   Integer Julian days (>= 32-bit).
[dd_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.
[us]   Integer microseconds.
[ns]   Integer nanoseconds.
[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).
[sD]   Integer denominator portion of fractional seconds (sN/sD).
[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).
[calendar] Associated Calendar, if any.
[calendarType] Associated CalendarType, if any.
[calendarTypeIn] INPUT argument: Alternate to, and mutually exclusive with, calendarIn above. More convenient way of specifying a built-in calendar type.
[timeString] Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [5] and [2]. See also method ESMF_TimeIntervalPrint().
[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[.f], where f is decimal form of any fractional seconds. See [5] and [2]. See also method ESMF_TimeIntervalPrint().
[rc] Return code; equals ESMF_SUCCESS if there are no errors.
32.4.20  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(inout) :: 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.

32.4.21  ESMF_TimeIntervalPrint - Print the contents of a TimeInterval

INTERFACE:

    subroutine ESMF_TimeIntervalPrint(timeinterval, options, rc)

ARGUMENTS:

    type(ESMF_TimeInterval), intent(inout) :: 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.
Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order.  If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.
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 for all units through seconds. For any non-zero fractional seconds, prints in integer rational fraction form n/d. Format is PyYmMdDThHmMs[:n/d]S, where [:n/d] is the integer numerator and denominator of the fractional seconds value, if present. See [5] and [2]. See also method ESMF_TimeIntervalGet(..., timeString= , ...)
"string isofrac" - prints TimeInterval's value in strict ISO 8601 format for all units, including any fractional seconds part. Format is PyYmMdDTThMmMss[.f]S, where [.f] represents fractional seconds in decimal form, if present. See [5] and [2]. See also method ESMF_TimeIntervalGet(..., timeStringISOFrac= , ...)

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

32.4.22 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

! Private name; call using ESMF_TimeIntervalSet()
subroutine ESMF_TimeIntervalSetDur(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, 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_I4), 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_I4), intent(in), optional :: s_i8
integer(ESMF_KIND_I4), intent(in), optional :: ms
integer(ESMF_KIND_I4), intent(in), optional :: us
integer(ESMF_KIND_I4), intent(in), optional :: ns
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
integer(ESMF_KIND_I4), intent(in), optional :: sD
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. (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.
- [us] Integer microseconds. Default = 0.
- [ns] Integer nanoseconds. Default = 0.
- [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.
- [sD] Integer denominator portion of fractional seconds (sN/sD). Default = 1.
- [rc] Return code; equals ESMF_SUCCESS if there are no errors.
32.4.23  ESMF_TimeIntervalSet - Initialize or set a TimeInterval

**INTERFACE:**

```fortran
! Private name; call using ESMF_TimeIntervalSet()
subroutine ESMF_TimeIntervalSetDURStart(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, rc)
```

**ARGUMENTS:**

```fortran
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_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
integer(ESMF_KIND_I4), intent(in), optional :: us
integer(ESMF_KIND_I4), intent(in), optional :: ns
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
integer(ESMF_KIND_I4), intent(in), optional :: sN
integer(ESMF_KIND_I4), intent(in), optional :: sD
type(ESMF_Time), intent(in) :: startTime
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. (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 (\(\geq 32\)-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.
[us] Integer microseconds. Default = 0.
[ns] Integer nanoseconds. Default = 0.
[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.
[sD] Integer denominator portion of fractional seconds (sN/sD). Default = 1.

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 calendar also not set, calendar interval "floats" across all calendars and times.

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

32.4.24 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

    ! Private name; call using ESMF_TimeIntervalSet()
    subroutine ESMF_TimeIntervalSetSetDurCal(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, calendar, 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 :: us`
- `integer(ESMF_KIND_I4), intent(in), optional :: ns`
- `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
- `integer(ESMF_KIND_I4), intent(in), optional :: sN`
- `integer(ESMF_KIND_I4), intent(in), optional :: sD`
- `type(ESMF_Calendar), intent(in) :: calendar`
- `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. (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.
[us] Integer microseconds. Default = 0.

[ns] Integer nanoseconds. Default = 0.

[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.

[sD] Integer denominator portion of fractional seconds (sN/sD). Default = 1.

[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 also not specified, calendar interval "floats" across all calendars and times. Mutually exclusive with startTime since it contains a calendar. Alternate to, and mutually exclusive with, calendarType below. Primarily for specifying a custom calendar type.

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

32.4.25 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

! Private name; call using ESMF_TimeIntervalSet()
subroutine ESMF_TimeIntervalSetDurCalTyp(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, 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 :: h
integer(ESMF_KIND_I8), intent(in), optional :: d_i8
integer(ESMF_KIND_I8), intent(in), optional :: s_i8
integer(ESMF_KIND_I8), intent(in), optional :: ms
integer(ESMF_KIND_I8), intent(in), optional :: us
integer(ESMF_KIND_I8), intent(in), optional :: ns
integer(ESMF_KIND_I8), intent(in), optional :: ms_r8
integer(ESMF_KIND_I8), intent(in), optional :: us_r8
integer(ESMF_KIND_I8), intent(in), optional :: ns_r8
integer(ESMF_KIND_I8), intent(in), optional :: sN
integer(ESMF_KIND_I8), intent(in), optional :: sD
character(1), intent(in), optional :: calendarType
integer(ESMF_KIND_I4), intent(out) :: 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. (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.

[us] Integer microseconds. Default = 0.

[ns] Integer nanoseconds. Default = 0.

[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).
Double precision seconds. Default = 0.0. (Not implemented yet).

Double precision milliseconds. Default = 0.0. (Not implemented yet).

Double precision microseconds. Default = 0.0. (Not implemented yet).

Double precision nanoseconds. Default = 0.0. (Not implemented yet).

Integer numerator portion of fractional seconds (sN/sD). Default = 0.

Integer denominator portion of fractional seconds (sN/sD). Default = 1.

Alternate to, and mutually exclusive with, calendar above. More convenient way of specifying a built-in calendar type.

Return code; equals ESMF_SUCCESS if there are no errors.

32.4.26 ESMF_TimeIntervalValidate - Validate a TimeInterval

INTERFACE:

subroutine ESMF_TimeIntervalValidate(timeinterval, options, rc)

ARGUMENTS:

  type(ESMF_TimeInterval), intent(inout) :: timeinterval
  character (len=*), intent(in), optional :: options
  integer, intent(out), optional :: rc

DESCRIPTION:

Checks whether a timeinterval is valid. If fractional value, denominator must be non-zero. 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.
33 Clock Class

33.1 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_ClockGet 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 34.1 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, reversing direction, and synchronizing with a real clock.

33.2 Clock Options

33.2.1 ESMF_Direction

DESCRIPTION:
Specifies the time-stepping direction of a clock. Use with "direction" argument to methods ESMF_ClockSet() and ESMF_ClockGet(). Cannot be used with method ESMF_ClockCreate(), since it only initializes a clock in the default forward mode; a clock must be advanced (time-stepped) at least once before reversing direction via ESMF_ClockSet(). This also holds true for negative timestep clocks which are initialized (created) with stopTime < startTime, since "forward" means time-stepping from startTime towards stopTime (see ESMF_MODE_FORWARD below).

"Forward" and "reverse" directions are distinct from positive and negative timesteps. "Forward" means time-stepping in the direction established at ESMF_ClockCreate(), from startTime towards stopTime, regardless of the timestep sign. "Reverse" means time-stepping in the opposite direction, back towards the clock’s startTime, regardless of the timestep sign.

Clocks and alarms run in reverse in such a way that the state of a clock and its alarms after each time step is precisely replicated as it was in forward time-stepping mode. All methods which query clock and alarm state will return the same result for a given timeStep, regardless of the direction of arrival.

Valid values are:

ESMF_MODE_FORWARD Upon calling ESMF_ClockAdvance(), the clock will timestep from its startTime toward its stopTime. This is the default direction. A user can use either ESMF_ClockIsStopTime() or ESMF_ClockIsDone() methods to determine when stopTime is reached. This forward behavior also holds for negative timestep clocks which are initialized (created) with stopTime < startTime.

ESMF_MODE_REVERSE Upon calling ESMF_ClockAdvance(), the clock will timestep backwards toward its startTime. Use method ESMF_ClockIsDone() to determine when startTime is reached. This reverse behavior also holds for negative timestep clocks which are initialized (created) with stopTime < startTime.

33.3 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.

```fortran
! 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
!--------------------------------------------------------------

! ESDF 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)

33.3.1 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)
```

367
! initialize the clock with the above values
clock = ESMF_ClockCreate("Clock 1", timeStep, startTime, stopTime, rc=rc)

33.3.2 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

33.3.3 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."

33.3.4 Clock Reversal
This example shows how to time-step an ESMF_Clock in reverse mode.

call ESMF_ClockSet(clock, direction=ESMF_MODE_REVERSE, rc=rc)

! time step clock in reverse from stop time back to start time;
! note use of ESMF_ClockIsDone() rather than ESMF_ClockIsStopTime()
do while (.not.ESMF_ClockIsDone(clock, rc))

    call ESMF_ClockPrint(clock, "currTime string", rc)

    call ESMF_ClockAdvance(clock, rc=rc)

end do
33.3.5 Clock Destruction

This example shows how to destroy an ESMF_Clock.

```fortran
! destroy clock
call ESMF_ClockDestroy(clock, rc)

! finalize ESMF framework
call ESMF_Finalize(rc=rc)
```

end program ESMF_ClockEx

33.4 Restrictions and Future Work

1. **Alarm list allocation factor** The alarm list within a clock is dynamically allocated automatically, 200 alarm references at a time. This constant is defined in both Fortran and C++ with a #define for ease of modification.

2. **Clock variable timesteps in reverse** In order for a clock with variable timesteps to be run in ESMF MODE REVERSE, the user must supply those timesteps to ESMF_CLOCKADVANCE(). Essentially, the user must save the timesteps while in forward mode. In a future release, the Time Manager will assume this responsibility by saving the clock state (including the timeStep) at every timestep while in forward mode.

33.5 Class API

33.5.1 ESMF_ClockOperator(==) - Test if Clock 1 is equal to Clock 2

**INTERFACE:**

```fortran
interface operator(==)
  if (clock1 == clock2) then ... endif
  OR
  result = (clock1 == clock2)
end interface
```

**RETURN VALUE:**

```fortran
logical :: result
```

**ARGUMENTS:**

```fortran
  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.
33.5.2 ESMF_ClockOperator(/=) - Test if Clock 1 is not equal to Clock 2

INTERFACE:

interface operator(/=)
  if (clock1 /= clock2) then ... endif
  result = (clock1 /= clock2)
endinterface

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.

33.5.3 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(inout), 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). When the clock is in ESMF_MODE_FORWARD (default), this method adds the timeStep to the clock’s current time. In ESMF_MODE_REVERSE, timeStep is subtracted from the current time. In either case, timeStep can be positive or negative. See the "direction" argument in method ESMF_ClockSet(). ESMF_ClockAdvance() 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. timeStep can be positive or negative.

[**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.

---

### 33.5.4 ESMF_ClockCreate - Create a new ESMF Clock

**INTERFACE:**

```fortran
! Private name; call using ESMF_ClockCreate()
function ESMF_ClockCreateNew(name, timeStep, startTime, stopTime, &
    runDuration, runTimeStepCount, refTime, rc)
```

**RETURN VALUE:**

```fortran
type(ESMF_Clock) :: ESMF_ClockCreateNew
```

**ARGUMENTS:**

```fortran
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, which can be positive or negative.

- **startTime** The ESMF_Clock's starting time. Can be less than or greater than stopTime, depending on a positive or negative timeStep, respectively, and whether a stopTime is specified; see below.

- **[stopTime]** The ESMF_Clock's stopping time. Can be greater than or less than the startTime, depending on a positive or negative timeStep, respectively. 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. Can be positive or negative, consistent with the timeStep's sign. Mutually exclusive with stopTime and runTimeStep-Count.
**[runTimeStepCount]** Alternative way to specify ESMF_Clock’s stopping time; \( \text{stopTime} = \text{startTime} + (\text{runTimeStepCount} \times \text{timeStep}) \). stopTime can be before startTime if timeStep is negative. 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.

---

### 33.5.5 ESMF_ClockCreate - Create a copy of an existing ESMF Clock

**INTERFACE:**

```fortran
! Private name; call using ESMF_ClockCreate()
function ESMF_ClockCreateCopy(clock, rc)
```

**RETURN VALUE:**

```fortran
type(ESMF_Clock) :: ESMF_ClockCreateCopy
```

**ARGUMENTS:**

```fortran
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.

