Earth System Modeling Framework

# ESMF Reference Manual for C

**Version 9.0.0 beta snapshot**

*ESMF Joint Specification Team: V. Balaji, Byron Boville, Samson Cheung, Tom Clune, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Rosalinda de Fainchtein, Rocky Dunlap, Brian Eaton, Steve Goldhaber, Bob Hallberg, Tom Henderson, Chris Hill, Mark Iredell, Joseph Jacob, Rob Jacob, Phil Jones, Brian Kauffman, Erik Kluzek, Ben Koziol, Jay Larson, Peggy Li, Fei Liu, John Michalakes, Raffaele Montuoro, Sylvia Murphy, David Neckels, Ryan O Kuinghttons, Bob Oehmke, Chuck Panaccione, Daniel Rosen, Jim Rosinski, Mathew Rothstein, Bill Sacks, Kathy Saint, Will Sawyer, Earl Schwab, Shepard Smithline, Walter Spector, Don Stark, Max Suarez, Spencer Swift, Gerhard Theurich, Atanas Trayanov, Silverio Vasquez, Jon Wolfe, Weiyu Yang, Mike Young, Leonid Zaslavsky*

November 3, 2026

---

http://www.earthsystemmodeling.org

**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.

- Parallel I/O (PIO) developers at NCAR and DOE Laboratories for their excellent work on this package and their help in making it work with ESMF
- 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


**I ESMF Overview** **10**

**1 What is the Earth System Modeling Framework?** **11**

**2 The ESMF Reference Manual for C** **11**

**3 How to Contact User Support and Find Additional Information** **12**

**4 How to Submit Comments, Bug Reports, and Feature Requests** **12**

**5 The ESMF Application Programming Interface** **13**
5.1 Standard Methods and Interface Rules . . . 13
5.2 Deep and Shallow Classes . . . 13
5.3 Aliases . . . 14
5.4 Special Methods . . . 14
5.5 The ESMF Data Hierarchy . . . 15
5.6 ESMF Spatial Classes . . . 15
5.7 ESMF Maps . . . 16
5.8 ESMF Specification Classes . . . 16
5.9 ESMF Utility Classes . . . 16

**6 Integrating ESMF into Applications** **17**
6.1 Using the ESMF Superstructure . . . 17
6.2 Constants . . . 18

**7 Overall Rules and Behavior** **18**
7.1 Local and Global Views and Associated Conventions . . . 18
7.2 Allocation Rules . . . 18
7.3 Assignment, Equality, Copying and Comparing Objects . . . 19

**8 Overall Design and Implementation Notes** **19**

**II Command Line Tools** **20**

**III Superstructure** **21**

**9 Overview of Superstructure** **22**
9.1 Superstructure Classes . . . 22
9.2 Hierarchical Creation of Components . . . 23
9.3 Sequential and Concurrent Execution of Components . . . 24
9.4 Intra-Component Communication . . . 25
9.5 Data Distribution and Scoping in Components . . . 25
9.6 Performance . . . 25
9.7 Object Model . . . 29

**10 Application Driver and Required ESMF Methods** **29**
10.1 Description . . . 29
10.2 Required ESMF Methods . . . 30

10.2.1 ESMC_Initialize . . . 30
10.2.2 ESMC_FinalizeWithFlag . . . 32
10.2.3 ESMC_Finalize . . . 32

**11 GridComp Class** **32**
11.1 Description . . . 32
11.2 Class API . . . 33
11.2.1 ESMC_GridCompCreate . . . 33
11.2.2 ESMC_GridCompDestroy . . . 34
11.2.3 ESMC_GridCompFinalize . . . 34
11.2.4 ESMC_GridCompGetInternalState . . . 35
11.2.5 ESMC_GridCompInitialize . . . 36
11.2.6 ESMC_GridCompPrint . . . 37
11.2.7 ESMC_GridCompRun . . . 37
11.2.8 ESMC_GridCompSetEntryPoint . . . 38
11.2.9 ESMC_GridCompSetInternalState . . . 39
11.2.10 ESMC_GridCompSetServices . . . 39

**12 CplComp Class** **40**
12.1 Description . . . 40
12.2 Class API . . . 41
12.2.1 ESMC_CplCompCreate . . . 41
12.2.2 ESMC_CplCompDestroy . . . 41
12.2.3 ESMC_CplCompFinalize . . . 42
12.2.4 ESMC_CplCompGetInternalState . . . 43
12.2.5 ESMC_CplCompInitialize . . . 43
12.2.6 ESMC_CplCompPrint . . . 44
12.2.7 ESMC_CplCompRun . . . 45
12.2.8 ESMC_CplCompSetEntryPoint . . . 45
12.2.9 ESMC_CplCompSetInternalState . . . 46
12.2.10 ESMC_CplCompSetServices . . . 47

**13 SciComp Class** **47**
13.1 Description . . . 47
13.2 Class API . . . 48
13.2.1 ESMC_SciCompCreate . . . 48
13.2.2 ESMC_SciCompDestroy . . . 48
13.2.3 ESMC_SciCompPrint . . . 49

**14 State Class** **49**
14.1 Description . . . 49
14.2 Restrictions and Future Work . . . 49
14.3 Class API . . . 50
14.3.1 ESMC_StateAddArray . . . 50
14.3.2 ESMC_StateAddField . . . 50
14.3.3 ESMC_StateCreate . . . 51
14.3.4 ESMC_StateDestroy . . . 51
14.3.5 ESMC_StateGetArray . . . 52
14.3.6 ESMC_StateGetField . . . 52
14.3.7 ESMC_StatePrint . . . 53

**IV Infrastructure: Fields and Grids** **54**

**15 Overview of Infrastructure Data Handling** **55**
15.1 Infrastructure Data Classes . . . 55
15.2 Design and Implementation Notes . . . 56

**16 Field Class** **57**
16.1 Description . . . 57
16.2 Constants . . . 57
16.2.1 ESMC_REGRIDMETHOD . . . 57
16.3 Use and Examples . . . 58
16.3.1 Field create and destroy . . . 58
16.4 Class API . . . 58
16.4.1 ESMC_FieldCreateGridArraySpec . . . 58
16.4.2 ESMC_FieldCreateGridTypeKind . . . 60
16.4.3 ESMC_FieldCreateMeshArraySpec . . . 61
16.4.4 ESMC_FieldCreateMeshTypeKind . . . 62
16.4.5 ESMC_FieldCreateLocStreamArraySpec . . . 63
16.4.6 ESMC_FieldCreateLocStreamTypeKind . . . 64
16.4.7 ESMC_FieldDestroy . . . 65
16.4.8 ESMC_FieldGetArray . . . 66
16.4.9 ESMC_FieldGetMesh . . . 66
16.4.10 ESMC_FieldGetPtr . . . 67
16.4.11 ESMC_FieldGetBounds . . . 67
16.4.12 ESMC_FieldGetLocalDECount . . . 68
16.4.13 ESMC_FieldPrint . . . 68
16.4.14 ESMC_FieldRegridGetArea . . . 69
16.4.15 ESMC_FieldRegridStore . . . 69
16.4.16 ESMC_FieldRegridStoreFile . . . 72
16.4.17 ESMC_FieldRegrid . . . 74
16.4.18 ESMC_FieldRegridRelease . . . 75
16.4.19 ESMC_FieldSMMStore . . . 75

**17 Array Class** **76**
17.1 Description . . . 76
17.2 Class API . . . 77
17.2.1 ESMC_ArrayCreate . . . 77
17.2.2 ESMC_ArrayDestroy . . . 77
17.2.3 ESMC_ArrayGetName . . . 78
17.2.4 ESMC_ArrayGetLocalDECount . . . 78
17.2.5 ESMC_ArrayGetPtr . . . 79
17.2.6 ESMC_ArrayPrint . . . 79
17.3 Class API: DynamicMask Methods . . . 80
17.3.1 ESMC_DynamicMaskPredefinedSetR8R8R8 . . . 80
17.3.2 ESMC_DynamicMaskPredefinedSetR8R8R8V . . . 81
17.3.3 ESMC_DynamicMaskPredefinedSetR4R8R4 . . . 81
17.3.4 ESMC_DynamicMaskPredefinedSetR4R8R4V . . . 82

**18 ArraySpec Class** **83**
18.1 Description . . . 83
18.2 Class API . . . 83
18.2.1 ESMC_ArraySpecGet . . . 83

18.2.2 ESMC_ArraySpecSet . . . 83

**19 Grid Class** **84**
19.1 Description . . . 84
19.1.1 Grid Representation in ESMF . . . 84
19.1.2 Supported Grids . . . 85
19.1.3 Grid Topologies and Periodicity . . . 85
19.1.4 Grid Distribution . . . 86
19.1.5 Grid Coordinates . . . 87
19.1.6 Coordinate Specification and Generation . . . 87
19.1.7 Staggering . . . 87
19.1.8 Masking . . . 88
19.2 Constants . . . 88
19.2.1 ESMC_COORDSYS . . . 88
19.2.2 ESMC_GRIDITEM . . . 89
19.2.3 ESMC_GRIDSTATUS . . . 89
19.2.4 ESMC_POLEKIND . . . 90
19.2.5 ESMC_STAGGERLOC . . . 90
19.2.6 ESMC_FILEFORMAT . . . 92
19.3 Restrictions and Future Work . . . 92
19.4 Design and Implementation Notes . . . 93
19.4.1 Grid Topology . . . 93
19.5 Class API: General Grid Methods . . . 93
19.5.1 ESMC_GridCreateNoPeriDim . . . 93
19.5.2 ESMC_GridCreate1PeriDim . . . 94
19.5.3 ESMC_GridCreateCubedSphere . . . 94
19.5.4 ESMC_GridCreateFromFile . . . 95
19.5.5 ESMC_GridDestroy . . . 97
19.5.6 ESMC_GridAddItem . . . 97
19.5.7 ESMC_GridGetItem . . . 98
19.5.8 ESMC_GridAddCoord . . . 98
19.5.9 ESMC_GridGetCoord . . . 99
19.5.10 ESMC_GridGetCoordBounds . . . 100
19.5.11 ESMC_GridGetLocalDECount . . . 100

**20 Mesh Class** **101**
20.1 Description . . . 101
20.1.1 Mesh Representation in ESMF . . . 101
20.1.2 Supported Meshes . . . 101
20.2 Constants . . . 102
20.2.1 ESMC_MESHELEMTYPE . . . 102
20.2.2 ESMF_FILEFORMAT . . . 103
20.3 Class API . . . 103
20.3.1 ESMC_MeshGetMOAB . . . 103
20.3.2 ESMC_MeshSetMOAB . . . 104
20.3.3 ESMC_MeshAddElements . . . 104
20.3.4 ESMC_MeshAddNodes . . . 106
20.3.5 ESMC_MeshCreate . . . 106
20.3.6 ESMC_MeshCreateFromFile . . . 107
20.3.7 ESMC_MeshGetCoord . . . 108
20.3.8 ESMC_MeshGetElemCoord . . . 109

20.3.9 ESMC_MeshGetConnectivity . . . 110
20.3.10 ESMC_MeshDestroy . . . 110
20.3.11 ESMC_MeshFreeMemory . . . 111
20.3.12 ESMC_MeshGetElementCount . . . 111
20.3.13 ESMC_MeshGetNodeCount . . . 112
20.3.14 ESMC_MeshGetOwnedElementCount . . . 112
20.3.15 ESMC_MeshGetOwnedNodeCount . . . 113

**21 XGrid Class 113**
21.1 Description . . . 113
21.2 Restrictions and Future Work . . . 114
21.2.1 Restrictions and Future Work . . . 114
21.3 Design and Implementation Notes . . . 114
21.4 Class API . . . 114
21.4.1 ESMC_XGridCreate . . . 114
21.4.2 ESMC_XGridDestroy . . . 116
21.4.3 ESMC_XGridGetSideAGridCount . . . 117
21.4.4 ESMC_XGridGetSideAMeshCount . . . 117
21.4.5 ESMC_XGridGetSideBGridCount . . . 118
21.4.6 ESMC_XGridGetSideBMeshCount . . . 118
21.4.7 ESMC_XGridGetDimCount . . . 119
21.4.8 ESMC_XGridGetElementCount . . . 119
21.4.9 ESMC_XGridGetMesh . . . 120
21.4.10 ESMC_XGridGetElementArea . . . 120
21.4.11 ESMC_XGridGetElementCentroid . . . 121
21.4.12 ESMC_XGridGetSparseMatA2X . . . 121
21.4.13 ESMC_XGridGetSparseMatX2A . . . 122
21.4.14 ESMC_XGridGetSparseMatB2X . . . 123
21.4.15 ESMC_XGridGetSparseMatX2B . . . 124

**22 DistGrid Class 124**
22.1 Description . . . 124
22.2 Class API . . . 125
22.2.1 ESMC_DistGridCreate . . . 125
22.2.2 ESMC_DistGridDestroy . . . 126
22.2.3 ESMC_DistGridPrint . . . 126

**23 RouteHandle Class 126**
23.1 Description . . . 126
23.2 Use and Examples . . . 127
23.3 Restrictions and Future Work . . . 127
23.4 Design and Implementation Notes . . . 127
23.5 Class API . . . 127
23.5.1 ESMC_RouteHandleCreateFromFile . . . 127
23.5.2 ESMC_RouteHandlePrint . . . 128
23.5.3 ESMC_RouteHandleWrite . . . 128