---

### 33.5.6 ESMF_ClockDestroy - Free all resources associated with a Clock

**INTERFACE:**

```fortran
subroutine ESMF_ClockDestroy(clock, rc)
```

**ARGUMENTS:**

```fortran
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.
33.5.7  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, direction, 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
type(ESMF_Direction), intent(out), optional :: direction
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. Increments in ESMF_MODE_FORWARD
and decrements in ESMF_MODE_REVERSE; see "direction" argument below and in ESMF_ClockSet().
[alarmCount]  The number of ESMF_Alarms in the ESMF_Clock’s ESMF_Alarm list.
[direction]   The ESMF_Clock’s time stepping direction. See also ESMF_ClockIsReverse(), an alternative for
convenient use in "if" and "do while" constructs.
[rc]         Return code; equals ESMF_SUCCESS if there are no errors.

33.5.8 ESMF_ClockGetAlarm - Get an Alarm in a Clock’s Alarm list

INTERFACE:

    subroutine ESMF_ClockGetAlarm(clock, name, alarm, rc)

ARGUMENTS:

    type(ESMF_Clock), intent(inout) :: 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.

33.5.9 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.

33.5.10 ESMF_ClockGetNextTime - Calculate a Clock's next time

INTERFACE:

```fortran
subroutine ESMF_ClockGetNextTime(clock, nextTime, timeStep, rc)
```

ARGUMENTS:

- `type(ESMF_Clock), intent(in) :: clock`
- `type(ESMF_Time), intent(out) :: nextTime`
- `type(ESMF_TimeInterval), intent(inout), 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.

33.5.11  ESMF_ClockIsDone - Based on its direction, test if the Clock has reached or exceeded its stop time or start time

INTERFACE:

    function ESMF_ClockIsDone(clock, rc)

RETURN VALUE:

    logical :: ESMF_ClockIsDone

ARGUMENTS:

    type(ESMF_Clock), intent(in) :: clock
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns true if currentTime is greater than or equal to stopTime in ESMF_MODE_FORWARD, or if currentTime is less than or equal to startTime in ESMF_MODE_REVERSE. It returns false otherwise.

The arguments are:

clock  The object instance to check.

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

33.5.12  ESMF_ClockIsReverse - Test if the Clock is in reverse mode

INTERFACE:

    function ESMF_ClockIsReverse(clock, rc)

RETURN VALUE:

    logical :: ESMF_ClockIsReverse

ARGUMENTS:

    type(ESMF_Clock), intent(in) :: clock
    integer, intent(out), optional :: rc
DESCRIPTION:

Returns true if clock is in ESMF_MODE_REVERSE, and false if in ESMF_MODE_FORWARD. Allows convenient use in "if" and "do while" constructs. Alternative to ESMF_ClockGet(...direction=...).
The arguments are:
clock The object instance to check.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

33.5.13 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.

33.5.14 ESMF_ClockIsStopTimeEnabled - Test if the Clock’s stop time is enabled

INTERFACE:

function ESMF_ClockIsStopTimeEnabled(clock, rc)

RETURN VALUE:

logical :: ESMF_ClockIsStopTimeEnabled

ARGUMENTS:

type(ESMF_Clock), intent(in) :: clock
integer, intent(out), optional :: rc

DESCRIPTION:

Returns true if the clock’s stop time is set and enabled, and false otherwise.
The arguments are:
clock The object instance to check.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.
### 33.5.15 ESMF_ClockPrint - Print the contents of a Clock

**INTERFACE:**

```fortran
subroutine ESMF_ClockPrint(clock, options, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_Clock), intent(in) :: clock
class 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.

*Note:* Many `ESMF_<class>Print` methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to `stdout` such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call `flush(6)` as a workaround until this issue is fixed in a future release.

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.
  - "direction" - print the clock’s timestep direction.
  - "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.

---

### 33.5.16 ESMF_ClockSet - Set one or more properties of a Clock

**INTERFACE:**

```fortran
subroutine ESMF_ClockSet(clock, name, timeStep, startTime, stopTime, runDuration, runTimeStepCount, refTime, currTime, advanceCount, direction, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_Clock), intent(inout) :: clock
character (len=*) , intent(in), optional :: name
```
type(ESMF_TimeInterval), intent(inout), optional :: timeStep
type(ESMF_Time), intent(inout), optional :: startTime
type(ESMF_Time), intent(inout), optional :: stopTime
type(ESMF_TimeInterval), intent(inout), optional :: runDuration
integer, intent(in), optional :: runTimeStepCount
type(ESMF_Time), intent(inout), optional :: refTime
type(ESMF_Time), intent(inout), optional :: currTime
integer(ESMF_KIND_I8), intent(in), optional :: advanceCount
type(ESMF_Direction), intent(in), optional :: direction
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, which can be positive or negative. This is used to change a clock’s timestep property for those applications that need variable timesteps. See ESMF_ClockAdvance() below for specifying variable timesteps that are NOT saved as the clock’s internal time step property. See "direction" argument below for behavior with t ESMF_MODE_REVERSE direction.

startTime]  The ESMF_Clock’s starting time. Can be less than or greater than stopTime, depending on a positive or negative timeStep, respectively, and whether a stopTime is specified; see below.

stopTime]  The ESMF_Clock’s stopping time. Can be greater than or less than the startTime, depending on a positive or negative timeStep, respectively. 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. Can be positive or negative, consistent with the timeStep’s sign. Mutually exclusive with stopTime and runTimeStepCount.

runTimeStepCount]  Alternative way to specify ESMF_Clock’s stopping time; stopTime = startTime + (runTimeStepCount * timeStep). stopTime can be before startTime if timeStep is negative. 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 time stepped.

direction]  Sets the clock’s time-stepping direction. If called with ESMF_MODE_REVERSE, sets the clock in "reverse" mode, causing it to timestep back towards its startTime. If called with ESMF_MODE_FORWARD, sets the clock in normal, "forward" mode, causing it to timestep in the direction of its startTime to stopTime. This holds true for negative timestep clocks as well, which are initialized (created) with stopTime < startTime. The default mode is ESMF_MODE_FORWARD, established at ESMF_ClockCreate(). timeStep can also be specified as an argument at the same time, which allows for a change in magnitude and/or sign of the clock’s timeStep. If not specified with ESMF_MODE_REVERSE, the clock’s current timeStep is effectively negated. If timeStep is specified, its sign is used as specified; it is not negated internally. E.g., if the specified timeStep is negative and the clock is placed in ESMF_MODE_REVERSE, subsequent calls to ESMF_ClockAdvance() will cause the clock’s current time to be decremented by the new timeStep’s magnitude.
33.5.17 ESMF_ClockStopTimeDisable - Disable a Clock’s stop time

INTERFACE:

    subroutine ESMF_ClockStopTimeDisable(clock, rc)

ARGUMENTS:

    type(ESMF_Clock), intent(inout) :: clock
    integer, intent(out), optional :: rc

DESCRIPTION:

Disables a ESMF_Clock’s stop time; ESMF_ClockIsStopTime() will always return false, allowing a clock to run past its stopTime. The arguments are:

    clock  The object instance whose stop time to disable.

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

33.5.18 ESMF_ClockStopTimeEnable - Enable an Clock’s stop time

INTERFACE:

    subroutine ESMF_ClockStopTimeEnable(clock, stopTime, rc)

ARGUMENTS:

    type(ESMF_Clock), intent(inout) :: clock
    type(ESMF_Time), intent(in), optional :: stopTime
    integer, intent(out), optional :: rc

DESCRIPTION:

Enables a ESMF_Clock’s stop time, allowing ESMF_ClockIsStopTime() to respect the stopTime. The arguments are:

    clock  The object instance whose stop time to enable.

    stopTime  The stop time to set or reset.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
33.5.19  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.

33.5.20  ESMF_ClockValidate - Validate a Clock’s properties

INTERFACE:

    subroutine ESMF_ClockValidate(clock, options, rc)

ARGUMENTS:

    type(ESMF_Clock), intent(inout) :: clock
    character (len=*), intent(in), optional :: options
    integer, intent(out), optional :: rc

DESCRIPTION:

Checks whether a clock is valid. Must have a valid startTime and timeStep. If clock has a stopTime, its currTime
must be within startTime to stopTime, inclusive; also startTime’s and stopTime’s calendars must be the same.
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.
34 Alarm Class

34.1 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.

34.2 Alarm Options
34.2.1 ESMF_AlarmListType
DESCRIPTION: Specifies the characteristics of Alarms that populate a retrieved Alarm list.
Valid values are:
ESMFALARMLIST_ALL All alarms.
ESMFALARMLIST_NEXTRINGING Alarms that will ring before or at the next timestep.
ESMFALARMLIST_PREVRINGING Alarms that rang at or since the last timestep.
ESMFALARMLIST_RINGING Only ringing alarms.

34.3 Use and Examples
Alarms are used in conjunction with Clocks (see Section 33.1). 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. In ESMF_MODE_REVERSE (see Section 33.1), alarms ring in reverse, i.e., they begin ringing when they originally ended, and end ringing when they originally began. 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.

```fortran
! !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)

34.3.1 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)

34.3.2 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)
34.3.3 Clock Advance and Alarm Processing

This example shows how to advance an ESMF_Clock and process any resulting ringing alarms.

```fortran
! 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
```

34.3.4 Alarm and Clock Destruction

This example shows how to destroy ESMF_Alarms and ESMF_Clocks.

```fortran
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=rc)
```

end program ESMF_AlarmEx
34.4 Restrictions and Future Work

1. **Alarm list allocation factor** The alarm list within a clock is dynamically allocated automatically, 200 alarm references at a time. This constant is defined in both Fortran and C++ with a #define for ease of modification.

2. **Sticky alarm end times in reverse** For sticky alarms, there is an implicit limitation that in order to properly reverse timestep through a ring end time, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call ESMF_AlarmRingerOff(). An error message will be logged when this limitation is not satisfied.

3. **Sticky alarm ring interval in reverse** For repeating sticky alarms, it is currently assumed that the ringInterval is constant, so that only the time of the last call to ESMF_AlarmRingerOff() is saved. In ESMF_MODE_REVERSE, this information is used to turn sticky alarms back on. In a future release, ringIntervals will be allowed to be variable, by saving alarm state at every timestep.

34.5 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 contrast, 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.

34.6 Class API

34.6.1 ESMF_AlarmOperator(==) - Test if Alarm 1 is equal to Alarm 2

**INTERFACE:**
```
interface operator(==)
    if (alarm1 == alarm2) then ... endif
    OR
    result = (alarm1 == alarm2)
end interface
```

**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.
34.6.2 ESMF_AlarmOperator(/=) - Test if Alarm 1 is not equal to Alarm 2

INTERFACE:

```
interface operator(/=)
  if (alarm1 /= alarm2) then ... endif
  result = (alarm1 /= alarm2)
end interface
```

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.

34.6.3 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.
In ESMF_MODE_REVERSE (see Section 33.1), alarms ring in reverse, i.e., they begin ringing when they originally ended, and end ringing when they originally began.

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). There is an implicit limitation that in order to properly reverse timestep through a ring end time in ESMF_MODE_REVERSE, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call ESMF_AlarmRingerOff(). An error message will be logged when this limitation is not satisfied. See also ESMF_AlarmSticky(), ESMF_AlarmNotSticky().

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

34.6.4 ESMF_AlarmCreate - Create a copy of an existing ESMF Alarm

INTERFACE:

! Private name; call using ESMF_AlarmCreate()
defunction ESMF_AlarmCreateCopy(alarm, rc)

RETURN VALUE:

type(ESMF_Alarm) :: ESMF_AlarmCreateCopy
ARGUMENTS:

    type(ESMF_Alarm), intent(inout) :: 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.

34.6.5  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.

34.6.6  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.
34.6.7  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.

34.6.8  ESMF_AlarmGet - Get Alarm properties

INTERFACE:

    subroutine ESMF_AlarmGet(alarm, name, clock, ringTime, prevRingTime, &
                              ringInterval, stopTime, ringDuration, &
                              ringTimeStepCount, timeStepRingingCount, &
                              ringBegin, ringEnd, refTime, ringing, &
                              ringingOnPrevTimeStep, enabled, sticky, rc)

ARGUMENTS:

    type(ESMF_Alarm),            intent(inout) :: 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 :: ringEnd
    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). Increments in ESMF_MODE_FORWARD and decrements in ESMF_MODE_REVERSE; see Section 33.1.
[ringBegin] The time when the alarm began ringing. Used internally for tracking ringDuration (see above). Mutually exclusive with timeStepRingingCount (see above).
[ringEnd] The time when the alarm ended ringing. Used internally for re-ringing alarm in ESMF_MODE_REVERSE.
[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().

34.6.9 ESMF_AlarmIsEnabled - Check if Alarm is enabled

INTERFACE:

function ESMF_AlarmIsEnabled(alarm, rc)

RETURN VALUE:

logical :: ESMF_AlarmIsEnabled

ARGUMENTS:

type(ESMF_Alarm), intent(inout) :: 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.

34.6.10 ESMF_AlarmIsRinging - Check if Alarm is ringing

INTERFACE:

    function ESMF_AlarmIsRinging(alarm, rc)

RETURN VALUE:

    logical :: ESMF_AlarmIsRinging

ARGUMENTS:

    type(ESMF_Alarm), intent(inout) :: 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.

34.6.11 ESMF_AlarmIsSticky - Check if Alarm is sticky

INTERFACE:

    function ESMF_AlarmIsSticky(alarm, rc)

RETURN VALUE:

    logical :: ESMF_AlarmIsSticky

ARGUMENTS:

    type(ESMF_Alarm), intent(inout) :: 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.

34.6.12 ESMF_AlarmNotSticky - Unset an Alarm’s sticky flag

INTERFACE:

```fortran
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.

34.6.13 ESMF_AlarmPrint - Print out an Alarm’s properties

INTERFACE:

```fortran
subroutine ESMF_AlarmPrint(alarm, options, rc)
```

ARGUMENTS:

- type(ESMF_Alarm), intent(inout) :: 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.

Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn't appear in the expected order. If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.

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.
- "ringEnd" - print time when the alarm actually ends 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.

### 34.6.14 ESMF_AlarmRingerOff - Turn off an Alarm

**INTERFACE:**

```fortran
subroutine ESMF_AlarmRingerOff(alarm, rc)
```

**ARGUMENTS:**

```fortran
type(ESMF_Alarm), intent(inout) :: alarm
integer, intent(out), optional :: rc
```

**DESCRIPTION:**

Turn off an ESMF_Alarm; unsets ringing state. For a sticky alarm, this method must be called to turn off its ringing state. This is true for either ESMF_MODE_FORWARD (default) or ESMF_MODE_REVERSE. See Section [33.1](#).

The arguments are:

**alarm** The object instance to turn off.

**[rc]** Return code; equals ESMF_SUCCESS if there are no errors.
34.6.15  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.

34.6.16  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). There is an implicit limitation that in order to properly reverse timestep through a ring end time in ESMF_MODE_REVERSE, that time must have already been traversed in the forward direction! This is due to the fact that the Time Manager cannot predict when user code will call ESMF_AlarmRingerOff(). An error message will be logged when this limitation is not satisfied. See also ESMF_AlarmSticky(), ESMF_AlarmNotSticky().

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

34.6.17 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. There is an implicit limitation that in order to properly reverse timestep through a ring end time in ESMF_MODE_REVERSE, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call ESMF_AlarmRingerOff(). An error message will be logged when this limitation is not satisfied.

The arguments are:

alarm The object instance to be set sticky.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
34.6.18  ESMF_AlarmValidate - Validate an Alarm’s properties

INTERFACE:

    subroutine ESMF_AlarmValidate(alarm, options, rc)

ARGUMENTS:

    type(ESMF_Alarm), intent(inout) :: alarm
    character (len=*), intent(in), optional :: options
    integer, intent(out), optional :: rc

DESCRIPTION:

Performs a validation check on an ESMF_Alarm’s properties. Must have a valid ringTime, set either directly or indirectly via ringInterval. See ESMF_AlarmCreate().

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.

34.6.19  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(inout) :: 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.
34.6.20  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(inout) :: 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.

35  Config Class

35.1  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.

35.1.1  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.

35.2  Use and Examples

35.2.1  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:
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.

### 35.2.2 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.

### 35.2.3 A Quick Overview

### 35.2.4 Common Code Arguments

Common Arguments used in the following code fragments:

- `character(ESMF_MAXSTR) :: fname` ! file name
- `character*20 :: fn1, fn2, fn3`
- `integer :: rc` ! error return code (0 is OK)
- `integer :: i_n`
- `real :: radius`
- `real :: table(7,3)`
- `type(ESMF_Config) :: cf`

### 35.2.5 Creation of a Config

The first step is to create the `ESMF_Config` and load the ASCII resource (rc) file into memory:

```
cf = ESMF_ConfigCreate(rc)
```

---

3See next section for a complete description of parameters for each routine/function
fname = "myResourceFile.rc"
call ESMF_ConfigLoadFile(cf, fname, rc=rc)

### 35.2.6 Retrieval of constants

The next step is to select the label (record) of interest, say

```fortran
    call ESMF_ConfigFindLabel(cf, 'constants:', rc=rc)
```

Two constants, radius and i_n, can be retrieved with the following code fragment:

```fortran
    call ESMF_ConfigGetAttribute(cf, radius, rc=rc) ! results in radius = 3.1415
    call ESMF_ConfigGetAttribute(cf, i_n, rc=rc)   ! results in i_n = 25
```

### 35.2.7 Retrieval of file names

File names can be retrieved with the following code fragment:

```fortran
    call ESMF_ConfigFindLabel(cf, 'my_file_names:', rc=rc)
```

```fortran
    call ESMF_ConfigGetAttribute(cf, fn1, rc=rc) ! results in fn1 = 'jan87.dat'
    call ESMF_ConfigGetAttribute(cf, fn2, rc=rc) ! results in fn2 = 'jan88.dat'
    call ESMF_ConfigGetAttribute(cf, fn3, rc=rc) ! results in fn3 = 'jan89.dat'
```

### 35.2.8 Retrieval of tables

To access tabular data, the user first must use ESMF_ConfigFindLabel() to locate the beginning of the table, e.g.,

```fortran
    call ESMF_ConfigFindLabel(cf, 'my_table_name::', rc=rc)
```

Subsequently, call ESMF_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):

```fortran
    do i = 1, 7
        call ESMF_ConfigNextLine(cf, rc=rc)
        do j = 1, 3
            call ESMF_ConfigGetAttribute(cf, table(i,j), rc=rc)
        enddo
    enddo
```

### 35.2.9 Destruction of a Config

The work with the configuration file cf is finalized by call to ESMF_ConfigDestroy():

```fortran
    call ESMF_ConfigDestroy(cf, rc)
```
35.3 Class API

35.3.1 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.

35.3.2 ESMF_ConfigDestroy - Destroy a Config object

INTERFACE:

    subroutine ESMF_ConfigDestroy( config, rc )

ARGUMENTS:

    type(ESMF_Config) :: 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.

35.3.3 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.

35.3.4 ESMF_ConfigGetAttribute - Get a value

INTERFACE:

```fortran
subroutine ESMF_ConfigGetAttribute( config, <value argument>, &
label, default, rc )
```

ARGUMENTS:

- **type(ESMF_Config), intent(inout) :: config**: config
- **<value argument>, see below for supported values**: 
- **character(len=*), intent(in), optional :: label**: 
- **character(len=*), intent(in), optional :: default**: 
- **integer, intent(out), optional :: rc**: 

DESCRIPTION:

Gets a value from the config object. When the value is a sequence of characters it will be terminated by the first white space.

Supported values for <value argument> are:

- **character(len=*)**, intent(out) :: value
- **real(ESMF_KIND_R4)**, intent(out) :: value
- **real(ESMF_KIND_R8)**, intent(out) :: value
- **integer(ESMF_KIND_I4)**, intent(out) :: value
- **integer(ESMF_KIND_I8)**, intent(out) :: value
- **logical**, intent(out) :: value

The arguments are:

- **config**: Already created ESMF_Config object.
- **<value argument>**: 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.
35.3.5 ESMF_ConfigGetAttribute - Get a list of values

INTERFACE:

    subroutine ESMF_ConfigGetAttribute( config, <value list argument>, &
    count, label, default, rc )

ARGUMENTS:

    type(ESMF_Config), intent(inout) :: config
    <value list argument>, see below for values
    integer, intent(in) :: count
    character(len=*) , intent(in), optional :: label
    character(len=*) , intent(in), optional :: default
    integer, intent(out), optional :: rc

DESCRIPTION:

Gets a list of values from the config object.
Supported values for <value list argument> are:

    real(ESMF_KIND_R4), intent(inout) :: valueList(:)
    real(ESMF_KIND_R8), intent(inout) :: valueList(:)
    integer(ESMF_KIND_I4), intent(inout) :: valueList(:)
    integer(ESMF_KIND_I8), intent(inout) :: valueList(:)
    logical, intent(inout) :: valueList(:)

The arguments are:

config  Already created ESMF_Config object.

<value list argument>  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.

35.3.6 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.

35.3.7 ESMF_ConfigGetDim - Get table sizes

INTERFACE:

```fortran
subroutine ESMF_ConfigGetDim( config, lineCount, columnCount, label, 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.

35.3.8 ESMF_ConfigGetLen - Get the length of the line in words

INTERFACE:

```fortran
integer function ESMF_ConfigGetLen( config, label, rc )
```

403
ARGUMENTS:

\[
\begin{align*}
\text{type}(\text{ESMF\_Config}), \text{intent(inout)} &: \text{ config} \\
\text{character}(\text{len=*}), \text{intent(in), optional} &: \text{ label} \\
\text{integer, intent(out), optional} &: \text{ rc}
\end{align*}
\]

DESCRIPTION:

Gets the length of the line in words by counting words disregarding types. Returns the word count as an integer. The arguments are:

\text{config} Already created \text{ESMF\_Config} object.

\text{[label]} Identifying label. If not specified, use the current line.

\text{[rc]} Return code; equals \text{ESMF\_SUCCESS} if there are no errors.

35.3.9 \text{ESMF\_ConfigLoadFile} - Load resource file into memory

INTERFACE:

\[
\text{subroutine ESMF\_ConfigLoadFile( config, filename, delay out, unique, rc )}
\]

ARGUMENTS:

\[
\begin{align*}
\text{type}(\text{ESMF\_Config}), \text{intent(inout)} &: \text{ config} \\
\text{character}(\text{len=*}), \text{intent(in)} &: \text{ filename} \\
\text{type}(\text{ESMF\_DELayout}), \text{intent(in), optional} &: \text{ delay out} \\
\text{logical, intent(in), optional} &: \text{ unique} \\
\text{integer, intent(out), optional} &: \text{ rc}
\end{align*}
\]

DESCRIPTION:

Resource file with \text{filename} is loaded into memory. The arguments are:

\text{config} Already created \text{ESMF\_Config} object.

\text{filename} Configuration file name.

[\text{delay out}] \text{ESMF\_DELayout} associated with this \text{config} object.

[\text{unique}] If specified as true, uniqueness of labels are checked and error code set if duplicates found.

[\text{rc}] Return code; equals \text{ESMF\_SUCCESS} if there are no errors.

35.3.10 \text{ESMF\_ConfigNextLine} - Find next line

INTERFACE:

\[
\text{subroutine ESMF\_ConfigNextLine( config, tableEnd, rc)}
\]

ARGUMENTS:
DESCRIPTION:

Selects the next line (for tables).
The arguments are:

config Already created ESMF_Config object.

[tableEnd] If specified as TRUE, end of table mark (::) is checked.

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

35.3.11 ESMF_ConfigSetAttribute - Set a value

INTERFACE:

    subroutine ESMF_ConfigSetAttribute( config, <value argument>, &
                                         label, rc )

ARGUMENTS:

    type(ESMF_Config), intent(inout) :: config
    <value argument>, see below for supported values
    character(len=*), intent(in), optional :: label
    integer, intent(out), optional :: rc

DESCRIPTION:

Sets a value in the config object.
Supported values for <value argument> are:

    integer(ESMF_KIND_I4), intent(in) :: value

The arguments are:

config Already created ESMF_Config object.

<value argument> Value to set.

[label] Identifying attribute label.

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

35.3.12 ESMF_ConfigValidate - Validate a Config object

INTERFACE:

    subroutine ESMF_ConfigValidate(config, options, rc)

ARGUMENTS:
type(ESMF_Config), intent(inout) :: config
character(len=*), intent(in), optional :: options
integer, intent(out), optional :: rc

DESCRIPTION:
Checks whether a config object is valid.
The arguments are:
config ESMF_Config object to be validated.
[options] If none specified: simply check that the buffer is not full and the pointers are within range. "unusedAttributes" - Report to the default logfile all attributes not retrieved via a call to ESMF_ConfigGetAttribute() or ESMF_ConfigGetChar(). The attribute name (label) will be logged via ESMF_LogErr with the WARNING log message type. For an array-valued attribute, retrieving at least one value via ESMF_ConfigGetAttribute() or ESMF_ConfigGetChar() constitutes being "used."
[rc] Return code; equals ESMF_SUCCESS if there are no errors. Equals ESMF_RC_ATTR_UNUSED if any unused attributes are found with option "unusedAttributes" above.

36 LogErr Class

36.1 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. Most LogErr methods take a Log as an optional argument and apply to the default Log when another Log is not specified. A set of standard return codes and associated messages are provided for error handling.
LogErr provides capabilities to store message entries in a buffer, which is flushed to a file, either when the buffer is full, or when the user calls an ESMF_LogFlush() method. Currently, the default is for the Log to flush after every ten entries. This can easily be changed by using the ESMF_LogSet() method and setting the maxElements property to another value. The ESMF_LogFlush() method is automatically called when the program exits by any means (program completion, halt on error, or when the Log is closed).
The user has the capability to halt the program on an error or on a warning by using the ESMF_LogSet() method with the halt property. When the halt property is set to ESMF_LOG_HALTWARNING, the program will stop on any and all warning or errors. When the halt property is set to ESMF_LOG_HALTERROR, the program will only halt on errors. Lastly, the user can choose to never halt by setting the halt property to ESMF_LOG_HALTNEVER; this is the default.
LogErr will automatically put the PET number into the Log. Also, the user can either specify ESMF_LOG_SINGLE which writes all the entries to a single Log or ESMF_LOG_MULTI which writes entries to multiple Logs according to the PET number. To distinguish Logs from each other when using ESMF_LOG_MULTI, the PET number (in the format PETx.) will be prepended to the file name where x is the PET number.
Opening multiple log files and writing log messages from all the processors may affect the application performance while running on a large number of processors. For that reason, ESMF_LOG_NONE is provided to switch off the LogErr capability. All the LogErr methods have no effect in the ESMF_LOG_NONE mode.
Other options that are planned for LogErr are to adjust the verbosity of output, and to optionally write to stdout instead of file(s).

36.2 LogErr Options
36.2.1 ESMF_HaltType
DESCRIPTION:
Specifies when to halt - e.g., never, warning, error.
Valid values are:
ESMF_LOG_HALTNEVER Never halt.
ESMF_LOG_HALTWARN  Halt on a warning.

ESMF_LOG_HALTERROR  Halt on an error.

36.2.2 ESMF_MsgType

DESCRIPTION: Specifies what sort of message - e.g., info, warning, 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.

36.2.3 ESMF_LogType

DESCRIPTION: Specifies single, multi or no Log. Valid values are:

ESMF_LOG_SINGLE  Log is single Log.

ESMF_LOG_MULTI  Log is multi Log.

ESMF_LOG_NONE  There is no Log.

36.3 Use and Examples

By default ESMF_Initialize() opens a default Log in ESMF_LOG_MULTI mode. ESMF handles the initialization and finalization of the default Log so the user can immediately start using it. If additional Log objects are desired, they must be explicitly created or opened using ESMF_LogOpen(). ESMF_LogOpen() requires a Log object and filename argument. Additionally, the user can specify single or multi Logs by setting the logtype property to ESMF_LOG_SINGLE or ESMF_LOG_MULTI. This is useful as the PET numbers are automatically added to the Log entries. A single Log will put all entries, regardless of PET number, into a single log while a multi Log will create multiple Logs with the PET number prepended to the filename and all entries will be written to their corresponding Log by their PET number.

By default, the Log file is not truncated at the start of a new run; it just gets appended each time. Future functionality would 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 Fortran “inquire” method. The process repeats until a free unit number is found or when the unit number reaches ESMF_LOG_UPPER in which case an error is returned. As a result, the user should always check for free numbers using Fortran’s “inquire” to prevent potential unit number conflicts. In the 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 partially 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. The default is to flush after every ten entries because maxElements is initialized to ten (which means the buffer reaches its full state after every ten writes and then flushes).

For every ESMF_LogWrite(), a time and date stamp is prepended to the Log entry. The time is given in microsecond precision. The user can call other methods to write to the Log. In every case, all methods eventually make a call implicitly to ESMF_LogWrite() even though the user may never explicitly call it.

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_FILENAME must be
placed at the beginning of a file 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.

An example of Log output is given below running with logtype property set to ESMF_LOG_MULTI (default) using the default Log:

(Log file PET0.ESMF_LogFile)

20041105 163418.472210 INFO PET0 Running with ESMF Version 2.2.1

(Log file PET1.ESMF_LogFile)

20041105 163419.186153 ERROR PET1 ESMF_Field.F90 812
ESMF_FieldGet No Grid or Bad Grid attached to Field

The first entry shows date and time stamp. The time is given in microsecond precision. The next item shown is the type of message (INFO in this case). Next, the PET number is added. Lastly, the content is written.

The second entry shows something slightly different. In this case, we have an ERROR. The method name (ESMF_Field.F90) is automatically provided from the cpp macros as well as the line number (812). Then the content of the message is written.

When done writing messages, the default Log is closed by calling ESMF_LogFinalize() or ESMF_LogClose() for user created Logs. Both methods 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
! File define
#define ESMF_FILENAME "ESMF_LogErrEx.F90"
! Method define
#define ESMF_METHOD "program ESMF_LogErrEx"
#include "ESMF_LogMacros.inc"

! ESDF Framework module
use ESMF_Mod
implicit none

! return variables
integer :: r1, r2, r3, rcToTest, allocRcToTest
type(ESMF_LOG) :: alog !a log object that is not the default log
type(ESMF_LogType) :: defaultLogtype
type(ESMF_Time) :: time
integer, pointer :: intptr(:)

36.3.1 Default Log

This example shows how to use the default Log. This example does not use cpp macros but does use multi Logs. A separate Log will be created for each PET.

! Initialize ESMF to initialize the default Log
call ESMF_Initialize(rc=r1, defaultLogtype=ESMF_LOG_MULTI)
! LogWrite
call ESMF_LogWrite("Log Write 2", ESMF_LOG_INFO, rc=rc2)

! LogMsgSetError
call ESMF_LogMsgSetError(ESMF_FAILURE, "Convergence failure", &
rcToReturn=rc2)

! LogMsgFoundError
call ESMF_TimeSet(time, calendarType=ESMF_CAL_NOCALENDAR)
call ESMF_TimeSyncToRealTime(time, rcToTest)
if (ESMF_LogMsgFoundError(rcToTest, "getting wall clock time", &
rcToReturn=rc2)) then
! Error getting time. The previous call will have printed the error
! already into the log file. Add any additional error handling here.
! (This call is expected to provoke an error from the Time Manager.)
endif

! LogMsgFoundAllocError
allocate(intptr(10), stat=allocRcToTest)
if (ESMF_LogMsgFoundAllocError(allocRcToTest, "integer array", &
rcToReturn=rc2)) then
! Error during allocation. The previous call will have logged already
! an error message into the log.
endif
dedallocate(intptr)

36.3.2 User Created Log

This example shows how to use a user created Log. This example uses cpp macros.

! Open a Log named "Testlog.txt" associated with alog.
call ESMF_LogOpen(alog, "TestLog.txt", rc=rc1)

! LogWrite; ESMF_CONTEXT expands into __LINE__,ESMF_FILENAME,ESMF_METHOD
call ESMF_LogWrite("Log Write 2", ESMF_LOG_INFO, ESMF_CONTEXT, &
log=alog, rc=rc2)

! LogMsgSetError; ESMF_CONTEXT expands into
! __LINE__,ESMF_FILENAME,ESMF_METHOD
call ESMF_LogMsgSetError(ESMF_FAILURE, "Interpolation Failure", &
ESMF_CONTEXT, rcToReturn=rc2, log=alog)

36.3.3 Get and Set

This example shows how to use Get and Set routines, on both the default Log and the user created Log from the
previous examples.

! This is an example showing a query of the default Log. Please note that
! no Log is passed in the argument list, so the default Log will be used.
call ESMF_LogGet(logtype=defaultLogtype, rc=rc3)
! This is an example setting a property of a Log that is not the default. ! It was opened in a previous example, and the handle for it must be ! passed in the argument list.
call ESMF_LogSet(log=alog, halt=ESMF_LOG_HALTERROr, rc=rc2)

! Close the user log.
call ESMF_LogClose(alog, rc3)

! Finalize ESMF to close the default log
call ESMF_Finalize(rc=rc1)

36.4 Restrictions and Future Work

1. **Line, file and method are only available when using the C preprocessor** Message writing methods are expanded using the ESMF macro ESMF_CONTEXT that adds the predefined symbolic constants __LINE__ and __FILE__ (or the ESMF constant ESMF_FILENAME if defined) and the ESMF constant ESMF_METHOD to the argument list. Using these constants, we can associate a file name, line number and method name with the message. If the CPP preprocessor is not used, this expansion will not be done and hence the ESMF macro ESMF_CONTEXT can not be used, leaving the file name, line number and method out of the Log text.

2. **Get and set methods are partially implemented.** Currently, the ESMF_LogGet() and ESMF_LogSet() methods are partially implemented.

3. **Log only appends entries.** All writing to the Log is appended rather than overwriting the Log. Future enhancements include the option to either append to an existing Log or overwrite the existing Log.

4. **Avoiding conflicts with the default Log.** The private methods ESMF_LogInitialize() and ESMF_LogFinalize() are called during ESMF_Initialize() and ESMF_Finalize() respectively, 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.

5. **ESMF_LOG_SINGLE does not work properly.** When the ESMF_LogType is set to ESMF_LOG_SINGLE, different system may behave differently. The log messages from some processors may be lost or overwritten by other processors. Users are advised not to use this mode. The MPI-based I/O will be implemented to fix the problem in the future release.

36.5 Design and Implementation Notes

1. The Log class was implemented in Fortran and uses the Fortran I/O libraries when the class methods are called from Fortran. The C/C++ Log methods use the Fortran I/O library by calling utility functions that are written in Fortran. These utility functions call the standard Fortran write, open and close functions. At initialization an ESMF_LOG is created. The ESMF_LOG stores information for a specific Log file. When working with more than one Log file, multiple ESMF_LOG’s are required (one ESMF_LOG for each Log file). For each Log, a handle is returned through the ESMF_LogInitialize method for the default log or ESMF_LogOpen for a user created log. The user can specify single or multi logs by setting the logtype property in the ESMF_LogInitialize or ESMF_Open method to ESMF_LOG_SINGLE or ESMF_LOG_MULTI. Similarly, the user can set the defaultlogtype property for the default Log with the ESMF_Initialize method call. The logtype is useful as the PET numbers are automatically added to the log entries. A single log will put all entries, regardless of PET number, into a single log while a multi log will create multiple logs with the PET number prepended to the filename and all entries will be written to their corresponding log by their PET number.

The properties for a Log are set with the ESMF_LogSet() method and retrieved with the ESMF_LogGet() method.
Additionally, buffering is enabled. Buffering allows ESMF to manage output data streams in a desired way. Writing to the buffer is transparent to the user because all the Log entries are handled automatically by the ESMF_LogWrite() method. All the user has to do is specify the buffer size (the default is ten) by setting the maxElements property. Every time the ESMF_LogWrite() method is called, a LogEntry element is populated with the ESMF_LogWrite() information. 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, that is maxElements=1 or flushImmediately=ESMF_TRUE, the ESMF_LogWrite() method will immediately write to the Log file(s).

### 36.6 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.

![UML Diagram](image-url)

### 36.7 Class API

#### 36.7.1 ESMF_LogClose - Close Log file(s)

**INTERFACE:**

```fortran
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.
36.7.2 ESMF_LogFlush - Flushes the Log file(s)

**INTERFACE:**

```fortran
subroutine ESMF_LogFlush(log,rc)
ARGUMENTS:
    type(ESMF_Log), target,optional :: log
    integer, intent(out),optional :: rc
```

**DESCRIPTION:**

This subroutine flushes the ESMF_Log buffer to its associated file.
The arguments are:

- [log] An optional ESMF_Log object that can be used instead of the default Log.
- [rc] Return code; equals ESMF_SUCCESS if there are no errors.

36.7.3 ESMF_LogFoundAllocError - Check Fortran status for allocation error

**INTERFACE:**

```fortran
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(inout),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 error.
- [log] An optional ESMF_Log object that can be used instead of the default Log.
36.7.4 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(inout), target, 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.

36.7.5 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(inout), 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 error.

[log]  An optional ESMF_Log object that can be used instead of the default Log.

36.7.6  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(inout), target, 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 is 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.

36.7.7  ESMF_LogMsgSetError - Set ESMF return code for error and write msg

INTERFACE:

subroutine ESMF_LogMsgSetError(rcValue, msg, line, file, method, &
rcToReturn, log)

ARGUMENTS:

integer, intent(in) :: rcValue
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(inout), target, optional :: log

DESCRIPTION:

This subroutine sets the rcToReturn value to rcValue if rcToReturn is present and writes this error code to the ESMF_Log if an error is generated. A predefined error message will added to the ESMF_Log along with a user added msg, line, file and method.

The arguments are:

rcValue  rc value for set

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 rcValue value to rcToReturn. 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.
36.7.8  ESMF_LogOpen - Open Log file(s)

INTERFACE:

    subroutine ESMF_LogOpen(log, filename, logtype, rc)

ARGUMENTS:

    type(ESMF_Log) :: log
    character(len=*) :: filename
    type(ESMF_LogType), intent(in),optional :: logtype
    integer, intent(out),optional :: rc

DESCRIPTION:

This routine opens a file with filename and associates it with the ESMF_Log. This is only used when the user does not want to use the default Log.

The arguments are:


[filename]  Name of file. Maximum length 26 characters to allow for the PET number to be added and keep the total file name length under 32 characters.

[logtype]  Set the logtype. See section 36.2.3 for a list of valid options. If not specified, defaults to ESMF_LOG_MULTI.

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

36.7.9  ESMF_LogSet - Set Log parameters

INTERFACE:

    subroutine ESMF_LogSet(log,verbose,flush,rootOnly,halt, &
                          stream,maxElements,errorMask,rc)

ARGUMENTS:

    type(ESMF_Log), target,optional :: log
    type(ESMF_Logical), intent(in),optional :: verbose
    type(ESMF_Logical), intent(in),optional :: flush
    type(ESMF_Logical), intent(in),optional :: rootOnly
    type(ESMF_HaltType), intent(in),optional :: halt
    integer, intent(in),optional :: stream
    integer, intent(in),optional :: maxElements
    integer, intent(in),optional :: errorMask(:)
    integer, intent(out),optional :: rc

DESCRIPTION:

This subroutine sets the properties for the Log object.

The arguments are:

[log]  An optional ESMF_Log object that can be used instead of the default Log.

[verbose]  Verbose flag.
[rootOnly] Root only flag.

[halt] Halt definition, with the following valid values:

ESMF_LOG_HALTWARNING;
ESMF_LOG_HALTERROR;
ESMF_LOG_HALTNEVER.

[stream] The type of stream, with the following valid values and meanings:

0 free;
1 preordered.

[maxElements] Maximum number of elements in the Log.

[errorMask] List of error codes that will not be logged as errors.

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

36.7.10 ESMF_LogWrite - Write to Log file(s)

INTERFACE:

recursive subroutine ESMF_LogWrite(msg,MsgType,line,file,method,log,rc)

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),target,optional :: log

integer, intent(out),optional :: rc

DESCRIPTION:

This subroutine 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 36.2.2 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.

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

37.1 Description

The DELayout class provides an additional layer of abstraction on top of the Virtual Machine (VM) layer. DELayout does this by introducing DEs (Decomposition Elements) as logical resource units. The DELayout object keeps track of the relationship between its DEs and the resources of the associated VM object.

The relationship between DEs and VM resources (PETs (Persistent Execution Threads) and VASs (Virtual Address Spaces)) contained in a DELayout object is defined during its creation and cannot be changed thereafter. There are, however, a number of hint and specification arguments that can be used to shape the DELayout during its creation. Contrary to the number of PETs and VASs contained in a VM object which are fixed by the available resources the number of DEs contained in a DELayout can be chosen freely to best match the computational problem or other design criteria. Creating a DELayout with less DEs than there are PETs in the associated VM object can be used to share resources between decomposed objects within an ESMF component. Creating a DELayout with more DEs than there are PETs in the associated VM object can be used to evenly partition the computation over the available resources. The simplest case, however, is where the DELayout contains the same number of DEs as there are PETs in the associated VM context. In this case the DELayout may be used to re-label the hardware and operating system resources held by the VM. For instance, it is possible to order the resources so that specific DEs have best available communication paths. The DELayout will map the DEs to the PETs of the VM according to the resource details provided by the VM instance.

Furthermore, general DE to PET mapping can be used to offer computational resources with finer granularity than the VM does. The DELayout can be queried for computational and communication capacities of DEs and DE pairs, respectively. This information can be used to best utilize the DE resources when partitioning the computational problem. In combination with other ESMF classes general DE to PET mapping can be used to realize cache blocking, communication hiding and dynamic load balancing.

Finally, the DELayout layer offers primitives that allow a work queue style dynamic load balancing between DEs.

37.2 Use and Examples

The following examples demonstrate how to create, use and destroy DELayout objects.

37.2.1 Default DELayout

Without specifying any of the optional parameters the created ESMF_DELayout defaults into having as many DEs as there are PETs in the associated VM object. Consequently the resulting DELayout describes a simple 1-to-1 DE to PET mapping.