**V Infrastructure: Utilities 130**

**24 Overview of Infrastructure Utility Classes 131**

**25 Time Manager Utility** **132**
25.1 Time Manager Classes . . . 132
25.2 Calendar . . . 132
25.3 Time Instants and TimeIntervals . . . 133
25.4 Clocks . . . 133

**26 Calendar Class** **134**
26.1 Description . . . 134
26.2 Constants . . . 134
26.2.1 ESMC_CALKIND . . . 134
26.3 Class API . . . 135
26.3.1 ESMC_CalendarCreate . . . 135
26.3.2 ESMC_CalendarDestroy . . . 135
26.3.3 ESMC_CalendarPrint . . . 136

**27 Time Class** **137**
27.1 Description . . . 137
27.2 Class API . . . 137
27.2.1 ESMC_TimeGet . . . 137
27.2.2 ESMC_TimePrint . . . 138
27.2.3 ESMC_TimeSet . . . 138

**28 TimeInterval Class** **140**
28.1 Description . . . 140
28.2 Class API . . . 140
28.2.1 ESMC_TimeIntervalGet . . . 140
28.2.2 ESMC_TimeIntervalPrint . . . 140
28.2.3 ESMC_TimeIntervalSet . . . 141

**29 Clock Class** **142**
29.1 Description . . . 142
29.2 Class API . . . 142
29.2.1 ESMC_ClockAdvance . . . 142
29.2.2 ESMC_ClockCreate . . . 142
29.2.3 ESMC_ClockDestroy . . . 143
29.2.4 ESMC_ClockGet . . . 144
29.2.5 ESMC_ClockPrint . . . 144

**30 Config Class** **145**
30.1 Description . . . 145
30.1.1 Package history . . . 145
30.2 Class API . . . 145
30.2.1 ESMC_ConfigCreate . . . 145
30.2.2 ESMC_ConfigDestroy . . . 145
30.2.3 ESMC_ConfigCreateFromSection . . . 146
30.2.4 ESMC_ConfigFindLabel . . . 147
30.2.5 ESMC_ConfigFindNextLabel . . . 147
30.2.6 ESMC_ConfigGetDim . . . 148
30.2.7 ESMC_ConfigGetLen . . . 149
30.2.8 ESMC_ConfigLoadFile . . . 149
30.2.9 ESMC_ConfigNextLine . . . 150
30.2.10 ESMC_ConfigPrint . . . 150

30.2.11 ESMC_ConfigValidate . . . 151

**31 Log Class** **151**
31.1 Description . . . 151
31.2 Constants . . . 152
31.2.1 ESMC_LOGKIND . . . 152
31.2.2 ESMC_LOGMSG . . . 152
31.3 Class API . . . 152
31.3.1 ESMC_LogMsgFoundError . . . 152
31.3.2 ESMC_LogSet . . . 153
31.3.3 ESMC_LogWrite . . . 154

**32 VM Class** **154**
32.1 Description . . . 154
32.2 Class API . . . 155
32.2.1 ESMC_VMBarrier . . . 155
32.2.2 ESMC_VMBroadcast . . . 155
32.2.3 ESMC_VMGet . . . 156
32.2.4 ESMC_VMGetCurrent . . . 157
32.2.5 ESMC_VMGetGlobal . . . 157
32.2.6 ESMC_VMLogMemInfo . . . 158
32.2.7 ESMC_VMPrint . . . 158
32.2.8 ESMC_VMReduce . . . 159

**33 Profiling and Tracing** **160**
33.1 Description . . . 160
33.1.1 Profiling . . . 160
33.1.2 Tracing . . . 160
33.2 Restrictions and Future Work . . . 161
33.3 Class API . . . 161
33.3.1 ESMC_TraceRegionEnter . . . 161
33.3.2 ESMC_TraceRegionExit . . . 162

**VI References** **163**

**VII Appendices** **164**

**34 Appendix A: Master List of Constants** **164**
34.1 ESMC_CALKIND . . . 164
34.2 ESMC_COORDSYS . . . 164
34.3 ESMC_DECOMP . . . 164
34.4 ESMC_FILEFORMAT . . . 165
34.5 ESMC_GRIDITEM . . . 165
34.6 ESMC_GRIDSTATUS . . . 165
34.7 ESMC_INDEX . . . 165
34.8 ESMC_LINETYPE . . . 166
34.9 ESMC_LOGKIND . . . 166
34.10 ESMC_LOGMSG . . . 166
34.11 ESMC_MESHELEMTYPE . . . 166
34.12 ESMF_METHOD . . . 166

34.13ESMC_POLEKIND . . . . . 167
34.14ESMC_REDUCE . . . . . 167
34.15ESMC_REGION . . . . . 167
34.16ESMC_REGRIDMETHOD . . . . . 167
34.17ESMC_STAGGERLOC . . . . . 167
34.18ESMC_TYPEKIND . . . . . 168
34.19ESMC_UNMAPPEDACTION . . . . . 168

**35 Appendix B: A Brief Introduction to UML** **168**

**36 Appendix C: ESMF Error Return Codes** **169**

# 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 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.

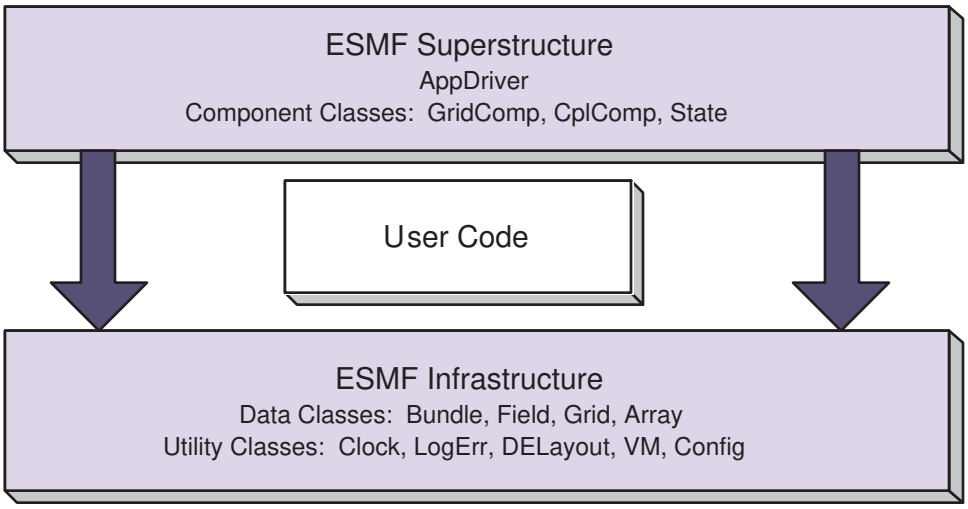
# 2 The ESMF Reference Manual for C

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

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

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.




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.

# 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.

The website, http://www.earthsystemmodeling.org, provide more information of the ESMF project as a whole. 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 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 is 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 C interface is implemented so that the variables associated with a class are stored in a C structure. For example, an `ESMC_Field` structure stores the data array, grid information, and metadata associated with a physical field. The structure for each class is defined in a C header file. The operations associated with each class are also defined in the header files.

The header files for ESMF are bundled together and can be accessed with a single `include` statement, `#include "ESMC.h"`. By convention, the C entry points are named using "ESMC" as a prefix.

## 5.1 Standard Methods and Interface Rules

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

- `ESMC_<Class>Create()` and `ESMC_<Class>Destroy()`, for creating and destroying objects of ESMF classes that require internal memory management (- called ESMF deep classes). The `ESMC_<Class>Create()` method allocates memory for the object itself and for internal variables, and initializes variables where appropriate. It is always written as a function that returns a derived type instance of the class, i.e. an object.
- `ESMC_<Class>Set()` and `ESMC_<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 `ESMC_<Class>Set<Something>()` and `ESMC_<Class>Get<Something>()` interfaces.
- `ESMC_<Class>Add()`, `ESMC_<Class>AddReplace()`, `ESMC_<Class>Remove()`, and `ESMC_<Class>Replace()`, for manipulating objects of ESMF container classes - such as `ESMC_State` and `ESMC_FieldBundle`. For example, the `ESMC_FieldBundleAdd()` method adds another Field to an existing FieldBundle object.
- `ESMC_<Class>Print()`, for printing the contents of an object to standard out. This method is mainly intended for debugging.
- `ESMC_<Class>ReadRestart()` and `ESMC_<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.
- `ESMC_<Class>Validate()`, for determining whether a class is internally consistent. For example, `ESMC_FieldValidate()` validates the internal consistency of a Field object.

## 5.2 Deep and Shallow Classes

ESMF contains two types of classes.

**Deep** classes require ESMC_<Class>Create() and ESMC_<Class>Destroy() calls. They involve memory allocation, take significant time to set up (due to memory management) 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 GridComp, CplComp, State, Fields, FieldBundles, Arrays, ArrayBundles, Grids, and Clocks, fall into this category.

**Shallow** classes do not possess ESMC_<Class>Create() and ESMC_<Class>Destroy() calls. They are simply declared and their values set using an ESMC_<Class>Set() call. Examples of shallow classes are Time, TimeInterval, and ArraySpec. 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 execution goes out of the declaring scope.

An exception to this is when a shallow object, such as a Time, is stored in a deep object such as a Clock. The deep Clock object then becomes the declaring scope of the Time object, persisting in memory. The Time object is deallocated with the ESMC_ClockDestroy() call.

See Section 8, 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*.

## 5.3 Aliases

Deep objects, i.e. instances of ESMF deep classes created by the appropriate ESMC_<Class>Create(), can be used with the standard assignment (=) operator.

The assignment

```
deep2 = deep1
```

makes deep2 an **alias** of deep1, meaning that both variables reference the same deep allocation in memory. Many aliases of the same deep object can be created.

All the aliases of a deep object are equivalent. In particular, there is no distinction between the variable on the left hand side of the actual ESMC_<Class>Create() call, and any aliases created from it. All actions taken on any of the aliases of a deep object affect the deep object, and thus all other aliases.

## 5.4 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.

- ESMC_Initialize() and ESMC_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 does not support restarts in the same executable, i.e. ESMC_Initialize() should not be called after ESMC_Finalize().
- ESMC_<Type>CompInitialize(), ESMC_<Type>CompRun(), and ESMC_<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 subroutines within user code.

## 5.5 The ESMF Data Hierarchy

The ESMF API is organized around a 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 offered by the ESMF C API, 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 language array.
- **Field** A Field represents a physical scalar or vector field. It contains a reference to an Array along with grid information and metadata.
- **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 Arrays and Fields can be queried for the C pointer to the actual data. You can perform communication operations either on the ESMF data objects or directly on C arrays through the VM class, which serves as a unifying wrapper for distributed and shared memory communication libraries.

## 5.6 ESMF Spatial Classes

Like the hierarchy of model data classes, ranging from the simple to the complex, ESMF is organized around a 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.

  The current ESMF C API does not provide user access to the DELayout class.

- **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 for a logically rectangular region in 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.
- **Mesh** A Mesh provides an abstraction for an unstructured grid. Coordinate information is set in nodes, which represent vertices or corners. Together the nodes establish the boundaries of mesh elements or cells.
- **LocStream** A LocStream is an abstraction for a set of unstructured data points without any topological relationship to each other.
- **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.

## 5.7 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.

## 5.8 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.

## 5.9 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. (Not currently available through the ESMF C API.)
- **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.

# 6 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.

## 6.1 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.

## 6.2 Constants

Named constants are used throughout ESMF to specify the values of many arguments with multiple well defined values in a consistent way. These constants are defined by a derived type that follows this pattern:

```
ESMF_<CONSTANT_NAME>_Flag
```

The values of the constant are then specified by this pattern:

```
ESMF_<CONSTANT_NAME>_<VALUE1>
ESMF_<CONSTANT_NAME>_<VALUE2>
ESMF_<CONSTANT_NAME>_<VALUE3>
...
```

A master list of all available constants can be found in section 34.

# 7 Overall Rules and Behavior

## 7.1 Local and Global Views and Associated Conventions

ESMF data objects such as Fields are distributed over DEs, with each DE getting a portion of the data. Depending on the task, a local or global view of the object may be preferable. In a local view, data indices start with the first element on the DE and end with the last element on the same DE. In a global view, there is an assumed or specified order to the set of DEs over which the object is distributed. Data indices start with the first element on the first DE, and continue across all the elements in the sequence of DEs. The last data index represents the number of elements in the entire object. The DistGrid provides the mapping between local and global data indices.

The convention in ESMF is that entities with a global view have no prefix. Entities with a DE-local (and in some cases, PET-local) view have the prefix "local."

Just as data is distributed over DEs, DEs themselves can be distributed over PETs. This is an advanced feature for users who would like to create multiple local chunks of data, for algorithmic or performance reasons. Local DEs are those DEs that are located on the local PET. Local DE labeling always starts at 0 and goes to localDeCount-1, where localDeCount is the number of DEs on the local PET. Global DE numbers also start at 0 and go to deCount-1. The DELayout class provides the mapping between local and global DE numbers.

## 7.2 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 `ESMC_FieldCreate()` with the `ESMF_DATACOPY_REFERENCE` 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.