```
delayout = ESMF_DELayoutCreate(rc=rc)
```

The default DE to PET mapping is simply:

```
DE 0  ->  PET 0
DE 1  ->  PET 1
...
```

DELayout objects that are not used any longer should be destroyed.

```
call ESMF_DELayoutDestroy(delayout, rc=rc)
```

The optional vm argument can be provided to DELayoutCreate() to lower the method’s overhead by the amount it takes to determine the current VM.

```
delayout = ESMF_DELayoutCreate(vm=vm, rc=rc)
```
By default all PETs of the associated VM will be considered. However, if the optional argument petList is present
DEs will only be mapped against the PETs contained in the list. When the following example is executed on four
PETs it creates a DELayout with four DEs by default that are mapped to the provided PETs in their given order. It is
erroneous to specify PETs that are not part of the VM context on which the DELayout is defined.

```c
delayout = ESMF_DELayoutCreate(petList=\( \langle i, i=\text{petCount}-1, 1, -1 \rangle \), rc=rc)
```

Once the end of the petList has been reached the DE to PET mapping continues from the beginning of the list. For a 4
PET VM the above created DELayout will end up with the following DE to PET mapping:

```
DE 0  -> PET 3
DE 1  -> PET 2
DE 2  -> PET 1
DE 2  -> PET 3
```

### 37.2.2 DELayout with specified number of DEs

The deCount argument can be used to specify the number of DEs. In this example a DELayout is created that
contains four times as many DEs as there are PETs in the VM.

```c
delayout = ESMF_DELayoutCreate(deCount=4*petCount, rc=rc)
```

Cyclic DE to PET mapping is the default. For 4 PETs this means:

```
DE 0, 4, 8, 12 -> PET 0
DE 1, 5, 9, 13 -> PET 1
DE 2, 6, 10, 14 -> PET 2
DE 3, 7, 11, 15 -> PET 3
```

The default DE to PET mapping can be overridden by providing the deGrouping argument. This argument provides
a positive integer group number for each DE in the DELayout. All of the DEs of a group will be mapped against the
same PET. The actual group index is arbitrary (but must be positive) and its value is of no consequence.

```c
delayout = ESMF_DELayoutCreate(deCount=4*petCount, &
    deGrouping=\( \langle i/4, i=0, 4*\text{petCount}-1 \rangle \), rc=rc)
```

This will achieve blocked DE to PET mapping. For 4 PETs this means:

```
DE 0, 1, 2, 3  -> PET 0
DE 4, 5, 6, 7  -> PET 1
DE 8, 9, 10, 11 -> PET 2
DE 12, 13, 14, 15 -> PET 3
```

### 37.2.3 DELayout with computational and communication weights

The quality of the partitioning expressed by the DE to PET mapping depends on the amount and quality of information
provided during DELayout creation. In the following example the compWeights argument is used to specify relative
computational weights for all DEs and communication weights for DE pairs are provided by the commWeights
argument. The example assumes four DEs.
allocate(compWeights(4))
allocate(commWeights(4, 4))
! setup compWeights and commWeights according to computational problem
delayout = ESMF_DELayoutCreate(deCount=4, compWeights=compWeights, &
    commWeights=commWeights, rc=rc)
deallocate(compWeights, commWeights)

The resulting DE to PET mapping depends on the specifics of the VM object and the provided compWeights and
commWeights arrays.

37.2.4 DELayout from petMap

Full control over the DE to PET mapping is provided via the petMap argument. This example maps the DEs to PETs
in reverse order. In the 4 PET case this will result in the following mapping:

DE 0 -> PET 3
DE 1 -> PET 1
DE 2 -> PET 2
DE 3 -> PET 0

delayout = ESMF_DELayoutCreate(petMap=((i,i=petCount-1,0,-1)/), rc=rc)

37.2.5 DELayout from petMap with multiple DEs per PET

The petMap argument gives full control over DE to PET mapping. The following example runs on 4 or more PETs
maps DEs to PETs according to the following table:

DE 0 -> PET 3
DE 1 -> PET 3
DE 2 -> PET 1
DE 3 -> PET 0
DE 4 -> PET 2
DE 5 -> PET 1
DE 6 -> PET 3
DE 7 -> PET 1

delayout = ESMF_DELayoutCreate(petMap=(/3, 3, 1, 0, 2, 1, 3, 1/), rc=rc)

37.2.6 Working with a DELayout - simple 1-to-1 DE to PET mapping

The simplest case is a DELayout with as many DEs as PETs where each DE is against a separate PET. This of course
implies that the number of DEs equals the number of PETs. This special 1-to-1 DE to PET mapping is very common
and many codes assume this mapping. The following example code shows how a DELayout can be queried about its
mapping.

delayout = ESMF_DELayoutCreate(rc=rc)
call ESMF_DELayoutGet(delayout, oneToOneFlag=oneToOneFlag, rc=rc)
if (rc /= ESMF_SUCCESS) finalrc=rc
if (oneToOneFlag == ESMF_FALSE) then
    ! handle the unexpected case of general DE to PET mapping
37.2.7 Working with a DELayout - general DE to PET mapping

In general a DELayout may describe a DE to PET mapping that is not 1-to-1. The following example shows how code can be written in a general form that will work on all PETs for DELayouts with general or 1-to-1 DE to PET mapping.

```fortran
37.2.8 Work queue dynamic load balancing

The DELayout API includes two calls that can be used to easily implement work queue dynamic load balancing. The work load is broken up into DEs (more than there are PETs) and processed by the PETs. Load balancing is only possible for ESMF multi-threaded VMs and requires that DEs are pinned to VASs instead of the PETs (default). The following example will run for any VM and DELayout, however, load balancing will only occur under the mentioned conditions.

```
37.3 Restrictions and Future Work

37.4 Design and Implementation Notes

The DELayout class is a light weight object. It stores the DE to PET and VAS mapping for all DEs within all PET instances and a list of local DEs for each PET instance. The DELayout does not store the computational and communication weights optionally provided as arguments to the create method. These hints are only used during create while they are available in user owned arrays.

37.5 Class API

37.5.1 ESMF_DELayoutCreate - Create DELayout object

INTERFACE:

function ESMF_DELayoutCreateDefault(deCount, deGrouping, dePinFlag, petList, &vm, rc)
ARGUMENTS:

integer, intent(in), optional :: deCount
integer, target, intent(in), optional :: deGrouping(:)
type(ESMF_DePinFlag), intent(in), optional :: dePinFlag
integer, target, intent(in), optional :: petList(:)
type(ESMF_VM), intent(in), optional :: vm
integer, intent(out),optional :: rc

RETURN VALUE:

type (ESMF_DELayout) :: ESMF_DELayoutCreateDefault

DESCRIPTION:

Create an ESMF_DELayout object on the basis of optionally provided restrictions. By default a DELayout with deCount equal to petCount will be created, each DE mapped to a single PET. However, the number of DEs as well grouping of DEs and PETs can be specified via the optional arguments.

The arguments are:

[deCount] Number of DEs to be provided by the created DELayout. By default the number of DEs equals the number of PETs in the associated VM context. Specifying a deCount smaller than the number of PETs will result in unassociated PETs. This may be used to share VM resources between DELayouts within the same ESMF component. Specifying a deCount greater than the number of PETs will result in multiple DE to PET mapping.

[deGrouping] This optional argument must be of size deCount. Its content assigns a DE group index to each DE of the DELayout. A group index of -1 indicates that the associated DE isn’t member of any particular group. The significance of DE groups is that all the DEs belonging to a certain group will be mapped against the same PET. This does not, however, mean that DEs belonging to different DE groups must be mapped to different PETs.

[dePinFlag] This flag specifies which type of resource DEs are pinned to. The default is to pin DEs to PETs. Alternatively it is also possible to pin DEs to VASs. See section ?? for a list of valid pinning options.
**petList**  List specifying PETs to be used by this DELayout. This can be used to control the PET overlap between DELayouts within the same ESMF component. It is erroneous to specify PETs that are not within the provided VM context. The default is to include all the PETs of the VM.

**vm**  Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method’s overhead.

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

### 37.5.2 ESMF_DELayoutCreate - Create DELayout from petMap

**INTERFACE:**

```fortran
! Private name; call using ESMF_DELayoutCreate()
function ESMF_DELayoutCreateFromPetMap(petMap, dePinFlag, vm, rc)
ARGUMENTS:
  integer, intent(in) :: petMap(:)
  type(ESMF_DePinFlag), intent(in), optional :: dePinFlag
  type(ESMF_VM), intent(in), optional :: vm
  integer, intent(out),optional :: rc
RETURN VALUE:
  type(ESMF_DELayout) :: ESMF_DELayoutCreateFromPetMap
DESCRIPTION:
Create an ESMF_DELayout with exactly specified DE to PET mapping.
This ESMF method must be called in unison by all PETs of the VM. Calling this method from a PET not part of the VM or not calling it from a PET that is part of the VM will result in undefined behavior. ESMF does not guard against violation of the unison requirement. The call is not collective, there is no communication between PETs.
The arguments are:

- **petMap**  List specifying the DE-to-PET mapping. The list elements correspond to DE 0, 1, 2, ... and map against the specified PET of the VM context. The size of the petMap argument determines the number of DEs in the created DELayout. It is erroneous to specify a PET identifier that lies outside the VM context.

- **dePinFlag**  This flag specifies which type of resource DEs are pinned to. The default is to pin DEs to PETs. Alternatively it is also possible to pin DEs to VASs. See section ?? for a list of valid pinning options.

- **vm**  Optional ESMF_VM object. The VM of the current context is the typical and default value.

- **rc**  Return code; equals ESMF_SUCCESS if there are no errors.

### 37.5.3 ESMF_DELayoutCreate - Create DELayout with weight hints

**INTERFACE:**

```fortran
! Private name; call using ESMF_DELayoutCreate()
function ESMF_DELayoutCreateHintWeights(deCount, compWeights, commWeights, &
  deGrouping, dePinFlag, petList, vm, rc)
```
ARGUMENTS:

- integer, intent(in), optional :: deCount
- integer, intent(in) :: compWeights(:)
- integer, intent(in) :: commWeights(:, :)
- integer, target, intent(in), optional :: deGrouping(:)
- type(ESMF_DePinFlag), intent(in), optional :: dePinFlag
- integer, target, intent(in), optional :: petList(:)
- type(ESMF_VM), intent(in), optional :: vm
- integer, intent(out), optional :: rc

RETURN VALUE:

- type(ESMF_DELayout) :: ESMF_DELayoutCreateHintWeights

DESCRIPTION:

Create an ESMF_DELayout on the basis of computational and communication weights. In addition this call provides control over the number of DEs, DE domains, DE pinning and the PETs to map against.

The arguments are:

- **deCount** Number of DEs to be provided by the created DELayout. By default the number of DEs equals the number of PETs in the associated VM context. Specifying a `deCount` smaller than the number of PETs will result in unassociated PETs. This may be used to share VM resources between DELayouts within the same ESMF component. Specifying a `deCount` greater than the number of PETs will result in multiple DE to PET mapping.

- **compWeights** This argument provides the computational weight hint. The `compWeights` list must contain at least `deCount` elements and specifies a relative measure of the computational weight for each DE in form of an integer number. The weights are a relative measure and only meaningful when compared to weights of the same DELayout.

- **commWeights** This argument provides the communication weight hint. `commWeights` is a 2D array and must contain at least `deCount` elements in each dimension. The element indices correspond to the DEs of the DELayout and each element specifies a relative communication weight for a DE pair. The `commWeights` matrix must be symmetric and diagonal elements are ignored. The weights are a relative measure and only meaningful when compared to weights of the same DELayout.

- **deStride** This optional argument can be used to specify DE domains. The argument holds two elements: `(interStride, intraStride)` which are used to form DE subsets from the full set `{ 0, 1, ..., deCount-1 }` of DEs. The elements of the kth subset are `{ k * interStride + i * intraStride }` where i and k start at 0. DEs within each subset are mapped against the same PET, causing multiple DE to PET mapping. The default is to generate homogeneously sized, blocked DE domains. If the number of DEs is a multiple of the number of PETs the default can be expressed as `deStride = (/deCount/petCount, 1/)`.

- **dePinFlag** This flag specifies which type of resource DEs are pinned to. The default is to pin DEs to PETs. Alternatively it is also possible to pin DEs to VASs. See section ?? for a list of valid pinning options.

- **petList** List specifying PETs to be used by this DELayout. This can be used to control the PET overlap between DELayouts within the same ESMF component. It is erroneous to specify PETs that are not within the provided VM context. The default is to include all the PETs of the VM.

- **vm** Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method’s overhead.

- **rc** Return code; equals `ESMF_SUCCESS` if there are no errors.
37.5.4 ESMF_DELayoutDestroy - Destroy DELayout object

INTERFACE:

    subroutine ESMF_DELayoutDestroy(delayout, rc)

ARGUMENTS:

    type(ESMF_DELayout), intent(inout) :: 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.