### 7.3 Assignment, Equality, Copying and Comparing Objects

The equal sign assignment has not been overloaded in ESMF, thus resulting in the standard C behavior. This behavior has been documented as the first entry in the API documentation section for each ESMF class. For deep ESMF objects the assignment results in setting an alias the the same ESMF object in memory. For shallow ESMF objects the assignment is essentially a equivalent to a copy of the object. For deep classes the equality operators have been overloaded to test for the alias condition as a counter part to the assignment behavior. This and the not equal operator are documented following the assignment in the class API documentation sections.

Deep object copies are implemented as a special variant of the `ESMC_<Class>Create()` methods. It takes an existing deep object as one of the required arguments. At this point not all deep classes have `ESMC_<Class>Create()` methods that allow object copy.

Due to the complexity of deep classes there are many aspects when comparing two objects of the same class. ESMF provide `ESMC_<Class>Match()` methods, which are functions that return a class specific match flag. At this point not all deep classes have `ESMC_<Class>Match()` methods that allow deep object comparison.

## 8 Overall Design and Implementation Notes

1. **Deep and shallow classes.** The deep and shallow classes described in Section 5.2 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

# Command Line Tools

The main product delivered by ESMF is the ESMF library that allows application developers to write programs based on the ESMF API. In addition to the programming library, ESMF distributions come with a small set of command line tools (CLT) that are of general interest to the community. These CLTs utilize the ESMF library to implement features such as printing general information about the ESMF installation, or generating regrid weight files. The provided ESMF CLTs are intended to be used as standard command line tools.

The bundled ESMF CLTs are built and installed during the usual ESMF installation process, which is described in detail in the ESMF User's Guide section "Building and Installing the ESMF". After installation, the CLTs will be located in the `ESMF_APPSDIR` directory, which can be found as a Makefile variable in the `esmf.mk` file. The `esmf.mk` file can be found in the `ESMF_INSTALL_LIBDIR` directory after a successful installation. The ESMF User's Guide discusses the `esmf.mk` mechanism to access the bundled CLTs in more detail in section "Using Bundled ESMF Command Line Tools".

The following sections provide in-depth documentation of the bundled ESMF CLTs. In addition, each tool supports the standard `--help` command line argument, providing a brief description of how to invoke the program.

## Part III

# Superstructure

# 9 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.

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

## 9.1 Superstructure Classes

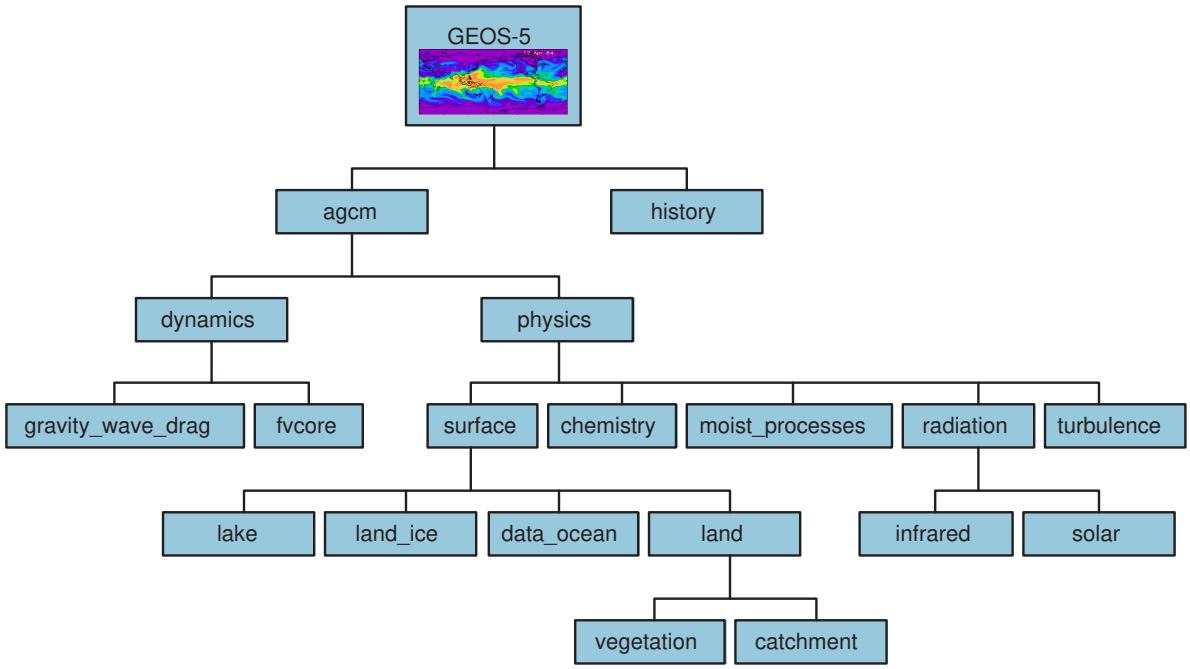
There are a small number of classes in the ESMF superstructure:

- **Component** An ESMF component has two parts, one that is supplied by 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 makes available to other Components.

Figure 2: ESMF enables applications such as the atmospheric general circulation model GEOS-5 to be structured hierarchically, and reconfigured and extended easily. Each box in this diagram is an ESMF Gridded Component.



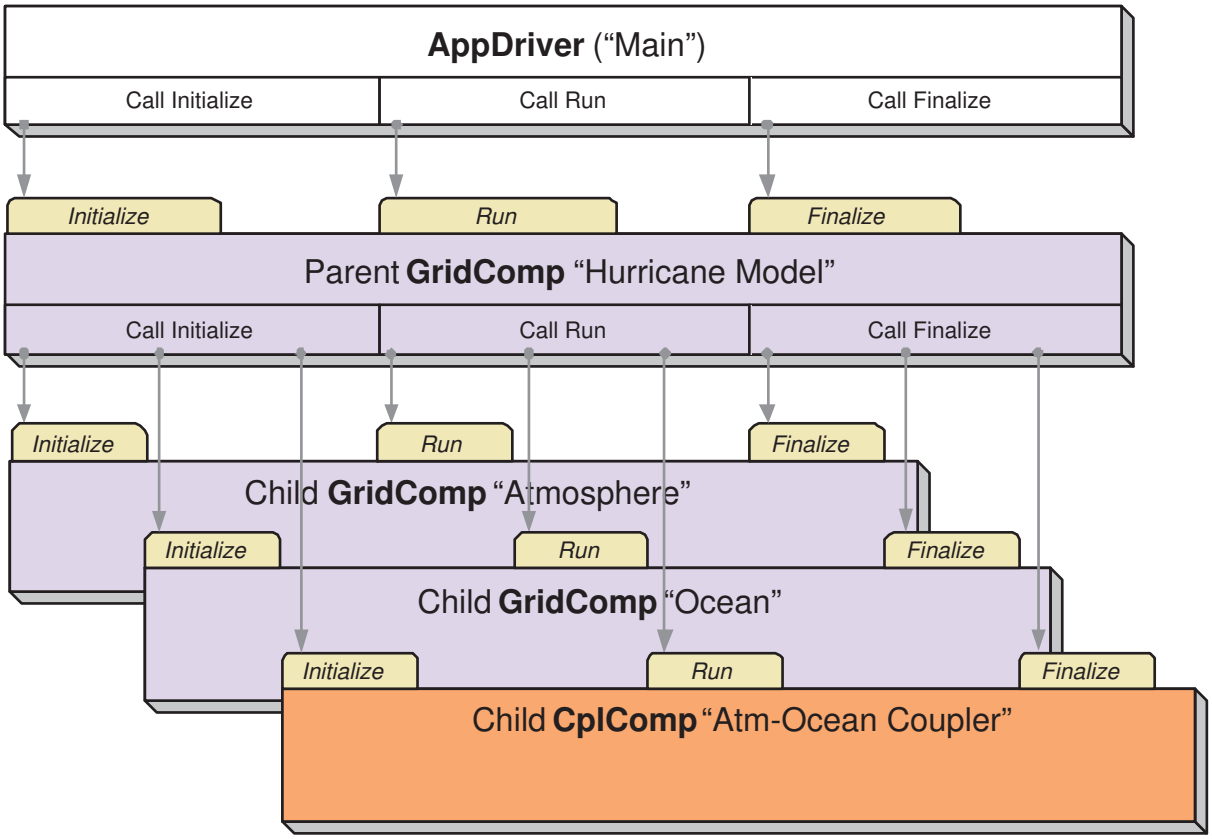

An ESMF coupled application typically involves a parent Gridded Component, two or more child Gridded Components and one or more Coupler Components.

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 application's "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 application, 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.

## 9.2 Hierarchical Creation of Components

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

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




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

## 9.3 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.

## 9.4 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.

## 9.5 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 superset 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.

## 9.6 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 application driver, Coupler, and all Gridded Components are distributed over nine PETs.

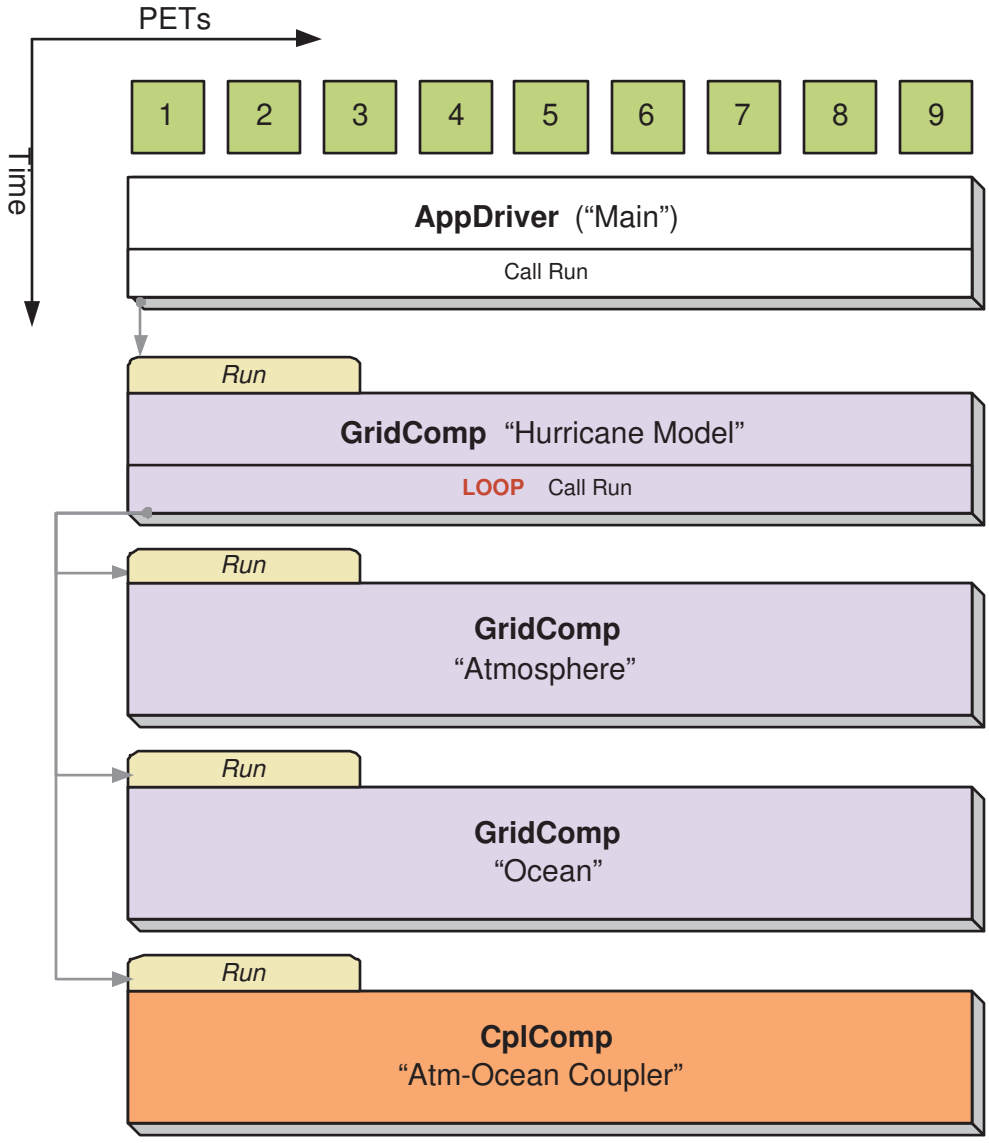


Figure 5: Schematic of the run method of a coupled application, with an "Atmosphere" and an "Ocean" Gridded Component running concurrently with an "Atm-Ocean Coupler." The top-level "Hurricane Model" Gridded Component contains the sequencing information and time advancement loop. The application driver, 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.

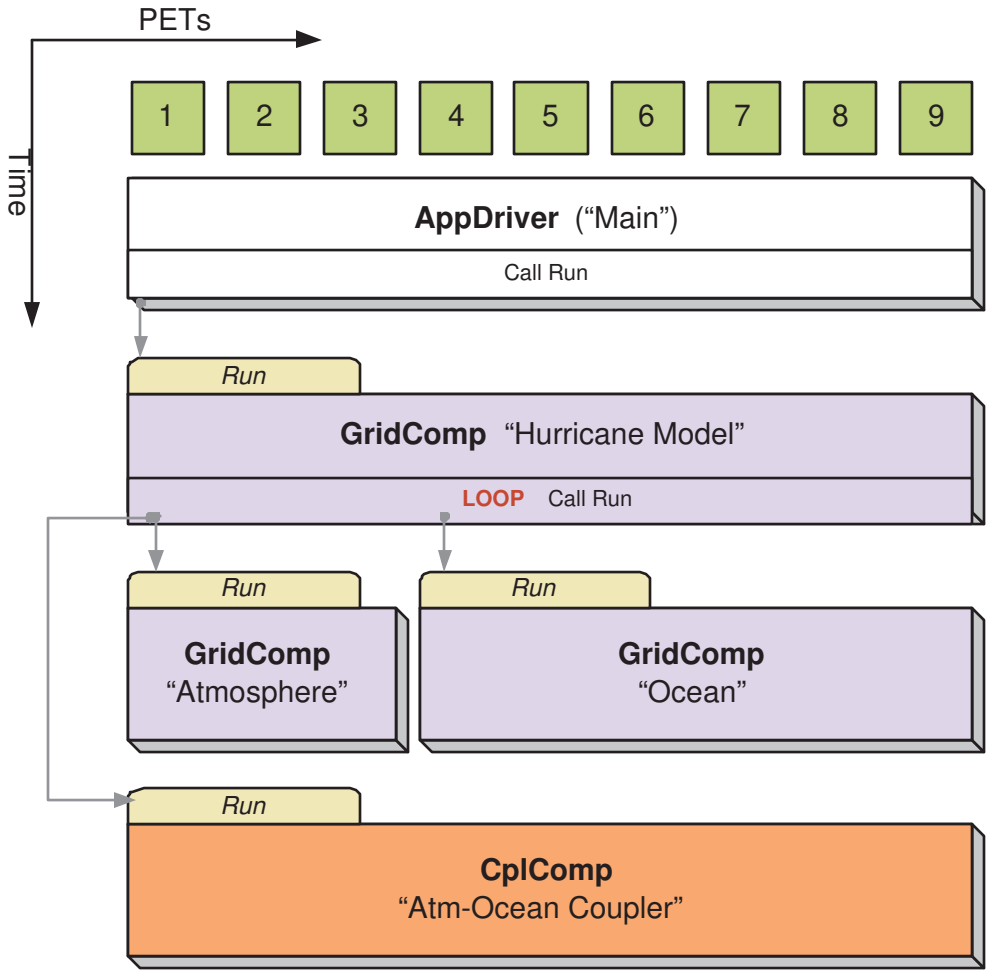
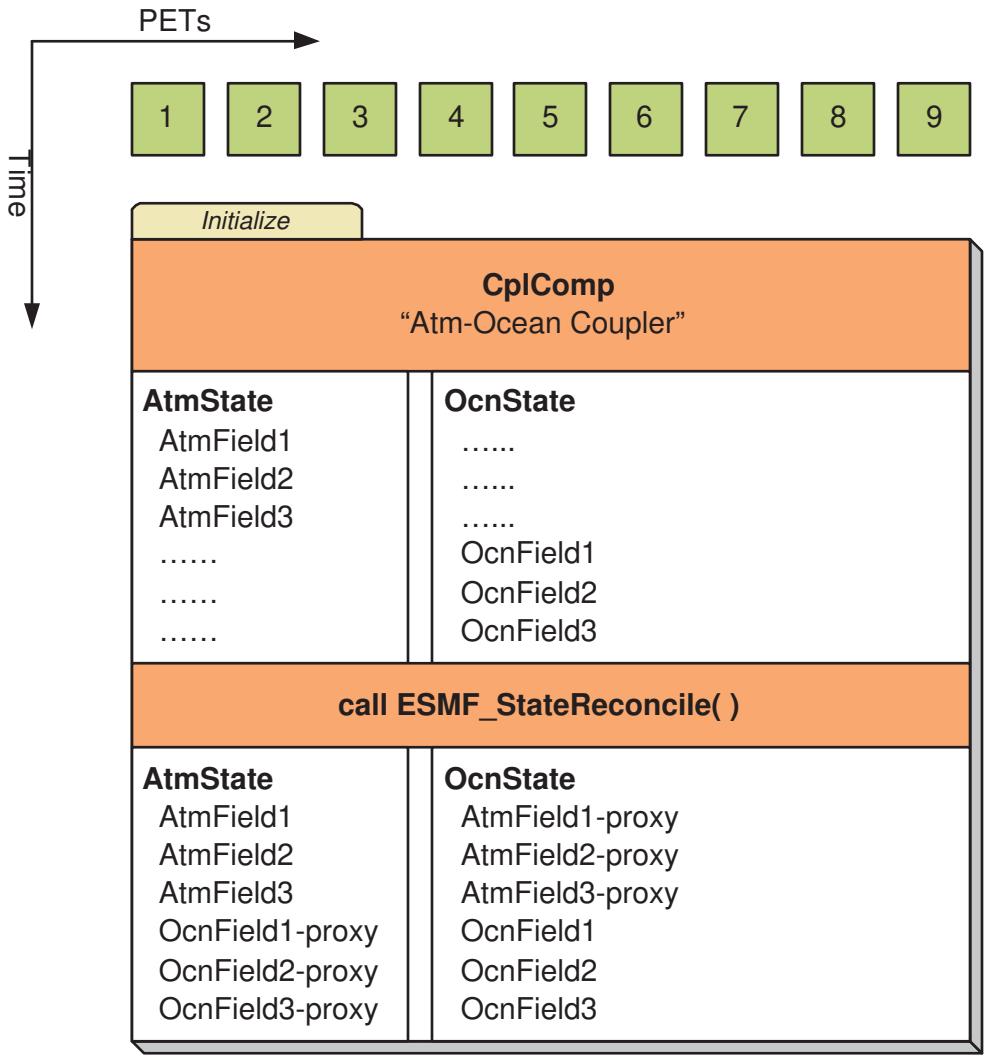
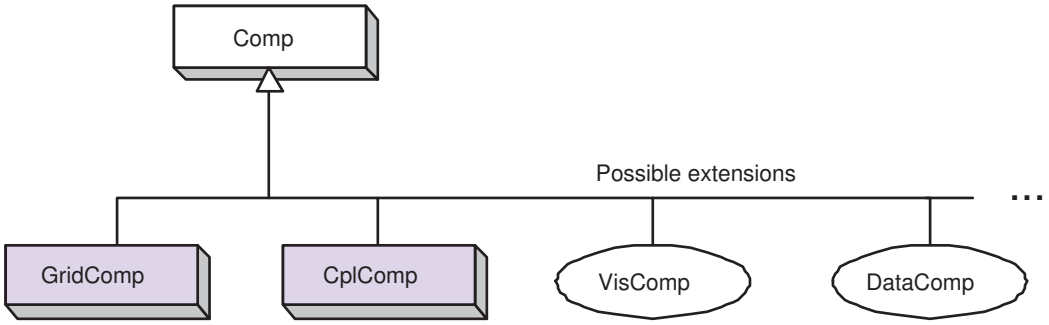


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



### 9.7 Object Model

The following is a simplified Unified Modeling Language (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.




## 10 Application Driver and Required ESMF Methods

### 10.1 Description

Every ESMF application needs a driver code. Typically the driver layer is implemented as the "main" of the application, although this is not strictly an ESMF requirement. For most ESMF applications the task of the application driver will be very generic: Initialize ESMF, create a top-level Component and call its Initialize, Run and Finalize methods, before destroying the top-level Component again and calling ESMF Finalize.

ESMF provides a number of different application driver templates in the `$ESMF_DIR/src/Superstructure/AppDriver` directory. An appropriate one can be chosen depending on how the application is to be structured:

**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. This means that the Coupler Component must be linked into all of the executables.

## 10.2 Required ESMF Methods

There are a few methods that every ESMF application must contain. First, `ESMC_Initialize()` and `ESMC_Finalize()` are in complete analogy to `MPI_Init()` and `MPI_Finalize()` known from MPI. All ESMF programs, serial or parallel, must initialize the ESMF system at the beginning, and finalize it at the end of execution. The behavior of calling any ESMF method before `ESMC_Initialize()`, or after `ESMC_Finalize()` is undefined.

Second, every ESMF Component that is accessed by an ESMF application requires that its set services routine is called through `ESMC_<Grid/Cpl>CompSetServices()`. The Component must implement one public entry point, its set services routine, that can be called through the `ESMC_<Grid/Cpl>CompSetServices()` library routine. The Component set services routine is responsible for setting entry points for the standard ESMF Component methods Initialize, Run, and Finalize.

Finally, the Component can optionally call `ESMC_<Grid/Cpl>CompSetVM()` *before* calling `ESMC_<Grid/Cpl>CompSetServices()`. Similar to `ESMC_<Grid/Cpl>CompSetServices()`, the
`ESMC_<Grid/Cpl>CompSetVM()` call requires a public entry point into the Component. It allows the Component to adjust certain aspects of its execution environment, i.e. its own VM, before it is started up.

The following sections discuss the above mentioned aspects in more detail.

### 10.2.1 ESMC_Initialize - Initialize ESMF

INTERFACE:

```
int ESMC_Initialize(
  int *rc,        // return code
```

```
...);           // optional arguments (see below)
```

*RETURN VALUE:*

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

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 `ESMC_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 `ESMC_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 MPICH1, to do I/O before the MPI environment is initialized. Please consult the documentation of your MPI implementation for details.

Optional arguments are recognised. To indicate the end of the optional argument list, `ESMC_ArgLast` must be used. A minimal call to `ESMC_Initialize()` would be:

```
ESMC_Initialize (NULL, ESMC_ArgLast);
```

The optional arguments are specified using the `ESMC_InitArg` macros. For example, to turn off logging so that no log files would be created, the `ESMC_Initialize()` call would be coded as:

```
ESMC_Initialize (&rc,
  ESMC_InitArgLogKindFlag(ESMC_LOGKIND_NONE),
  ESMC_ArgLast);
```

Before exiting the application the user must call `ESMC_Finalize()` to release resources and clean up the ESMF gracefully.

The arguments are:

**[rc]** Return code; equals `ESMF_SUCCESS` if there are no errors. `NULL` may be passed when the return code is not desired.

**[ESMC_InitArgDefaultCalKind(ARG)]** Macro specifying the default calendar kind for the entire application. Valid values for `ARG` are documented in section 26.2.1. If not specified, defaults to `ESMC_CALKIND_NOCALENDAR`.

**[ESMC_InitArgDefaultConfigFilename(ARG)]** Macro specifying the name of the default configuration file for the Config class. If not specified, no default file is used.

**[ESMC_InitArgLogFilename(ARG)]** Macro specifying the name used as part of the default log file name for the default log. If not specified, defaults to `ESMF_LogFile`.

**[ESMC_InitArgLogKindFlag(ARG)]** Macro specifying the default Log kind to be used by ESMF Log Manager. Valid values for `ARG` are documented in section 31.2.1. If not specified, defaults to `ESMC_LOGKIND_MULTI`.

**ESMC_ArgLast** Macro indicating the end of the optional argument list. This must be provided even when there are no optional arguments.

---

#### 10.2.2 ESMC_FinalizeWithFlag - Finalize the ESMF Framework and specify the type of finalization.

INTERFACE:

```
int ESMC_FinalizeWithFlag(
  ESMC_End_Flag endFlag); // enumerator for exit action (see below)
```

*RETURN VALUE:*

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

DESCRIPTION:

This must be called once on each PET before the application exits to allow ESMF to flush buffers, close open connections, and release internal resources cleanly.

The `endFlag` argument has one of three options:

**`ESMC_END_NORMAL`** Finalize normally.

**`ESMC_END_KEEPMPI`** Finalize normally without finalizing MPI.

**`ESMC_END_ABORT`** Abort on finalization.

---

#### 10.2.3 ESMC_Finalize - Finalize the ESMF Framework

INTERFACE:

```
int ESMC_Finalize(void);
```

*RETURN VALUE:*

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

DESCRIPTION:

This must be called once on each PET before the application exits to allow ESMF to flush buffers, close open connections, and release internal resources cleanly.

# 11 GridComp Class

## 11.1 Description

In Earth system modeling, the most natural way to think about an ESMF Gridded Component, or `ESMC_GridComp`, is as a piece of code representing a particular physical domain, such as 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 for things like 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 in and out of Gridded Components using an ESMF State, this is 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 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.

ESMF provides the Gridded Component derived type, `ESMC_GridComp`. An `ESMC_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 `ESMC_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 `ESMC_GridComp` derived type through a routine called `ESMC_SetServices()`. This is a routine that the user must write, and declare public. Inside the SetServices routine the user must call `ESMC_SetEntryPoint()` methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code.

## 11.2 Class API

### 11.2.1 ESMC_GridCompCreate - Create a Gridded Component

INTERFACE:

```
ESMC_GridComp ESMC_GridCompCreate(
  const char *name,                     // in
  const char *configFile,               // in
  ESMC_Clock clock,                     // in
  int *rc                               // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_GridComp object.
```

DESCRIPTION:

This interface creates an ESMC_GridComp object. By default, a separate VM context will be created for each component. This implies creating a new MPI communicator and allocating additional memory to manage the VM resources.

The arguments are:

**name** Name of the newly-created ESMC_GridComp.

**configFile** The filename of an ESMC_Config format file. If specified, this file is opened an ESMC_Config configuration object is created for the file, and attached to the new component.