37.5.5 ESMF_DELayoutGet - Get DELayout internals

INTERFACE:

    subroutine ESMF_DELayoutGet(delayout, vm, deCount, petMap, vasMap, &
                                compCapacity, commCapacity, oneToOneFlag, dePinFlag, &
                                localDeCount, localDeList, vasLocalDeCount, vasLocalDeList, rc)

ARGUMENTS:

    type(ESMF_DELayout), intent(in) :: delayout
    type(ESMF_VM), intent(out), optional :: vm
    integer, intent(out), optional :: deCount
    integer, target, intent(out), optional :: petMap(:)
    integer, target, intent(out), optional :: vasMap(:)
    integer, target, intent(out), optional :: compCapacity(:)
    integer, target, intent(out), optional :: commCapacity(:,:)
    type(ESMF_Logical), intent(out), optional :: oneToOneFlag
    type(ESMF_DePinFlag), intent(out), optional :: dePinFlag
    integer, intent(out), optional :: localDeCount
    integer, target, intent(out), optional :: localDeList(:)
    integer, target, intent(out), optional :: vasLocalDeCount
    integer, target, intent(out), optional :: vasLocalDeList(:)
    integer, intent(out), optional :: rc

DESCRIPTION:

Access to DELayout information.
The arguments are:

delayout  Queried ESMF_DELayout object.

[vm]  Upon return this holds the ESMF_VM object on which the delayout is defined.
Upon return this holds the total number of DEs.

Upon return this holds the list of PETs against which the DEs are mapped. The petMap argument must at least be of size deCount.

Upon return this holds the list of VASs against which the DEs are mapped. The vasMap argument must at least be of size deCount.

Upon return this holds a relative measure of the computational capacity for each DE. The compCapacity argument must at least be of size deCount.

Upon return this holds a relative measure of the communication capacity for each pair of DEs. The commCapacity argument is a 2D array where each dimension must at least be of size deCount.

Upon return this holds ESMF_TRUE if the specified ESMF_DELayout describes a 1-to-1 mapping between DEs and PETs, ESMF_FALSE otherwise.

Upon return this flag will indicate the type of DE pinning. See section ?? for a list of valid pinning options.

Upon return this holds the number of DEs associated with the local PET.

Upon return this holds the list of DEs associated with the local PET. The provided argument must at least be of size localDeCount.

Upon return this holds the number of DEs associated with the local VAS.

Upon return this holds the list of DEs associated with the local VAS. The provided argument must at least be of size vasLocalDeCount.

Return code; equals ESMF_SUCCESS if there are no errors.

37.5.6 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.

Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order.
If this occurs, it is recommended to use the standard Fortran call flush(6) as a workaround until this issue is fixed in a future release.

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.
37.5.7 ESMF_DELayoutServiceComplete - Close service window

INTERFACE:

recursive subroutine ESMF_DELayoutServiceComplete(delayout, de, rc)

ARGUMENTS:

    type(ESMF_DELayout), intent(in) :: delayout
    integer, intent(in) :: de
    integer, intent(out), optional :: rc

DESCRIPTION:

The PET who’s service offer was accepted for de must use ESMF_DELayoutServiceComplete to close the service window.

The arguments are:

    delayout Specified ESMF_DELayout object.
    de DE for which to close service window.
    [rc] Return code; equals ESMF_SUCCESS if there are no errors.

37.5.8 ESMF_DELayoutServiceOffer - Offer service for a DE in DELayout

INTERFACE:

recursive function ESMF_DELayoutServiceOffer(delayout, de, rc)

ARGUMENTS:

    type(ESMF_DELayout), intent(in) :: delayout
    integer, intent(in) :: de
    integer, intent(out), optional :: rc

RETURN VALUE:

    type(ESMF_DELayoutServiceReply) :: ESMF_DELayoutServiceOffer

DESCRIPTION:

Offer service for a DE in the ESMF_DELayout object. This call together with ESMF_DELayoutServiceComplete() provides the synchronization primitives between the PETs of an ESMF multi-threaded VM necessary for dynamic load balancing via a work queue approach. The calling PET will either receive ESMF_DELAYOUT_SERVICE_ACCEPT if the service offer has been accepted by DELayout or ESMF_DELAYOUT_SERVICE_DENY if the service offer was denied. The service offer paradigm is different from a simple mutex approach in that DELayout keeps track of the number of service offers issued for each DE by each PET and accepts only one PET’s offer for each offer increment. This requires that all PETs use ESMF_DELayoutServiceOffer() in unison.

The arguments are:

    delayout Specified ESMF_DELayout object.
    de DE for which service is offered by the calling PET.
    [rc] Return code; equals ESMF_SUCCESS if there are no errors.
37.5.9 ESMF_DELayoutValidate - Validate DELayout internals

INTERFACE:

    subroutine ESMF_DELayoutValidate(delayout, rc)

ARGUMENTS:

    type(ESMF_DELayout), intent(in) :: delayout
    integer, intent(out), optional :: rc

DESCRIPTION:

Validates that the delayout is internally consistent. The method returns an error code if problems are found. The arguments are:

delayout Specified ESMF_DELayout object.

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

38 VM Class

38.1 Description

The ESMF_VM (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF component, providing the execution environment for the component code. 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 current version of ESMF a PET is equivalent to an MPI process. Future ESMF releases, however, will provide more flexibility on the PET level.

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 its own VM object is provided to the ESMF_GridCompCreate() or ESMF_CplCompCreate() method. Optionally a petList can be specified to limit the resources the parent gives 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. All registered methods of the component will henceforth execute in the thus defined child VM. Notice that the SetServices routine, although part of the child component, executes before the child VM has been started up. Hence it runs within the parent VM context.

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, unlike MPI, future versions of ESMF will allow PETs to be POSIX threads within multi-threaded POSIX processes. The VM communications API is completely transparent with respect to the different natures of the PETs and provides a common interface to shared memory and message passing communications.

38.2 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 anywhere 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, just like any other VM. 

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. This is done in the child’s SetServices 
routine. 

Notice that the SetServices routine, although part of the child component, executes within the parent VM context. 
The child’s VM has not been started up when the SetServices routine is being called. It is during the return of the 
SetServices call that all required information about the child component’s VM is available and the child’s VM can be 
started up. 

After a child component has been created by the parent, and its SetServices has been called, it may be entered via 
one of the registered initialize / run / finalize entry points. Each time a component is entered through these registered 
methods the associated component routine will start running within the context of the child’s own VM. On return of a 
registered component method the VM is placed on hold, waiting for the next invocation. It is not until the a component 
is destroyed that the associated VM is shut down. 

The user component code may gain access to the VM of its context by querying the active component object via the re-
spective CompGet call. Alternatively, a simpler way to obtain the current VM context is to use the ESMF_VMGetCurrent() 
call which does not require any input information and returns the VM of the current context. Either way, once a 
ESMF_VM object has been obtained it may be used in query and communication calls, and - creating a hierarchy of 
components - to create child components. 

38.2.1 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, vas

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 *, "This PET is localPet: ", localPet
print *, "of a total of ", petCount, " PETs in this VM."
print *, "There are ", peCount, " PEs referenced by this VM"
call ESMF_VMGetPETLocalInfo(vm, localPet, peCount=peCount, ssiId=ssiId, &
   vas=vas, rc=rc)

print *, "This PET is executing in virtual address space (VAS) ", vas
print *, "located on single system image (SSI) ", ssiId
print *, "and is associated with ", peCount, " PEs."
call ESMF_Finalize(rc=rc)
end program

38.2.2 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 across the context.

integer:: mpic
ifndef ESMF_MPIUNI
   integer:: mpic2
endif

call ESMF_VMGet(vm, mpiCommunicator=mpic, rc=rc)

call MPI_Comm_dup(mpic, mpic2, ierr)
call MPI_Barrier(mpic2, ierr)

38.2.3 ESMF inside user MPI application

The following example code demonstrates how ESMF can be used inside of a user application that explicitly calls
MPI_Init() and MPI_Finalize().

call MPI_Init(ierr)

call ESMF_Initialize(rc=rc)

call ESMF_Finalize(terminationflag=ESMF_KEEPMPI, rc=rc)
call MPI_Finalize(ierr)
38.2.4 ESMF inside user defined MPI communicator

The following example code demonstrates how ESMF can run inside of a user defined MPI communicator.

```fortran
    call MPI_Init(ierr)

    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

    if (rank < 2) then
        call ESMF_Initialize(mpiCommunicator=esmfComm, rc=rc)

        call ESMF_Finalize(terminationflag=ESMF_KEEPMPI, rc=rc)
    endif

    call MPI_Finalize(ierr)
```

38.2.5 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.

```fortran
    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)
```

38.2.6 VMScatter/VMGather Example

The VM layer provides MPI-like collective communication. This example demonstrates the use of VM-wide VMScatter and VMGather.

```fortran
    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)
```

38.2.7 VMAllFullReduce Example

The VMAllFullReduce method can be used to find the VM-wide global sum of a data set.

```fortran
    call ESMF_VMAllFullReduce(vm, sendData=array1, recvData=result, count=nsize, &
                               reduceflag=ESMF_SUM, rc=rc)
```
38.2.8 VM Component Example

The following example shows the role that VMs play in connection 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.

```fortran
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 !------------------------------------- -------------------------
```

432
end subroutine !----------------------------------------------------------
end module

program ESMF_VMComponentEx

use ESMF_VMComponentEx_gcomp_mod

gcomp = ESMF_GridCompCreate(name='My gridded component', rc=rc)
call ESMF_GridCompSetServices(gcomp, mygcomp_register, rc)
call ESMF_GridCompInitialize(gcomp, rc=rc)
call ESMF_GridCompRun(gcomp, rc=rc)
call ESMF_GridCompFinalize(gcomp, rc=rc)
call ESMF_GridCompDestroy(gcomp, rc=rc)
call ESMF_Finalize(rc=rc)

end program

38.3 Restrictions and Future Work

1. Non-blocking Reduce() operations not implemented. None of the reduce communication calls have an implementation for the non-blocking feature. This affects:
   • ESMF_VMAllFullReduce()
   • ESMF_VMAllReduce()
   • ESMF_VMRreduce().

2. Limitations when using mpiuni mode. In mpiuni mode non-blocking communications are limited to one outstanding message per source-destination PET pair. Furthermore, in mpiuni mode the message length must be smaller than the internal ESMF buffer size.

3. ESMF-Threading not supported. The ESMF multi-threading features of the VM are enabled but not currently supported. By default VMs run without threads. The entry points to threaded VMs are not currently advertised.

4. Alternative communication paths not accessible. All user accessible VM communication calls are currently implemented using MPI-1.2. VM’s implementation of alternative communication techniques, such as shared memory between threaded PETs and POSIX IPC between PETs located on the same single system image, are currently inaccessible to the user. (One exception to this is the mpiuni case for which the VM automatically utilizes a shared memory path.)

5. Data arrays in VM comm calls are assumed shape with rank=1. Currently all dummy arrays in VM comm calls are defined as assumed shape arrays of rank=1. While this guards against the Fortran copy in/out problem it may not be as flexible as desired. Alternatively all dummy arrays could be defined as assumed size arrays, as it is done in most MPI implementations, thus allowing arrays of various rank to be passed into the comm methods.

6. None of the topology features have been implemented.
38.4 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 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 workloads 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 PETs are associated with its PET and if there are multiple PETs 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 PETs 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.

### 38.5 Class API

#### 38.5.1 ESMF_VMAllFullReduce - Fully reduce data across VM, result on all PETs

**INTERFACE:**

```plaintext
! Private name; call using ESMF_VMAllFullReduce()
subroutine ESMF_VMAllFullReduce<type><kind>(vm, sendData, recvData, count, &
   reduceflag, blockingflag, commhandle, rc)

ARGUMENTS:

  type(ESMF_VM),          intent(in)     :: vm
  <type>(ESMF_KIND_<kind>), target,    intent(in)     :: sendData(:)
  <type>(ESMF_KIND_<kind>), 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
```

436
DESCRIPTION:

Collective ESMF_VM communication call that reduces a contiguous data array of <type><kind> across the ESMF_VM object into a single value of the same <type><kind>. The result is returned on all PETs. Different reduction operations can be specified.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the non-blocking feature. When calling this method with blockingflag = ESMF_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Single data variable to be received. All PETs must specify a valid result variable.

count Number of elements in sendData. Must be the same on all PETs.

reduceflag Reduction operation. See section 9.1.10 for a list of valid reduce operations.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

ESMF_BLOCKING (default) Block until local operation has completed.
ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

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

38.5.2 ESMF_VMAllGather - Gather data across VM, result on all PETs

INTERFACE:

! Private name; call using ESMF_VMAllGather()
subroutine ESMF_VMAllGather<type><kind>(vm, sendData, recvData, count, & blockingflag, commhandle, rc)

ARGUMENTS:

  type(ESMF_VM), intent(in) :: vm
  <type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
  <type>(ESMF_KIND_<kind>), target, intent(out) :: recvData(:)
  integer, intent(in) :: count
  type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
  type(ESMF_CommHandle), intent(out), optional :: commhandle
  integer, intent(out), optional :: rc

437
DESCRIPTION:

Collective ESMF_VM communication call that gathers contiguous data from all PETs of an ESMF_VM object into an array on all PETs.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL.

The arguments are:

- **vm** ESMF_VM object.
- **sendData** Contiguous data array holding data to be send. All PETs must specify a valid source array.
- **recvData** Contiguous data array for data to be received. All PETs must specify a valid recvData argument.
- **count** Number of elements to be gathered from each PET. Must be the same on all PETs.
- **[blockingflag]** Flag indicating whether this call behaves blocking or non-blocking:
  - ESMF_BLOCKING (default) Block until local operation has completed.
  - ESMF_NONBLOCKING Return immediately without blocking.
- **[commhandle]** If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.
- **[rc]** Return code; equals ESMF_SUCCESS if there are no errors.