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

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

---

### 11.2.2 ESMC_GridCompDestroy - Destroy a Gridded Component

INTERFACE:

```
int ESMC_GridCompDestroy(
  ESMC_GridComp *comp               // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this ESMC_GridComp.

The arguments are:

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

---

### 11.2.3 ESMC_GridCompFinalize - Finalize a Gridded Component

INTERFACE:

```
int ESMC_GridCompFinalize(
  ESMC_GridComp comp,           // inout
  ESMC_State importState,       // inout
  ESMC_State exportState,       // inout
  ESMC_Clock clock,             // in
  int phase,                    // in
  int *userRc                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call the associated user finalize code for a GridComp.

The arguments are:

**comp** `ESMC_GridComp` to call finalize routine for.

**importState** `ESMC_State` containing import data for coupling.

**exportState** `ESMC_State` containing export data for coupling.

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

**phase** Component providers must document whether 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 multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument must be 1.

**[userRc]** Return code set by `userRoutine` before returning.

---

### 11.2.4 ESMC_GridCompGetInternalState - Get the Internal State of a Gridded Component

INTERFACE:

```
void *ESMC_GridCompGetInternalState(
  ESMC_GridComp comp,           // in
  int *rc                       // out
);
```

*RETURN VALUE:*

```
    Pointer to private data block that is stored in the internal state.
```

DESCRIPTION:

Available to be called by an ESMC_GridComp at any time after ESMC_GridCompSetInternalState has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be 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 ESMC_GridComp, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding ESMC_GridCompSetInternalState call sets the data pointer to this block, and this call retrieves the data pointer.

Only the *last* data block set via ESMC_GridCompSetInternalState will be accessible.

The arguments are:

**comp** An ESMC_GridComp object.

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

---

### 11.2.5 ESMC_GridCompInitialize - Initialize a Gridded Component

INTERFACE:

```
int ESMC_GridCompInitialize(
  ESMC_GridComp comp,           // inout
  ESMC_State importState,       // inout
  ESMC_State exportState,       // inout
  ESMC_Clock clock,             // in
  int phase,                    // in
  int *userRc                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call the associated user initialization code for a GridComp.

The arguments are:

**comp** ESMC_GridComp to call initialize routine for.

**importState** ESMC_State containing import data for coupling.

**exportState** ESMC_State containing export data for coupling.

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

**phase** Component providers must document whether 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 multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument must be 1.

**[userRc]** Return code set by `userRoutine` before returning.

---

### 11.2.6 ESMC_GridCompPrint - Print the contents of a GridComp

INTERFACE:

```
int ESMC_GridCompPrint(
  ESMC_GridComp comp     // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints information about an `ESMC_GridComp` to `stdout`.

The arguments are:

**comp** An `ESMC_GridComp` object.

---

### 11.2.7 ESMC_GridCompRun - Run a Gridded Component

INTERFACE:

```
int ESMC_GridCompRun(
  ESMC_GridComp comp,           // inout
  ESMC_State importState,       // inout
  ESMC_State exportState,       // inout
  ESMC_Clock clock,             // in
  int phase,                    // in
  int *userRc                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call the associated user run code for a GridComp.

The arguments are:

**comp** `ESMC_GridComp` to call run routine for.

**importState** `ESMC_State` containing import data for coupling.

**exportState** `ESMC_State` containing export data for coupling.

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

**phase** Component providers must document whether 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 multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument must be 1.

**[userRc]** Return code set by `userRoutine` before returning.

---

### 11.2.8 ESMC_GridCompSetEntryPoint - Set user routine as entry point for standard Component method

INTERFACE:

```
int ESMC_GridCompSetEntryPoint(
  ESMC_GridComp comp,                                                  // in
  enum ESMC_Method method,                                             // in
  void (*userRoutine)                                                  // in
    (ESMC_GridComp, ESMC_State, ESMC_State, ESMC_Clock *, int *),
  int phase                                                            // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Registers a user-supplied `userRoutine` as the entry point for one of the predefined Component methods. After this call the `userRoutine` becomes accessible via the standard Component method API.

The arguments are:

**comp** An `ESMC_GridComp` object.

**method** One of a set of predefined Component methods - e.g. ESMF_METHOD_INITIALIZE, ESMF_METHOD_RUN, ESMF_METHOD_FINALIZE. See section 34.12 for a complete list of valid method options.

**userRoutine** The user-supplied subroutine to be associated for this Component method. This subroutine does not have to be public.

**phase** The phase number for multi-phase methods.

---

### 11.2.9 ESMC_GridCompSetInternalState - Set the Internal State of a Gridded Component

INTERFACE:

```
int ESMC_GridCompSetInternalState(
  ESMC_GridComp comp,           // inout
  void *data                    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Available to be called by an ESMC_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 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 ESMC_GridComp, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding ESMC_GridCompGetInternalState call retrieves the data pointer.

Only the *last* data block set via ESMC_GridCompSetInternalState will be accessible.

The arguments are:

**comp** An ESMC_GridComp object.

**data** Pointer to private data block to be stored.

---

### 11.2.10 ESMC_GridCompSetServices - Call user routine to register GridComp methods

INTERFACE:

```
int ESMC_GridCompSetServices(
  ESMC_GridComp comp,                           // in
  void (*userRoutine)(ESMC_GridComp, int *),    // in
  int *userRc                                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call into user provided `userRoutine` which is responsible for setting Component's Initialize(), Run() and Finalize() services.

The arguments are:

**comp** Gridded Component.

**userRoutine** Routine to be called.

**userRc** Return code set by `userRoutine` before returning.

The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument.

The `userRoutine`, when called by the framework, must make successive calls to `ESMC_GridCompSetEntryPoint()` to preset callback routines for standard Component Initialize(), Run() and Finalize() methods.

# 12 CplComp Class

## 12.1 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 11.1). A Coupler Component, or `ESMC_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 to a data assimilation system for numerical weather prediction in another. A single Coupler Component can couple two or more Gridded Components.

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. This portion of the Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.

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. For instance, ESMF does not currently offer tools for unit tranformations or time averaging operations, so users must manage those operations themselves.

The second part of a Coupler Component is the `ESMC_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. [1]

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

The user-written part of a Coupler Component is associated with an ESMC_CplComp derived type through a routine called ESMC_SetServices(). This is a routine that the user must write and declare public. Inside the ESMC_SetServices() routine the user must call ESMC_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.

## 12.2 Class API

### 12.2.1 ESMC_CplCompCreate - Create a Coupler Component

INTERFACE:

```
ESMC_CplComp ESMC_CplCompCreate(
  const char *name,                   // in
  const char *configFile,             // in
  ESMC_Clock clock,                   // in
  int *rc                             // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_CplComp object.
```

DESCRIPTION:

This interface creates an ESMC_CplComp object. By default, a separate VM context will be created for each component. This implies creating a new MPI communicator and allocating additional memory to manage the VM resources.

The arguments are:

**name** Name of the newly-created ESMC_CplComp.

**configFile** The filename of an ESMC_Config format file. If specified, this file is opened an ESMC_Config configuration object is created for the file, and attached to the new component.

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

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

---

### 12.2.2 ESMC_CplCompDestroy - Destroy a Coupler Component

INTERFACE:

```
int ESMC_CplCompDestroy(
  ESMC_CplComp *comp              // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this `ESMC_CplComp`.

The arguments are:

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

---

### 12.2.3 ESMC_CplCompFinalize - Finalize a Coupler Component

INTERFACE:

```
int ESMC_CplCompFinalize(
  ESMC_CplComp comp,            // inout
  ESMC_State importState,       // inout
  ESMC_State exportState,       // inout
  ESMC_Clock clock,             // in
  int phase,                    // in
  int *userRc                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call the associated user finalize code for a CplComp.

The arguments are:

**comp** `ESMC_CplComp` to call finalize routine for.

**importState** `ESMC_State` containing import data for coupling.

**exportState** `ESMC_State` containing export data for coupling.

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

**phase** Component providers must document whether 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 multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument must be 1.

**[userRc]** Return code set by `userRoutine` before returning.

---

### 12.2.4 ESMC_CplCompGetInternalState - Get the internal State of a Coupler Component

INTERFACE:

```
void *ESMC_CplCompGetInternalState(
  ESMC_CplComp comp,            //in
  int *rc                       // out
);
```

*RETURN VALUE:*

```
    Pointer to private data block that is stored in the internal state.
```

DESCRIPTION:

Available to be called by an `ESMC_CplComp` at any time after `ESMC_CplCompSetInternalState` has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be 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 `ESMC_CplComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMC_CplCompSetInternalState` call sets the data pointer to this block, and this call retrieves the data pointer.

Only the *last* data block set via `ESMC_CplCompSetInternalState` will be accessible.

The arguments are:

**comp** An `ESMC_CplComp` object.

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

---

### 12.2.5 ESMC_CplCompInitialize - Initialize a Coupler Component

INTERFACE:

```
int ESMC_CplCompInitialize(
  ESMC_CplComp comp,            // inout
  ESMC_State importState,       // inout
  ESMC_State exportState,       // inout
  ESMC_Clock clock,             // in
  int phase,                    // in
  int *userRc                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call the associated user initialize code for a CplComp.

The arguments are:

**comp** `ESMC_CplComp` to call initialize routine for.

**importState** `ESMC_State` containing import data for coupling.

**exportState** `ESMC_State` containing export data for coupling.

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

**phase** Component providers must document whether 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 multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument must be 1.

**[userRc]** Return code set by `userRoutine` before returning.

---

### 12.2.6 ESMC_CplCompPrint - Print a Coupler Component

INTERFACE:

```
 int ESMC_CplCompPrint(
   ESMC_CplComp comp       // in
 );
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints information about an `ESMC_CplComp` to `stdout`.

The arguments are:

**comp** An `ESMC_CplComp` object.

---

### 12.2.7 ESMC_CplCompRun - Run a Coupler Component

INTERFACE:

```
int ESMC_CplCompRun(
  ESMC_CplComp comp,            // inout
  ESMC_State importState,       // inout
  ESMC_State exportState,       // inout
  ESMC_Clock clock,             // in
  int phase,                    // in
  int *userRc                   // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call the associated user run code for a CplComp.

The arguments are:

**comp** `ESMC_CplComp` to call run routine for.

**importState** `ESMC_State` containing import data for coupling.

**exportState** `ESMC_State` containing export data for coupling.

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

**phase** Component providers must document whether 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 multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument must be 1.

**[userRc]** Return code set by `userRoutine` before returning.

---

### 12.2.8 ESMC_CplCompSetEntryPoint - Set the Entry point of a Coupler Component

INTERFACE:

```
int ESMC_CplCompSetEntryPoint(
  ESMC_CplComp comp,                                        // in
  enum ESMC_Method method,                                  // in
```

```
  void (*userRoutine)                                                  // in
    (ESMC_CplComp, ESMC_State, ESMC_State, ESMC_Clock *, int *),
  int phase                                                            // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Registers a user-supplied `userRoutine` as the entry point for one of the predefined Component methods. After this call the `userRoutine` becomes accessible via the standard Component method API.

The arguments are:

**comp** An `ESMC_CplComp` object.

**method** One of a set of predefined Component methods - e.g. `ESMF_METHOD_INITIALIZE`, `ESMF_METHOD_RUN`, `ESMF_METHOD_FINALIZE`. See section 34.12 for a complete list of valid method options.

**userRoutine** The user-supplied subroutine to be associated for this Component `method`. This subroutine does not have to be public.

**phase** The phase number for multi-phase methods.

---

### 12.2.9 ESMC_CplCompSetInternalState - Set the internal State of a Coupler Component

INTERFACE:

```
int ESMC_CplCompSetInternalState(
  ESMC_CplComp comp,            // inout
  void *data                    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Available to be called by an `ESMC_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 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 `ESMC_CplComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMC_CplCompGetInternalState` call retrieves the data pointer.

Only the *last* data block set via `ESMC_CplCompSetInternalState` will be accessible.

The arguments are:

**comp** An ESMC_CplComp object.

**data** Pointer to private data block to be stored.

---

#### 12.2.10 ESMC_CplCompSetServices - Destroy a Coupler Component

INTERFACE:

```
int ESMC_CplCompSetServices(
  ESMC_CplComp comp,                                // in
  void (*userRoutine)(ESMC_CplComp, int *),         // in
  int *userRc                                       // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Call into user provided userRoutine which is responsible for setting Component's Initialize(), Run() and Finalize() services.

The arguments are:

**comp** Gridded Component.

**userRoutine** Routine to be called.

**userRc** Return code set by userRoutine before returning.

The Component writer must supply a subroutine with the exact interface shown above for the userRoutine argument.

The userRoutine, when called by the framework, must make successive calls to ESMC_CplCompSetEntryPoint() to preset callback routines for standard Component Initialize(), Run() and Finalize() methods.

# 13 SciComp Class

## 13.1 Description

In Earth system modeling, a particular piece of code representing a physical domain, such as an atmospheric model or an ocean model, is typically implemented as an ESMF Gridded Component, or ESMC_GridComp. However, there are times when physical domains, or realms, need to be represented, but aren't actual pieces of code, or software. These domains can be implemented as ESMF Science Components, or ESMC_SciComp.

Unlike Gridded and Coupler Components, Science Components are not associated with software; they don't include execution routines such as initialize, run and finalize.

## 13.2 Class API

### 13.2.1 ESMC_SciCompCreate - Create a Science Component

INTERFACE:

```
ESMC_SciComp ESMC_SciCompCreate(
  const char *name,                     // in
  int *rc                               // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_SciComp object.
```

DESCRIPTION:

This interface creates an `ESMC_SciComp` object.

The arguments are:

**name** Name of the newly-created `ESMC_SciComp`.

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

---

### 13.2.2 ESMC_SciCompDestroy - Destroy a Science Component

INTERFACE:

```
int ESMC_SciCompDestroy(
  ESMC_SciComp *comp                // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this `ESMC_SciComp`.

The arguments are:

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

---

#### 13.2.3 ESMC_SciCompPrint - Print the contents of a SciComp

INTERFACE:

```
int ESMC_SciCompPrint(
  ESMC_SciComp comp     // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints information about an `ESMC_SciComp` to `stdout`.

The arguments are:

**comp** An `ESMC_SciComp` object.

# 14 State Class

## 14.1 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. However, the current C API only provides State access to Arrays, Fields and nested States. States cannot directly contain native language arrays (i.e. Fortran or C style 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 method, called `ESMF_StateReconcile()`, is provided by ESMF to broadcast information about objects which were created in sub-components. Currently this method is only available through the ESMF Fortran API. Hence the Coupler Component reponsible for reconciling States from Component that execute on subsets of PETs must be written in Fortran.

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

## 14.2 Restrictions and Future Work

1. **No synchronization of object IDs at object create time.** Object IDs are used 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.

One important request by the user community during the ESMF object design was that there be no communication overhead or synchronization when creating distributed ESMF objects. As a consequence it is required to create these objects in **unison** across all PETs in order to keep the ESMF object identification in sync.

## 14.3 Class API

### 14.3.1 ESMC_StateAddArray - Add an Array object to a State

INTERFACE:

```
int ESMC_StateAddArray(
  ESMC_State state,   // in
  ESMC_Array array    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Add an Array object to a `ESMC_State` object.

The arguments are:

**state** The State object.

**array** The Array object to be included within the State.

---

### 14.3.2 ESMC_StateAddField - Add a Field object to a State

INTERFACE:

```
int ESMC_StateAddField(
  ESMC_State state,   // in
  ESMC_Field field    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Add an Array object to a `ESMC_State` object.

The arguments are:

**state** The State object.

**array** The Array object to be included within the State.

---

#### 14.3.3 ESMC_StateCreate - Create an Array

INTERFACE:

```
ESMC_State ESMC_StateCreate(
  const char *name,  // in
  int *rc            // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_State object.
```

DESCRIPTION:

Create an `ESMC_State` object.

The arguments are:

**[name]** The name for the State object. If not specified, i.e. NULL, a default unique name will be generated: "StateNNN" where NNN is a unique sequence number from 001 to 999.

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

---

#### 14.3.4 ESMC_StateDestroy - Destroy a State

INTERFACE:

```
int ESMC_StateDestroy(
  ESMC_State *state    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Destroy a `ESMC_State` object.

The arguments are:

**state** The State to be destroyed.

---

### 14.3.5 ESMC_StateGetArray - Obtains an Array object from a State

INTERFACE:

```
int ESMC_StateGetArray(
  ESMC_State state,    // in
  const char *name,    // in
  ESMC_Array *array    // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Obtain a pointer to an `ESMC_Array` object contained within a State.

The arguments are:

**state** The State object.

**name** The name of the desired Array object.

**array** A pointer to the Array object.

---

### 14.3.6 ESMC_StateGetField - Obtains a Field object from a State

INTERFACE:

```
int ESMC_StateGetField(
  ESMC_State state,    // in
  const char *name,    // in
  ESMC_Field *field    // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Obtain a pointer to a `ESMC_Field` object contained within a State.

The arguments are:

**state** The State object.

**name** The name of the desired Field object.

**array** A pointer to the Field object.

---

### 14.3.7 ESMC_StatePrint - Print the contents of a State

INTERFACE:

```
int ESMC_StatePrint(
  ESMC_State state    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints the contents of a `ESMC_State` object.

The arguments are:

**state** The State to be printed.

**Part IV**

# Infrastructure: Fields and Grids

# 15 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 data array and the most complex data class represents a bundle of physical fields that are discretized on the same grid. However, the current C API does not support bundled data structures yet. Array and Field are the two data classes offered by the ESMF C language binding. Data class methods are called both from user-written code and from other classes internal to the framework.

Data classes are distributed over **DE**s, or **Decomposition Elements**. A DE represents a piece of a decomposition. A DELayout is a collection of DEs with some associated connectivity that describes a specific distribution. For example, the distribution of a grid divided into four segments in the x-dimension would be expressed in ESMF as a DELayout with four DEs lying along an x-axis. This abstract concept enables a data decomposition to be defined in terms of threads, MPI processes, virtual decomposition elements, or combinations of these without changes to user code. This is a primary strategy for ensuring optimal performance and portability for codes using the ESMF for communications.

ESMF data classes 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.

## 15.1 Infrastructure Data Classes

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

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

  Data elements in Arrays are partitioned into categories defined by the role the data element plays in distributed halo operations. Haloing - sometimes called ghosting - is the practice of copying portions of array data to multiple memory locations to ensure that data dependencies can be satisfied quickly when performing a calculation.

ESMF Arrays contain an **exclusive** domain, which contains data elements updated exclusively and definitively by a given DE; a **computational** domain, which contains all data elements with values that are updated by the DE in computations; and a **total** domain, which includes both the computational domain and data elements from other DEs which may be read but are not updated in computations.

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

## 15.2 Design and Implementation Notes

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

# 16 Field Class

## 16.1 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 and discretized field data, a reference to its associated grid, and metadata. The Field class stores the grid *staggering* for that physical field. This is the relationship of how the data array of a field 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, etc.). 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.

Field communication capabilities include: data redistribution, regridding, scatter, gather, sparse-matrix multiplication, and halo update. These are discussed in more detail in the documentation for the specific method calls. ESMF does not currently support vector fields, so the components of a vector field must be stored as separate Field objects.

## 16.2 Constants

### 16.2.1 ESMC_REGRIDMETHOD

DESCRIPTION:
Specify which interpolation method to use during regridding.

The type of this flag is:

```
type(ESMC_RegridMethod_Flag)
```

The valid values are:

**ESMC_REGRIDMETHOD_BILINEAR** Bilinear interpolation. Destination value is a linear combination of the source values in the cell which contains the destination point. The weights for the linear combination are based on the distance of destination point from each source value.

**ESMC_REGRIDMETHOD_PATCH** Higher-order patch recovery interpolation. Destination value is a weighted average of 2D polynomial patches constructed from cells surrounding the source cell which contains the destination point. This method typically results in better approximations to values and derivatives than bilinear. However, because of its larger stencil, it also results in a much larger interpolation matrix (and thus routeHandle) than the bilinear.

**ESMC_REGRIDMETHOD_NEAREST_STOD** In this version of nearest neighbor interpolation each destination point is mapped to the closest source point. A given source point may go to multiple destination points, but no destination point will receive input from more than one source point.

**ESMC_REGRIDMETHOD_NEAREST_DTOS** In this version of nearest neighbor interpolation each source point is mapped to the closest destination point. A given destination point may receive input from multiple source points, but no source point will go to more than one destination point.

**ESMC_REGRIDMETHOD_CONSERVE** First-order conservative interpolation. The main purpose of this method is to preserve the integral of the field between the source and destination. Will typically give a less accurate approximation to the individual field values than the bilinear or patch methods. The value of a destination cell

is calculated as the weighted sum of the values of the source cells that it overlaps. The weights are determined by the amount the source cell overlaps the destination cell. Needs corner coordinate values to be provided in the Grid. Currently only works for Fields created on the Grid center stagger or the Mesh element location.

**ESMC_REGRIDMETHOD_CONSERVE_2ND** Second-order conservative interpolation. As with first-order, preserves the integral of the value between the source and destination. However, typically produces a smoother more accurate result than first-order. Also like first-order, the value of a destination cell is calculated as the weighted sum of the values of the source cells that it overlaps. However, second-order also includes additional terms to take into account the gradient of the field across the source cell. Needs corner coordinate values to be provided in the Grid. Currently only works for Fields created on the Grid center stagger or the Mesh element location.

## 16.3 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.

### 16.3.1 Field create and destroy

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 `ESMC_FieldCreate()` routines require a Mesh object as input. The Mesh 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 depend on which of the variants of the `ESMC_FieldCreate()` call is used.

When finished with an `ESMC_Field`, the `ESMC_FieldDestroy` method removes it. However, the objects inside the `ESMC_Field` created externally should be destroyed separately, since objects can be added to more than one `ESMC_Field`. For example, the same `ESMF_Mesh` can be referenced by multiple `ESMC_Field`s. In this case the internal Mesh is not deleted by the `ESMC_FieldDestroy` call.

## 16.4 Class API

### 16.4.1 ESMC_FieldCreateGridArraySpec - Create a Field from Grid and ArraySpec

INTERFACE:

```
ESMC_Field ESMC_FieldCreateGridArraySpec(
  ESMC_Grid grid,                           // in
```

```
  ESMC_ArraySpec arrayspec,                 // in
  enum ESMC_StaggerLoc staggerloc,          // in
  ESMC_InterArrayInt *gridToFieldMap,       // in
  ESMC_InterArrayInt *ungriddedLBound,      // in
  ESMC_InterArrayInt *ungriddedUBound,      // in
  const char *name,                         // in
  int *rc                                   // out
);
```

*RETURN VALUE:*

```
Newly created ESMC_Field object.
```

## DESCRIPTION:

Creates a `ESMC_Field` object.

The arguments are:

**grid** A `ESMC_Grid` object.

**arrayspec** A `ESMC_ArraySpec` object describing data type and kind specification.

**staggerloc** Stagger location of data in grid cells. 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. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

**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.

**[name]** The name for the newly created field. If not specified, i.e. NULL, a default unique name will be generated: "FieldNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 16.4.2 ESMC_FieldCreateGridTypeKind - Create a Field from Grid and typekind

INTERFACE:

```
ESMC_Field ESMC_FieldCreateGridTypeKind(
  ESMC_Grid grid,                              // in
  enum ESMC_TypeKind_Flag typekind,            // in
  enum ESMC_StaggerLoc staggerloc,             // in
  ESMC_InterArrayInt *gridToFieldMap,          // in
  ESMC_InterArrayInt *ungriddedLBound,         // in
  ESMC_InterArrayInt *ungriddedUBound,         // in
  const char *name,                            // in
  int *rc                                      // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Field object.
```

DESCRIPTION:

Creates a `ESMC_Field` object.

The arguments are:

**grid** A `ESMC_Grid` object.

**typekind** The ESMC_TypeKind_Flag that describes this Field data.

**staggerloc** Stagger location of data in grid cells. 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. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

**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.

**[name]** The name for the newly created field. If not specified, i.e. NULL, a default unique name will be generated: "FieldNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 16.4.3 ESMC_FieldCreateMeshArraySpec - Create a Field from Mesh and ArraySpec

INTERFACE:

```
ESMC_Field ESMC_FieldCreateMeshArraySpec(
  ESMC_Mesh mesh,                           // in
  ESMC_ArraySpec arrayspec,                 // in
  ESMC_InterArrayInt *gridToFieldMap,       // in
  ESMC_InterArrayInt *ungriddedLBound,      // in
  ESMC_InterArrayInt *ungriddedUBound,      // in
  const char *name,                         // in
  int *rc                                   // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Field object.
```

DESCRIPTION:

Creates a `ESMC_Field` object.

The arguments are:

**mesh** A `ESMC_Mesh` object.

**arrayspec** A `ESMC_ArraySpec` object describing data type and kind specification.

**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. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

**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.

**[name]** The name for the newly created field. If not specified, i.e. NULL, a default unique name will be generated: "FieldNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 16.4.4 ESMC_FieldCreateMeshTypeKind - Create a Field from Mesh and typekind

INTERFACE:

```
ESMC_Field ESMC_FieldCreateMeshTypeKind(
  ESMC_Mesh mesh,                              // in
  enum ESMC_TypeKind_Flag typekind,            // in
  enum ESMC_MeshLoc_Flag meshloc,              // in
  ESMC_InterArrayInt *gridToFieldMap,          // in
  ESMC_InterArrayInt *ungriddedLBound,         // in
  ESMC_InterArrayInt *ungriddedUBound,         // in
  const char *name,                            // in
  int *rc                                      // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Field object.
```

DESCRIPTION:

Creates a `ESMC_Field` object.

The arguments are:

**mesh** A `ESMC_Mesh` object.

**typekind** The ESMC_TypeKind_Flag that describes this Field data.

**meshloc** The ESMC_MeshLoc_Flag that describes this Field data.

**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. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

**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.

**[name]** The name for the newly created field. If not specified, i.e. NULL, a default unique name will be generated: "FieldNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 16.4.5 ESMC_FieldCreateLocStreamArraySpec - Create a Field from LocStream and ArraySpec

INTERFACE:

```
ESMC_Field ESMC_FieldCreateLocStreamArraySpec(
  ESMC_LocStream locstream,                 // in
  ESMC_ArraySpec arrayspec,                 // in
  ESMC_InterArrayInt *gridToFieldMap,       // in
  ESMC_InterArrayInt *ungriddedLBound,      // in
  ESMC_InterArrayInt *ungriddedUBound,      // in
  const char *name,                         // in
  int *rc                                   // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Field object.
```

DESCRIPTION:

Creates a `ESMC_Field` object.