38.5.3 ESMF_VMAllGatherV - GatherV data across VM, result on all PETs

INTERFACE:

```fortran
! Private name; call using ESMF_VMAllGatherV()
subroutine ESMF_VMAllGatherV<type><kind>(vm, sendData, sendCount, recvData, &
   recvCounts, recvOffsets, blockingflag, commhandle, rc)

ARGUMENTS:

  type(ESMF_VM), intent(in) :: vm
  <type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
  integer, intent(in) :: sendCount
  <type>(ESMF_KIND_<kind>), target, intent(out) :: recvData(:)
  integer, intent(in) :: recvCounts(:)
  integer, intent(in) :: recvOffsets(:)
  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 contiguous data from all PETs of an ESMF_VM object into an array on all PETs.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.
TODO: The current version of this method does not provide an implementation of the non-blocking feature. When calling this method with `blockingflag = ESMF_NONBLOCKING` error code `ESMF_RC_NOT_IMPL` will be returned and an error will be logged.

The arguments are:

- **vm**: ESMF_VM object.
- **sendData**: Contiguous data array holding data to be send. All PETs must specify a valid source array.
- **sendCount**: Number of `sendData` elements to send from local PET to all other PETs.
- **recvData**: Single data variable to be received. All PETs must specify a valid result variable.
- **recvCounts**: Number of `recvData` elements to be received from corresponding source PET.
- **recvOffsets**: Offsets in units of elements in `recvData` marking the start of element sequence to be received from source PET.
- **[blockingflag]**: Flag indicating whether this call behaves blocking or non-blocking:
  - `ESMF_BLOCKING` (default) Block until local operation has completed.
  - `ESMF_NONBLOCKING` Return immediately without blocking.
- **[commhandle]**: If present, a communication handle will be returned in case of a non-blocking request (see argument `blockingflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommQueueWait()` may be used to block on all currently queued communication calls of the VM context.
- **[rc]**: Return code; equals `ESMF_SUCCESS` if there are no errors.

### 38.5.4 ESMF_VMAllReduce - Reduce data across VM, result on all PETs

**INTERFACE:**

```fortran
! Private name; call using ESMF_VMAllReduce()
subroutine ESMF_VMAllReduce<type><kind>(vm, sendData, recvData, count, &
  reduceflag, blockingflag, commhandle, rc)
```

**ARGUMENTS:**

- `type(ESMF_VM), intent(in) :: vm`
- `<type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)`
- `<type>(ESMF_KIND_<kind>), target, 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 reduces a contiguous data array across the ESMF_VM object into a contiguous data array of the same `<type><kind>`. The result array is returned on all PETs. Different reduction operations
can be specified.

This method is overloaded for: `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

**TODO:** The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with `blockingflag = ESMF_NONBLOCKING` error code `ESMF_RC_NOT_IMPL` will be returned and an error will be logged.

The arguments are:

- **`vm`** `ESMF_VM` object.
- **`sendData`** Contiguous data array holding data to be send. All PETs must specify a valid source array.
- **`recvData`** Single data variable to be received. All PETs must specify a valid result variable.
- **`count`** Number of elements in `sendData` and `recvData`. Must be the same on all PETs.
- **`reduceflag`** Reduction operation. See section [9.1.10](#) for a list of valid reduce operations.

- **[`blockingflag`]** Flag indicating whether this call behaves blocking or non-blocking:
  - `ESMF_BLOCKING` (default) Block until local operation has completed.
  - `ESMF_NONBLOCKING` Return immediately without blocking.

- **[`commhandle`]** If present, a communication handle will be returned in case of a non-blocking request (see argument `blockingflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommQueueWait()` may be used to block on all currently queued communication calls of the VM context.

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

### 38.5.5 ESMF_VMAllToAllV - AllToAllV communications across VM

**INTERFACE:**

```fortran
INTERFACE:

  ! Private name; call using ESMF_VMAllToAllV()
  subroutine ESMF_VMAllToAllV<type><kind>(vm, sendData, sendCounts, sendOffsets, &
    recvData, recvCounts, recvOffsets, blockingflag, commhandle, rc)

ARGUMENTS:

  type(ESMF_VM),                intent(in) :: vm
  <type>(ESMF_KIND_<kind>),     target, intent(in) :: sendData(:)
  integer,                      intent(in) :: sendCounts(:)
  integer,                      intent(in) :: sendOffsets(:)
  <type>(ESMF_KIND_<kind>),     target, intent(out) :: recvData(:)
  integer,                      intent(in) :: recvCounts(:)
  integer,                      intent(in) :: recvOffsets(:)
  type(ESMF_BlockingFlag),      intent(in), optional :: blockingflag
  type(ESMF_CommHandle),        intent(out), optional :: commhandle
  integer,                      intent(out), optional :: rc
```

440
Collective ESMF_VM communication call that performs a total exchange operation, sending pieces of the contiguous data buffer sendData to all other PETs while receiving data into the contiguous data buffer recvData from all other PETs.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the non-blocking feature. When calling this method with blockingflag = ESMF_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

- vm ESMF_VM object.
- sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.
- sendCounts Number of sendData elements to send from local PET to destination PET.
- sendOffsets Offsets in units of elements in sendData marking to start of element sequence to be send from local PET to destination PET.
- recvData Single data variable to be received. All PETs must specify a valid result variable.
- recvCounts Number of recvData elements to be received by local PET from source PET.
- recvOffsets Offsets in units of elements in recvData marking to start of element sequence to be received by local PET from source PET.
- [blockingflag] Flag indicating whether this call behaves blocking or non-blocking:
  - ESMF_BLOCKING (default) Block until local operation has completed.
  - ESMF_NONBLOCKING Return immediately without blocking.
- [commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.
- [rc] Return code; equals ESMF_SUCCESS if there are no errors.

38.5.6 ESMF_VMBarrier - VM wide barrier

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 PETs of the VM context have issued the call.

The arguments are:
vm ESMF_VM object.

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

---

38.5.7 ESMF_VMBroadcast - Broadcast data across VM

INTERFACE:

```fortran
! Private name; call using ESMF_VMBroadcast()
subroutine ESMF_VMBroadcast<type><kind>(vm, bcstData, count, root, &
    blockingflag, commhandle, rc)
```

ARGUMENTS:

```fortran
    type(ESMF_VM), intent(in) :: vm
    <type>(ESMF_KIND_<kind>), target, intent(inout) :: bcstData(:)
    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 broadcasts a contiguous data array from PET root to all other PETs of the ESMF_VM object.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL

The arguments are:

vm ESMF_VM object.

bcstData Contiguous data array. On root PET bcstData holds data that is to be broadcasted to all other PETs. On all other PETs bcstData is used to receive the broadcasted data.

count Number of elements in sendData and recvData. Must be the same on all 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 (default) Block until local operation has completed.
ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
38.5.8  ESMF_VMGather - Gather data from across VM

INTERFACE:

! Private name; call using ESMF_VMGather()
subroutine ESMF_VMGather<type><kind>(vm, sendData, recvData, count, root, &
blockingflag, commhandle, rc)

ARGUMENTS:

type(ESMF_VM), intent(in) :: vm
<type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
<type>(ESMF_KIND_<kind>), target, intent(out) :: recvData(:)
integer, intent(in) :: count
integer, intent(in) :: root
[blockingflag] type(ESMF_BlockingFlag), intent(in), optional :: blockingflag
[commhandle] type(ESMF_CommHandle), intent(out), optional :: commhandle
integer, intent(out), optional :: rc

DESCRIPTION:

Collective ESMF_VM communication call that gathers contiguous data from all PETs of an ESMF_VM object (including root) into an array on the root PET.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL

The arguments are:

vm  ESMF_VM object.

sendData  Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData  Contiguous 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 each PET to root. Must be the same on all 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  (default) Block until local operation has completed.
ESMF_NONBLOCKING  Return immediately without blocking.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
38.5.9 ESMF_VMGatherV - GatherV data from across VM

INTERFACE:

! Private name; call using ESMF_VMGatherV()
subroutine ESMF_VMGatherV<type><kind>(vm, sendData, sendCount, recvData, &
recvCounts, recvOffsets, root, rc)

ARGUMENTS:

type(ESMF_VM), intent(in) :: vm
<type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
integer, intent(in) :: sendCount
<type>(ESMF_KIND_<kind>), target, intent(out) :: recvData(:)
integer, intent(in) :: recvCounts(:)
integer, intent(in) :: recvOffsets(:)
integer, intent(in) :: root
integer, intent(out), optional :: rc

DESCRIPTION:

Collective ESMF_VM communication call that gathers contiguous data from all PETs of an ESMF_VM object into an array on root PET.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the non-blocking feature. When calling this method with blockingflag = ESMF_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

sendCount Number of sendData elements to send from local PET to all other PETs.

recvData Single data variable to be received. All PETs must specify a valid result variable.

recvCounts Number of recvData elements to be received from corresponding source PET.

recvOffsets Offsets in units of elements in recvData marking the start of element sequence to be received from source PET.

root Id of the root PET within the ESMF_VM object.

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

38.5.10 ESMF_VMGet - Get VM internals

INTERFACE:

subroutine ESMF_VMGet(vm, localPet, petCount, peCount, mpiCommunicator, &
supportPthreadsFlag, supportOpenMMPFlag, 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 :: supportPthreadsFlag
type (ESMF_Logical), intent (out), optional :: supportOpenMPFlag
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 issued this call.

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.

supportPthreadsFlag  Upon return this holds a flag indicating whether Pthreads are supported by the specified ESMF_VM object.

ESMF_TRUE  Pthreads are supported.
ESMF_FALSE  Pthreads are not supported.

supportOpenMPFlag  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.

38.5.11  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 is created during ESMF_Initialize() and is the ultimate parent of all ESMF_VM objects in an ESMF application.

The arguments are:
Upon return this holds the global default ESMF_VM object.

Return code; equals ESMF_SUCCESS if there are no errors.

38.5.12 ESMF_VMGetCurrent - Get Current VM

INTERFACE:

    subroutine ESMF_VMGetCurrent(vm, rc)

ARGUMENTS:

    type(ESMF_VM), intent(out) :: vm
    integer, intent(out), optional :: rc

DESCRIPTION:

Get the ESMF_VM object of the current execution context.

The arguments are:

vm Upon return this holds the ESMF_VM object of the current context.

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

38.5.13 ESMF_VMGetPETLocalInfo - Get VM PET local internals

INTERFACE:

    subroutine ESMF_VMGetPETLocalInfo(vm, pet, peCount, ssiId, threadCount, &
    threadId, vas, 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 :: vas
    integer, intent(out), optional :: rc

DESCRIPTION:

Get internal information about a specific PET within an 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.
Upon return this holds the id of the single-system image (SSI) the specified PET is running on.

Upon return this holds the number of PETs in the specified PET’s thread group.

Upon return this holds the thread id of the specified PET within the PET’s thread group.

Virtual address space in which this PET operates.

Return code; equals ESMF_SUCCESS if there are no errors.

### 38.5.14 ESMF_VMPrint - Print VM internals

**INTERFACE:**

```fortran
subroutine ESMF_VMPrint(vm, rc)
```

**ARGUMENTS:**

- `type(ESMF_VM), intent(in) :: vm`
- `integer, intent(out), optional :: rc`

**DESCRIPTION:**

Print internal information about the specified ESMF_VM to stdout.

Note: Many ESMF_<class>Print methods are implemented in C++. On some platforms/compilers there is a potential issue with interleaving Fortran and C++ output to stdout such that it doesn’t appear in the expected order. If this occurs, it is recommended to use the standard Fortran call `flush(6)` as a workaround until this issue is fixed in a future release.

The arguments are:

- `vm` Specified ESMF_VM object.
- `rc` Return code; equals ESMF_SUCCESS if there are no errors.

### 38.5.15 ESMF_VMRecv - Receive data from src PET

**INTERFACE:**

```fortran
! Private name; call using ESMF_VMRecv()
subroutine ESMF_VMRecv<type><kind>(vm, recvData, count, src, blockingflag, &
commhandle, rc)
```

**ARGUMENTS:**

- `type(ESMF_VM), intent(in) :: vm`
- `integer(ESMF_KIND_I4), target, intent(out) :: recvData(:)`
- `integer, intent(in) :: count`
- `integer, intent(in), optional :: blockingflag`
- `type(ESMF_CommHandle), intent(out), optional :: commhandle`
- `integer, intent(out), optional :: rc`
DESCRIPTION:

Receive contiguous data from src PET within the same ESMF_VM object.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL, ESMF_TYPEKIND_CHARACTER.

The arguments are:

vm ESMF_VM object.
recvData Contiguous 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 (default) Block until local operation has completed.
  ESMF_NONBLOCKING Return immediately without blocking.

commhandle If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

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

38.5.16 ESMF_VMReduce - Reduce data from across VM

INTERFACE:

! Private name; call using ESMF_VMReduce()
subroutine ESMF_VMReduce<type><kind>(vm, sendData, recvData, count, & reduceflag, root, blockingflag, commhandle, rc)

ARGUMENTS:

  type(ESMF_Vm), intent(in) :: vm
  <type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
  <type>(ESMF_KIND_<kind>), target, intent(out) :: recvData(:)
  integer, intent(in) :: count
  type(ESMF_ReduceFlag), intent(in) :: reduceflag
  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 reduces a contiguous data array across the ESMF_VM object into a contiguous data array of the same <type><kind>. The result array is returned on root PET. Different reduction operations can be specified.
This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the non-blocking feature. When calling this method with blockingflag = ESMF_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Single data variable to be received. All PETs must specify a valid result variable.

count Number of elements in sendData and recvData. Must be the same on all PETs.

reduceflag Reduction operation. See section 9.1.10 for a list of valid reduce operations.

root Id of the root PET within the ESMF_VM object.

[blockingflag] Flag indicating whether this call behaves blocking or non-blocking:

- ESMF_BLOCKING (default) Block until local operation has completed.
- ESMF_NONBLOCKING Return immediately without blocking.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

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

38.5.17 ESMF_VMSscatter - Scatter data across VM

INTERFACE:

! Private name; call using ESMF_VMSscatter()
subroutine ESMF_VMSscatter<type><kind>(vm, sendData, recvData, count, root, &
blockingflag, commhandle, rc)

ARGUMENTS:

type(ESMF_VM), intent(in) :: vm
<type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
<type>(ESMF_KIND_<kind>), target, 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 contiguous data from the root PET to all PETs across the ESMF_VM object (including root).

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL.

The arguments are:
vm  ESMF_VM object.

sendData  Contiguous data array holding data to be send. Only the sendData array specified by the root PET will be used by this method.

recvData  Contiguous 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. Must be the same on all 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  (default) Block until local operation has completed.
  ESMF_NONBLOCKING  Return immediately without blocking.

[commhandle]  If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

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

---

38.5.18  ESMF_VMScatterV - ScatterV across VM

INTERFACE:

! Private name; call using ESMF_VMScatterV()
subroutine ESMF_VMSscatterV<type><kind>(vm, sendData, sendCounts, sendOffsets, & recvData, recvCount, root, rc)

ARGUMENTS:

  type(ESMF_VM),  intent(in) :: vm
  <type>(ESMF_KIND_<kind>), target,  intent(in) :: sendData():
  integer,  intent(in) :: sendCounts():
  integer,  intent(in) :: sendOffsets():
  <type>(ESMF_KIND_<kind>), target,  intent(out) :: recvData():
  integer,  intent(in) :: recvCount
  integer,  intent(in) :: root
  integer,  intent(out), optional :: rc

DESCRIPTION:

Collective ESMF_VM communication call that scatters contiguous data from the root PET to all PETs across the ESMF_VM object (including root).

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

The arguments are:

vm  ESMF_VM object.

sendData  Contiguous data array holding data to be send. Only the sendData array specified by the root PET will be used by this method.
**sendCounts**  Number of *sendData* elements to be send to corresponding receive PET.

**sendOffsets**  Offsets in units of elements in *sendData* marking the start of element sequence to be send to receive PET.

**recvData**  Single data variable to be received. All PETs must specify a valid result variable.

**recvCount**  Number of *recvData* elements to receive by local PET from root PET.

**root**  Id of the root PET within the ESMF_VM object.

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

---

### 38.5.19 ESMF_VMSend - Send data to dst PET

**INTERFACE:**

```fortran
! Private name; call using ESMF_VMSend()
subroutine ESMF_VMSend<type><kind>(vm, sendData, count, dst, blockingflag, &
  commhandle, rc)
ARGUMENTS:
  type(ESMF_VM), intent(in) :: vm
  <type>(ESMF_KIND_<kind>), target, 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 contiguous data to dst PET within the same ESMF_VM object.

The arguments are:

**vm**  ESMF_VM object.

**sendData**  Contiguous 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 (default) Block until local operation has completed.
- ESMF_NONBLOCKING Return immediately without blocking.

[commhandle]  If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
38.5.20  ESMF_VMSendrecv - Send and Recv data to and from PETs

INTERFACE:

! Private name; call using ESMF_VMSendrecv()
subroutine ESMF_VMSendrecv<type><kind>(vm, sendData, sendCount, dst, &
recvData, recvCount, src, blockingflag, commhandle, rc)

ARGUMENTS:

type(ESMF_VM), intent(in) :: vm
<type>(ESMF_KIND_<kind>), target, intent(in) :: sendData(:)
integer, intent(in) :: sendCount
integer, intent(in) :: dst
<type>(ESMF_KIND_<kind>), target, intent(out) :: 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 contiguous data to dst PET within the same ESMF_VM object while receiving contiguous data from src PET within the same ESMF_VM object. The sendData and recvData arrays must be disjoint!

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL, ESMF_TYPEKIND_CHARACTER.

The arguments are:

vm  ESMF_VM object.
sendData  Contiguous 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  Contiguous 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  (default) Block until local operation has completed.
ESMF_NONBLOCKING  Return immediately without blocking.

[commhandle]  If present, a communication handle will be returned in case of a non-blocking request (see argument blockingflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommQueueWait() may be used to block on all currently queued communication calls of the VM context.

[rc]  Return code; equals ESMF_SUCCESS if there are no errors.
38.5.21 ESMF_VMValidate - Validate VM internals

INTERFACE:

    subroutine ESMF_VMValidate(vm, rc)

ARGUMENTS:

    type(ESMF_VM), intent(in) :: vm
    integer, intent(out), optional :: rc

DESCRIPTION:

Validates that the vm is internally consistent. The method returns an error code if problems are found. The arguments are:

vm  Specified ESMF_VM object.

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

38.5.22 ESMF_VMCommWait - Wait for non-blocking VM communication to complete

INTERFACE:

    subroutine ESMF_VMCommWait(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 specified by the commhandle to complete.

The arguments are:

vm  ESMF_VM object.

commhandle Handle specifying a previously issued non-blocking communication request.

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

38.5.23 ESMF_VMCommQueueWait - Wait for all non-blocking VM comms to complete

INTERFACE:

    subroutine ESMF_VMCommQueueWait(vm, rc)

ARGUMENTS:
38.5.24 ESMF_VMWtime - Get floating-point number of seconds

INTERFACE:

subroutine ESMF_VMWtime(time, rc)

ARGUMENTS:

real(ESMF_KIND_R8), intent(out) :: time
integer, intent(out), optional :: rc

DESCRIPTION:

Get floating-point number of seconds of elapsed wall-clock time since some time in the past.

The arguments are:

time Time in seconds.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

38.5.25 ESMF_VMWtimeDelay - Delay execution

INTERFACE:

subroutine ESMF_VMWtimeDelay(delay, rc)

ARGUMENTS:

real(ESMF_KIND_R8), intent(in) :: delay
integer, intent(out), optional :: rc

DESCRIPTION:

Delay execution for amount of seconds.

The arguments are:

delay Delay time in seconds.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.
38.5.26 ESMF_VMWtimePrec - Timer precision as floating-point number of seconds

INTERFACE:

    subroutine ESMF_VMWtimePrec(prec, rc)

ARGUMENTS:

    real(ESMF_KIND_R8) :: prec
    integer, optional :: rc

DESCRIPTION:

Get a run-time estimate of the timer precision as floating-point number of seconds. This is a relatively expensive call since the timer precision is measured several times before the maximum is returned as the estimate. The returned value is PET-specific and may differ across the VM context.

The arguments are:

prec  Timer precision in seconds.

rc    Return code; equals ESMF_SUCCESS if there are no errors.
39 References

References


40 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. [8].

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.
# Appendix B: ESMF Error Return Codes

The tables below show the possible error return codes for Fortran and C++ methods.

<table>
<thead>
<tr>
<th>Success/Failure Return codes for both Fortran and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESMF_SUCCESS 0</td>
</tr>
<tr>
<td>ESMF_FAILURE -1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran Symmetric Return Codes 1-500</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESMF_RC_OBJ_BAD 1</td>
</tr>
<tr>
<td>ESMF_RC_OBJ_INIT 2</td>
</tr>
<tr>
<td>ESMF_RC_OBJ_CREATE 3</td>
</tr>
<tr>
<td>ESMF_RC_OBJ_COR 4</td>
</tr>
<tr>
<td>ESMF_RC_OBJ_WRONG 5</td>
</tr>
<tr>
<td>ESMF_RC_ARG_BAD 6</td>
</tr>
<tr>
<td>ESMF_RC_ARG_RANK 7</td>
</tr>
<tr>
<td>ESMF_RC_ARG_SIZE 8</td>
</tr>
<tr>
<td>ESMF_RC_ARG_VALUE 9</td>
</tr>
<tr>
<td>ESMF_RC_ARG_DUP 10</td>
</tr>
<tr>
<td>ESMF_RC_ARG_SAMETYPE 11</td>
</tr>
<tr>
<td>ESMF_RC_ARG_SAMECOMM 12</td>
</tr>
<tr>
<td>ESMF_RC_ARG_INCOMP 13</td>
</tr>
<tr>
<td>ESMF_RC_ARG_CORRUPT 14</td>
</tr>
<tr>
<td>ESMF_RC_ARG_WRONG 15</td>
</tr>
<tr>
<td>ESMF_RC_ARG_OUTOF RANGE 16</td>
</tr>
<tr>
<td>ESMF_RC_ARG_OPT 17</td>
</tr>
<tr>
<td>ESMF_RC_NOT_IMPL 18</td>
</tr>
<tr>
<td>ESMF_RC_FILE_OPEN 19</td>
</tr>
<tr>
<td>ESMF_RC_FILE_CREATE 20</td>
</tr>
<tr>
<td>ESMF_RC_FILE_READ 21</td>
</tr>
<tr>
<td>ESMF_RC_FILE_WRITE 22</td>
</tr>
<tr>
<td>ESMF_RC_FILE_UNEXPECTED 23</td>
</tr>
<tr>
<td>ESMF_RC_FILE_CLOSE 24</td>
</tr>
<tr>
<td>ESMF_RC_FILE_ACTIVE 25</td>
</tr>
<tr>
<td>ESMF_RC_PTR_NULL 26</td>
</tr>
<tr>
<td>ESMF_RC_PTR_BAD 27</td>
</tr>
<tr>
<td>ESMF_RC_PTR_NOTALLOC 28</td>
</tr>
<tr>
<td>ESMF_RC_PTR_ISALLOC 29</td>
</tr>
<tr>
<td>ESMF_RC_MEM 30</td>
</tr>
<tr>
<td>ESMF_RC_MEMC 31</td>
</tr>
<tr>
<td>ESMF_RC_DUP_NAME 32</td>
</tr>
<tr>
<td>ESMF_RC_LONG_NAME 33</td>
</tr>
<tr>
<td>ESMF_RC_LONG_STR 34</td>
</tr>
<tr>
<td>ESMF_RC_COPY_FAIL 35</td>
</tr>
<tr>
<td>ESMF_RC_DIV_ZERO 36</td>
</tr>
<tr>
<td>ESMF_RC_CANNOT_GET 37</td>
</tr>
<tr>
<td>ESMF_RC_CANNOT_SET 38</td>
</tr>
<tr>
<td>ESMF_RC_NOT_FOUND 39</td>
</tr>
<tr>
<td>ESMF_RC_NOT_VALID 40</td>
</tr>
<tr>
<td>ESMF_RC_INTNRL_LIST 41</td>
</tr>
</tbody>
</table>
ESMF_RC_INTNRL_INCONS 42
ESMF_RC_INTNRL_BAD 43
ESMF_RC_SYS 44
ESMF_RC_BUSY 45
ESMF_RC_LIB 46
ESMF_RC_ATTR_UNUSED 47
ESMF_RC_OBJ_NOT_CREATED 48
ESMF_RC_OBJ_DELETED 49
ESMF_RC_NOT_SET 50
ESMF_RC_VAL_WRONG 51
ESMF_RC_VAL_ERRBOUND 52
ESMF_RC_VAL_OUTOFRANGE 53
ESMF_RC_OBJ_UNINIT 54

54-500 reserved for future Fortran symmetric return code definitions

=====================================  
C++ Symmetric Return Codes 501-999  
=====================================  

ESMC_RC_OBJ_BAD 501
ESMC_RC_OBJ_INIT 502
ESMC_RC_OBJ_CREATE 503
ESMC_RC_OBJ_COR 504
ESMC_RC_OBJ_WRONG 505
ESMC_RC_ARG_BAD 506
ESMC_RC_ARG_RANK 507
ESMC_RC_ARG_SIZE 508
ESMC_RC_ARG_VALUE 509
ESMC_RC_ARG_DUP 510
ESMC_RC_ARG_SAMETYPE 511
ESMC_RC_ARG_SAMECOMM 512
ESMC_RC_ARG_CORRUPT 513
ESMC_RC_ARG WRONG 514
ESMC_RC_ARG_OUTOFRANGE 515
ESMC_RC_ARG_OPT 516
ESMC_RC_NOT_IMPL 517
ESMC_RC_FILE_OPEN 519
ESMC_RC_FILE_CREATE 520
ESMC_RC_FILE_READ 521
ESMC_RC_FILE_WRITE 522
ESMC_RC_FILE_UNEXPECTED 523
ESMC_RC_FILE_CLOSE 524
ESMC_RC_PTR_NULL 526
ESMC_RC_PTR_BAD 527
ESMC_RC_PTR_NOTALLOC 528
ESMC_RC_PTR_ISALLOC 529
ESMC_RC_MEM 530
ESMC_RC_MEMC 531
ESMC_RC_DUP_NAME 532
ESMC_RC_LONG_NAME 533
ESMC_RC_LONG_STR 534
ESMC_RC_COPY_FAIL 535
ESMC_RC_DIV_ZERO      536
ESMC_RC_CANNOT_GET    537
ESMC_RC_CANNOT_SET    538
ESMC_RC_NOT_FOUND     539
ESMC_RC_NOT_VALID     540
ESMC_RC_INTNRL_LIST   541
ESMC_RC_INTNRL_INCONS 542
ESMC_RC_INTNRL_BAD    543
ESMC_RC_SYS           544
ESMC_RC_BUSY          545
ESMC_RC_LIB           546
ESMC_RC_ATTR_UNUSED   547
ESMC_RC_OBJ_NOT_CREATED 548
ESMC_RC_OBJ_DELETED   549
ESMC_RC_NOT_SET       550
ESMC_RC_VAL_WRONG     551
ESMC_RC_VAL_ERRBOUND  552
ESMC_RC_VAL_OUTOFRAQUE 553
ESMC_RC_OBJ_UNINIT    554

554-999 reserved for future C++ symmetric return code definitions

=====================================  
C++ Non-symmetric Return Codes 1000
=====================================  

ESMC_RC_OPTARG_BAD      1000