The arguments are:

**locstream** A `ESMC_LocStream` object.

**arrayspec** A `ESMC_ArraySpec` object describing data type and kind specification.

**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. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

**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.

**[name]** The name for the newly created field. If not specified, i.e. NULL, a default unique name will be generated: "FieldNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 16.4.6 ESMC_FieldCreateLocStreamTypeKind - Create a Field from LocStream and typekind

INTERFACE:

```
ESMC_Field ESMC_FieldCreateLocStreamTypeKind(
  ESMC_LocStream locstream,                  // in
  enum ESMC_TypeKind_Flag typekind,          // in
  ESMC_InterArrayInt *gridToFieldMap,        // in
  ESMC_InterArrayInt *ungriddedLBound,       // in
  ESMC_InterArrayInt *ungriddedUBound,       // in
  const char *name,                          // in
  int *rc                                    // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Field object.
```

DESCRIPTION:

Creates a `ESMC_Field` object.

The arguments are:

**locstream** A `ESMC_LocStream` object.

**typekind** The ESMC_TypeKind_Flag that describes this Field data.

**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. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

**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.

**[name]** The name for the newly created field. If not specified, i.e. NULL, a default unique name will be generated: "FieldNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 16.4.7 ESMC_FieldDestroy - Destroy a Field

INTERFACE:

```
int ESMC_FieldDestroy(
  ESMC_Field *field     // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this `ESMC_Field`. Return code; equals `ESMF_SUCCESS` if there are no errors.

The arguments are:

**field** Destroy contents of this `ESMC_Field`.

---

### 16.4.8 ESMC_FieldGetArray - Get the internal Array stored in the Field

INTERFACE:

```
ESMC_Array ESMC_FieldGetArray(
  ESMC_Field field,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The ESMC_Array object stored in the ESMC_Field.
```

DESCRIPTION:

Get the internal Array stored in the `ESMC_Field`.

The arguments are:

**field** Get the internal Array stored in this `ESMC_Field`.

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

---

### 16.4.9 ESMC_FieldGetMesh - Get the internal Mesh stored in the Field

INTERFACE:

```
ESMC_Mesh ESMC_FieldGetMesh(
  ESMC_Field field,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The ESMC_Mesh object stored in the ESMC_Field.
```

DESCRIPTION:

Get the internal Mesh stored in the `ESMC_Field`.

The arguments are:

**field** Get the internal Mesh stored in this `ESMC_Field`.

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

---

### 16.4.10 ESMC_FieldGetPtr - Get the internal Fortran data pointer stored in the Field

INTERFACE:

```
void *ESMC_FieldGetPtr(
  ESMC_Field field,     // in
  int localDe,          // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The Fortran data pointer stored in the ESMC_Field.
```

DESCRIPTION:

Get the internal Fortran data pointer stored in the `ESMC_Field`.

The arguments are:

**field** Get the internal Fortran data pointer stored in this `ESMC_Field`.

**localDe** Local DE for which information is requested. `[0,..,localDeCount-1]`.

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

---

### 16.4.11 ESMC_FieldGetBounds - Get the Field bounds

INTERFACE:

```
int ESMC_FieldGetBounds(
  ESMC_Field field,      // in
  int *localDe,
  int *exclusiveLBound,
  int *exclusiveUBound,
  int rank
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Get the Field bounds from the `ESMC_Field`.

The arguments are:

**field** ESMC_Field whose bounds will be returned

**localDe** The local DE of the ESMC_Field (not implemented)

**exclusiveLBound** The exclusive lower bounds of the ESMC_Field

**exclusiveUBound** The exclusive upper bounds of the ESMC_Field

**rank** The rank of the ESMC_Field, to size the bounds arrays

---

#### 16.4.12 ESMC_FieldGetLocalDECount - Get the number of DEs in a Field associated with the local PET

INTERFACE:

```
int ESMC_FieldGetLocalDECount(
  ESMC_Field field,           // in
  int *localDECount           // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Get the number of DEs in the specified ESMC_Field object on the local PET.

The arguments are:

**field** Get the local DE count for this ESMC_Field.

**localDECount** The number of DEs in the Field on the local PET.

---

#### 16.4.13 ESMC_FieldPrint - Print the internal information of a Field

INTERFACE:

```
int ESMC_FieldPrint(
  ESMC_Field field      // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Print the internal information within this ESMC_Field.

The arguments are:

**field** Print contents of this ESMC_Field.

---

### 16.4.14 ESMC_FieldRegridGetArea - Get the area of the cells used for

conservative interpolation

INTERFACE:

```
int ESMC_FieldRegridGetArea(
  ESMC_Field field      // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

This subroutine gets the area of the cells used for conservative interpolation for the grid object associated with areaField and puts them into areaField. If created on a 2D Grid, it must be built on the ESMF_STAGGERLOC_CENTER stagger location. If created on a 3D Grid, it must be built on the ESMF_STAGGERLOC_CENTER_VCENTER stagger location. If created on a Mesh, it must be built on the ESMF_MESHLOC_ELEMENT mesh location.

The arguments are:

**areaField** The Field to put the area values in.

---

### 16.4.15 ESMC_FieldRegridStore - Precompute a Field regridding operation and return a RouteHandle

INTERFACE:

```
int ESMC_FieldRegridStore(
    ESMC_Field srcField,                          // in
    ESMC_Field dstField,                          // in
    ESMC_InterArrayInt *srcMaskValues,            // in
    ESMC_InterArrayInt *dstMaskValues,            // in
```

```
ESMC_RouteHandle *routehandle,                  // inout
enum ESMC_RegridMethod_Flag *regridmethod,      // in
enum ESMC_PoleMethod_Flag *polemethod,          // in
int *regridPoleNPnts,                           // in
enum ESMC_LineType_Flag *lineType,              // in
enum ESMC_NormType_Flag *normType,              // in
enum ESMC_Logical *vectorRegrid,                // in
enum ESMC_ExtrapMethod_Flag *extrapMethod,      // in
int *extrapNumSrcPnts,                          // in
float *extrapDistExponent,                      // in
int *extrapNumLevels,                           // in
enum ESMC_UnmappedAction_Flag *unmappedaction, // in
enum ESMC_Logical *ignoreDegenerate,            // in
int *srcTermProcessing,                         // in
double **factorList,                            // inout
int **factorIndexList,                          // inout
int *numFactors,                                // inout
ESMC_Field *srcFracField,                       // inout
ESMC_Field *dstFracField);                      // inout
```

*RETURN VALUE:*

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

DESCRIPTION:

Creates a sparse matrix operation (stored in routehandle) that contains the calculations and communications necessary to interpolate from srcField to dstField. The routehandle can then be used in the call ESMC_FieldRegrid() to interpolate between the Fields.

The arguments are:

**srcField** ESMC_Field with source data.

**dstField** ESMC_Field with destination data.

**srcMaskValues** List of values that indicate a source point should be masked out. If not specified, no masking will occur.

**dstMaskValues** List of values that indicate a destination point should be masked out. If not specified, no masking will occur.

**routehandle** The handle that implements the regrid, to be used in `ESMC_FieldRegrid()`.

**regridmethod** The type of interpolation. If not specified, defaults to `ESMF_REGRIDMETHOD_BILINEAR`.

**polemethod** Which type of artificial pole to construct on the source Grid for regridding. If not specified, defaults to `ESMF_POLEMETHOD_ALLAVG` for non-conservative regrid methods, and `ESMF_POLEMETHOD_NONE` for conservative methods. If not specified, defaults to `ESMC_POLEMETHOD_ALLAVG`.

**regridPoleNPnts** If `polemethod` is `ESMC_POLEMETHOD_NPNTAVG`. This parameter indicates how many points should be averaged over. Must be specified if `polemethod` is `ESMC_POLEMETHOD_NPNTAVG`.

**[lineType]** This argument controls the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated. As would be expected, this argument

is only applicable when srcField and dstField are built on grids which lie on the surface of a sphere. Section 34.8 shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section 34.8 has the defaults by line type.

**normType** This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with regridmethod=ESMF_REGRIDMETHOD_CONSERVE. If not specified normType defaults to ESMF_NORMTYPE_DSTAREA.

**[vectorRegrid]** If true, treat a single ungridded dimension in the source and destination Fields as the components of a vector. If true and there is more than one ungridded dimension in either the source or destination, then an error will be returned. Currently, only undistributed (vector) dimensions of size 2 are supported. In the vector dimension, the first entry is interpreted as the east component and the second as the north component. In addition, this functionality presently only works when both the source and destination Fields are build on a geometry (e.g. an ESMF Grid) with a spherical coordinate system (e.g. ESMF_COORDSYS_SPH_DEG). Also, this functionality is not currently supported with conservative regrid methods (e.g. regridmethod=ESMF_REGRIDMETHOD_CONSERVE). We expect these restrictions to be loosened over time as new requirements come in from users. For further information on this functionality, see the "Vector regridding" section in the ESMF Reference Manual for Fortran. If not specified, this argument defaults to false.

**[extrapMethod]** The type of extrapolation. Please see Section 34.3 for a list of valid options. If not specified, defaults to ESMC_EXTRAPMETHOD_NONE.

**[extrapNumSrcPnts]** The number of source points to use for the extrapolation methods that use more than one source point (e.g. ESMC_EXTRAPMETHOD_NEAREST_IDAVG). If not specified, defaults to 8.

**[extrapDistExponent]** The exponent to raise the distance to when calculating weights for the ESMC_EXTRAPMETHOD_NEAREST_IDAVG extrapolation method. A higher value reduces the influence of more distant points. If not specified, defaults to 2.0.

**[extrapNumLevels]** The number of levels to output for the extrapolation methods that fill levels (e.g. ESMC_EXTRAPMETHOD_CREEP). When a method is used that requires this, then an error will be returned if it is not specified.

**unmappedaction** Specifies what should happen if there are destination points that can't be mapped to a source cell. Options are ESMF_UNMAPPEDACTION_ERROR or ESMF_UNMAPPEDACTION_IGNORE. If not specified, defaults to ESMF_UNMAPPEDACTION_ERROR.

**[factorList]** The list of coefficients for a sparse matrix which interpolates from srcField to dstField. The array coming out of this variable is in the appropriate format to be used in other ESMF sparse matrix multiply calls, for example ESMC_FieldSMMStore(). The factorList array is allocated by the method and the user is responsible for deallocating it.

**[factorIndexList]** The indices for a sparse matrix which interpolates from srcField to dstField. This argument is a 2D array containing pairs of source and destination sequence indices corresponding to the coefficients in the factorList argument. The first dimension of factorIndexList is of size 2. factorIndexList(1,:) specifies the sequence index of the source element in the srcField. factorIndexList(2,:) specifies the sequence index of the destination element in the dstField. The second dimension of factorIndexList steps through the list of pairs, i.e. size(factorIndexList,2)==size(factorList). The array coming out of this variable is in the appropriate format to be used in other ESMF sparse matrix multiply calls, for example ESMC_FieldSMMStore(). The factorIndexList array is allocated by the method and the user is responsible for deallocating it.

**[numFactors]** The number of factors returned in factorList.

**[srcFracField]** The fraction of each source cell participating in the regridding. Only valid when regridmethod is ESMC_REGRIDMETHOD_CONSERVE. This Field needs to be created on the same location (e.g staggerloc) as the srcField.

**[dstFracField]** The fraction of each destination cell participating in the regridding. Only valid when regridmethod is ESMF_REGRIDMETHOD_CONSERVE. This Field needs to be created on the same location (e.g staggerloc) as the dstField.

---

### 16.4.16 ESMC_FieldRegridStoreFile - Precompute a Field regridding operation and return a RouteHandle

INTERFACE:

```
int ESMC_FieldRegridStoreFile(
    ESMC_Field srcField,                          // in
    ESMC_Field dstField,                          // in
    const char *filename,                         // in
    ESMC_InterArrayInt *srcMaskValues,            // in
    ESMC_InterArrayInt *dstMaskValues,            // in
    ESMC_RouteHandle *routehandle,                // inout
    enum ESMC_RegridMethod_Flag *regridmethod,    // in
    enum ESMC_PoleMethod_Flag *polemethod,        // in
    int *regridPoleNPnts,                         // in
    enum ESMC_LineType_Flag *lineType,            // in
    enum ESMC_NormType_Flag *normType,            // in
    enum ESMC_Logical *vectorRegrid,              // in
    enum ESMC_UnmappedAction_Flag *unmappedaction, // in
    enum ESMC_Logical *ignoreDegenerate,          // in
    int *srcTermProcessing,                       // in
    enum ESMC_Logical *create_rh,                 // in
    enum ESMC_FileMode_Flag *filemode,            // in
    const char *srcFile,                          // in
    const char *dstFile,                          // in
    enum ESMC_FileFormat_Flag *srcFileType,       // in
    enum ESMC_FileFormat_Flag *dstFileType,       // in
    enum ESMC_Logical *largeFileFlag,             // in
    ESMC_Field *srcFracField,                     // out
    ESMC_Field *dstFracField);                    // out
```

*RETURN VALUE:*

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

DESCRIPTION:

Creates a sparse matrix operation (stored in routehandle) that contains the calculations and communications necessary to interpolate from srcField to dstField. The routehandle can then be used in the call ESMC_FieldRegrid() to interpolate between the Fields. The weights will be output to the file with name `filename`.

The arguments are:

**srcField** ESMC_Field with source data.

**dstField** ESMC_Field with destination data.

**[filename]** The output filename for the factorList and factorIndexList.

**[srcMaskValues]** List of values that indicate a source point should be masked out. If not specified, no masking will occur.

**[dstMaskValues]** List of values that indicate a destination point should be masked out. If not specified, no masking will occur.

**[routehandle]** The handle that implements the regrid, to be used in `ESMC_FieldRegrid()`.

**[regridmethod]** The type of interpolation. If not specified, defaults to `ESMC_REGRIDMETHOD_BILINEAR`.

**[polemethod]** Which type of artificial pole to construct on the source Grid for regridding. If not specified, defaults to `ESMC_POLEMETHOD_ALLAVG` for non-conservative regrid methods, and `ESMC_POLEMETHOD_NONE` for conservative methods. If not specified, defaults to `ESMC_POLEMETHOD_ALLAVG`.

**[regridPoleNPnts]** If `polemethod` is `ESMC_POLEMETHOD_NPNTAVG`. This parameter indicates how many points should be averaged over. Must be specified if `polemethod` is `ESMC_POLEMETHOD_NPNTAVG`.

**[lineType]** This argument controls the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated. As would be expected, this argument is only applicable when `srcField` and `dstField` are built on grids which lie on the surface of a sphere. Section 34.8 shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section 34.8 has the defaults by line type.

**[normType]** This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with `regridmethod=ESMC_REGRIDMETHOD_CONSERVE`. If not specified normType defaults to `ESMC_NORMTYPE_DSTAREA`.

**[vectorRegrid]** If true, treat a single ungridded dimension in the source and destination Fields as the components of a vector. If true and there is more than one ungridded dimension in either the source or destination, then an error will be returned. Currently, only undistributed (vector) dimensions of size 2 are supported. In the vector dimension, the first entry is interpreted as the east component and the second as the north component. In addition, this functionality presently only works when both the source and destination Fields are build on a geometry (e.g. an ESMF Grid) with a spherical coordinate system (e.g. ESMF_COORDSYS_SPH_DEG). Also, this functionality is not currently supported with conservative regrid methods (e.g. `regridmethod=ESMF_REGRIDMETHOD_CONSERVE`). We expect these restrictions to be loosened over time as new requirements come in from users. For further information on this functionality, see the "Vector regridding" section in the ESMF Reference Manual for Fortran. If not specified, this argument defaults to false.

**[unmappedaction]** Specifies what should happen if there are destination points that can't be mapped to a source cell. Options are `ESMC_UNMAPPEDACTION_ERROR` or `ESMC_UNMAPPEDACTION_IGNORE`. If not specified, defaults to `ESMC_UNMAPPEDACTION_ERROR`.

**create_rh** Specifies whether or not to create a routehandle, or just write weights to file. If not specified, defaults to `ESMF_TRUE`.

**filemode** Specifies the mode to use when creating the weight file. Options are `ESMC_FILEMODE_BASIC` and `ESMC_FILEMODE_WITHAUX`, which will write a file that includes center coordinates of the grids. The default value is `ESMC_FILEMODE_BASIC`.

**srcFile** The name of the source file used to create the `ESMC_Grid` used in this regridding operation.

**dstFile** The name of the destination file used to create the `ESMC_Grid` used in this regridding operation.

**srcFileType** The type of the file used to represent the source grid.

**dstFileType** The type of the file used to represent the destination grid.

**[srcFracField]** The fraction of each source cell participating in the regridding. Only valid when regridmethod is ESMC_REGRIDMETHOD_CONSERVE. This Field needs to be created on the same location (e.g staggerloc) as the srcField.

**[dstFracField]** The fraction of each destination cell participating in the regridding. Only valid when regridmethod is ESMC_REGRIDMETHOD_CONSERVE. This Field needs to be created on the same location (e.g staggerloc) as the dstField.

---

### 16.4.17 ESMC_FieldRegrid - Compute a regridding operation

INTERFACE:

```
int ESMC_FieldRegrid(
  ESMC_Field srcField,                  // in
  ESMC_Field dstField,                  // inout
  ESMC_RouteHandle routehandle,         // in
  enum ESMC_Region_Flag *zeroregion,    // in
  ESMC_DynamicMask *dynamicMask);       // in
```

*RETURN VALUE:*

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

DESCRIPTION:

Execute the precomputed regrid operation stored in routehandle to interpolate from srcField to dstField. See ESMF_FieldRegridStore() on how to precompute the routehandle. It is erroneous to specify the identical Field object for srcField and dstField arguments. This call is collective across the current VM.

The arguments are:

**srcField** ESMC_Field with source data.

**dstField** ESMC_Field with destination data.

**routehandle** Handle to the precomputed Route.

**[zeroregion]** If set to ESMC_REGION_TOTAL *(default)* the total regions of all DEs in dstField will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to ESMC_REGION_EMPTY the elements in dstField will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting zeroregion to ESMC_REGION_SELECT will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication.

**[dynamicMask]** Dynamic mask object.

---

### 16.4.18 ESMC_FieldRegridRelease - Free resources used by a regridding operation

INTERFACE:

```
int ESMC_FieldRegridRelease(ESMC_RouteHandle *routehandle);  // inout
```

*RETURN VALUE:*

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

DESCRIPTION:

Free resources used by regrid object

The arguments are:

**routehandle** Handle carrying the sparse matrix

---

### 16.4.19 ESMC_FieldSMMStore - Precompute a Field regridding operation and return a RouteHandle

INTERFACE:

```
int ESMC_FieldSMMStore(
    ESMC_Field srcField,                          // in
    ESMC_Field dstField,                          // in
    const char *filename,                         // in
    ESMC_RouteHandle *routehandle,                // out
    enum ESMC_Logical *ignoreUnmatchedIndices,    // in
    int *srcTermProcessing,                       // in
    int *pipeLineDepth);                          // in
```

*RETURN VALUE:*

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

DESCRIPTION:

Creates a sparse matrix operation (stored in routehandle) that contains the calculations and communications necessary to interpolate from srcField to dstField. The routehandle can then be used in the call ESMC_FieldRegrid() to interpolate between the Fields.

The arguments are:

**srcField** ESMC_Field with source data.

**dstField** ESMC_Field with destination data.

**filename** Path to the file containing weights for creating an `ESMC_RouteHandle`. Only "row", "col", and "S" variables are required. They must be one-dimensionsal with dimension "n_s".

**routehandle** The handle that implements the regrid, to be used in `ESMC_FieldRegrid()`.

**[ignoreUnmatchedIndices]** A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcField` or `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

**[srcTermProcessing]** The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side. Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. The `ESMC_FieldSMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is $< 0$, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

**[pipelineDepth]** The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors. Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange. The `ESMC_FieldSMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is $< 0$, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

# 17 Array Class

## 17.1 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, 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 communication operations.

Like Fields, Arrays can be added to a State and used in inter-Component data communications.

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 the ESMF DistGrid. The Array class offers common communication patterns within the index space formalism.

## 17.2 Class API

### 17.2.1 ESMC_ArrayCreate - Create an Array

INTERFACE:

```
ESMC_Array ESMC_ArrayCreate(
  ESMC_ArraySpec arrayspec,   // in
  ESMC_DistGrid distgrid,     // in
  const char* name,           // in
  int *rc                     // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Array object.
```

DESCRIPTION:

Create an `ESMC_Array` object.

The arguments are:

**arrayspec** `ESMC_ArraySpec` object containing the type/kind/rank information.

**distgrid** `ESMC_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.

**[name]** The name for the Array object. If not specified, i.e. NULL, a default unique name will be generated: "ArrayNNN" where NNN is a unique sequence number from 001 to 999.

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

---

### 17.2.2 ESMC_ArrayDestroy - Destroy an Array

INTERFACE:

```
int ESMC_ArrayDestroy(
  ESMC_Array *array           // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Destroy an `ESMC_Array` object.

The arguments are:

**array** `ESMC_Array` object to be destroyed.

---

### 17.2.3 ESMC_ArrayGetName - Get the name of an Array

INTERFACE:

```
 const char *ESMC_ArrayGetName(
   ESMC_Array array,           // in
   int *rc                     // out
 );
```

*RETURN VALUE:*

```
    Pointer to the Array name string.
```

DESCRIPTION:

Get the name of the specified `ESMC_Array` object.

The arguments are:

**array** `ESMC_Array` object to be queried.

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

---

### 17.2.4 ESMC_ArrayGetLocalDECount - Get the number of DEs in an Array associated with the local PET

INTERFACE:

```
 int ESMC_ArrayGetLocalDECount(
   ESMC_Array array,           // in
   int *localDECount           // out
 );
```

*RETURN VALUE:*

Return code; equals {\tt ESMF_SUCCESS} if there are no errors.

DESCRIPTION:

Get the number of DEs in the specified ESMC_Array object on the local PET.

The arguments are:

**array** ESMC_Array object to be queried.

**localDECount** The number of DEs in the Array on the local PET.

---

### 17.2.5 ESMC_ArrayGetPtr - Get pointer to Array data.

INTERFACE:

```
void *ESMC_ArrayGetPtr(
  ESMC_Array array,           // in
  int localDe,                // in
  int *rc                     // out
);
```

*RETURN VALUE:*

Pointer to the Array data.

DESCRIPTION:

Get pointer to the data of the specified ESMC_Array object.

The arguments are:

**array** ESMC_Array object to be queried.

**localDe** Local De for which to data pointer is queried.

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

---

### 17.2.6 ESMC_ArrayPrint - Print an Array

INTERFACE:

```
int ESMC_ArrayPrint(
  ESMC_Array array            // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Print internal information of the specified `ESMC_Array` object.

The arguments are:

**array** `ESMC_Array` object to be printed.

## 17.3 Class API: DynamicMask Methods

### 17.3.1 ESMC_DynamicMaskPredefinedSetR8R8R8 - Set R8R8R8 dynamic mask

INTERFACE:

```
int ESMC_DynamicMaskPredefinedSetR8R8R8(
  ESMC_DynamicMask *dynamicMask,                // inout
  enum  ESMC_DynamicMaskPredef_Flag predefFlag, // in
  int *handleAllElements,                       // in
  ESMC_R8 *dynamicSrcMaskValue,                 // in
  ESMC_R8 *dynamicDstMaskValue                  // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Set R8R8R8 dynamic mask with predfined dynamic mask routine.

The arguments are:

**dynamicMask** The `ESMC_DynamicMask` to set.

**predefFlag** Predefined dynamic mask routine.

**handleAllElements** Make all elements available to the dynamic mask routine.

**dynamicSrcMaskValue** Source element mask value.

**dynamicDstMaskValue** Destination element mask value.

---

### 17.3.2 ESMC_DynamicMaskPredefinedSetR8R8R8V - Set R8R8R8V dynamic mask

INTERFACE:

```
int ESMC_DynamicMaskPredefinedSetR8R8R8V(
  ESMC_DynamicMask *dynamicMask,                // inout
  enum  ESMC_DynamicMaskPredef_Flag predefFlag, // in
  int *handleAllElements,                       // in
  ESMC_R8 *dynamicSrcMaskValue,                 // in
  ESMC_R8 *dynamicDstMaskValue                  // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Set R8R8R8V dynamic mask with predfined dynamic mask routine.

The arguments are:

**dynamicMask** The `ESMC_DynamicMask` to set.

**predefFlag** Predefined dynamic mask routine.

**handleAllElements** Make all elements available to the dynamic mask routine.

**dynamicSrcMaskValue** Source element mask value.

**dynamicDstMaskValue** Destination element mask value.

---

### 17.3.3 ESMC_DynamicMaskPredefinedSetR4R8R4 - Set R4R8R4 dynamic mask

INTERFACE:

```
int ESMC_DynamicMaskPredefinedSetR4R8R4(
  ESMC_DynamicMask *dynamicMask,                // inout
  enum  ESMC_DynamicMaskPredef_Flag predefFlag, // in
  int *handleAllElements,                       // in
  ESMC_R4 *dynamicSrcMaskValue,                 // in
  ESMC_R4 *dynamicDstMaskValue                  // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Set R4R8R4 dynamic mask with predfined dynamic mask routine.

The arguments are:

**dynamicMask** The `ESMC_DynamicMask` to set.

**predefFlag** Predefined dynamic mask routine.

**handleAllElements** Make all elements available to the dynamic mask routine.

**dynamicSrcMaskValue** Source element mask value.

**dynamicDstMaskValue** Destination element mask value.

---

### 17.3.4 ESMC_DynamicMaskPredefinedSetR4R8R4V - Set R4R8R4V dynamic mask

INTERFACE:

```
int ESMC_DynamicMaskPredefinedSetR4R8R4V(
  ESMC_DynamicMask *dynamicMask,                // inout
  enum  ESMC_DynamicMaskPredef_Flag predefFlag, // in
  int *handleAllElements,                       // in
  ESMC_R4 *dynamicSrcMaskValue,                 // in
  ESMC_R4 *dynamicDstMaskValue                  // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Set R4R8R4V dynamic mask with predfined dynamic mask routine.

The arguments are:

**dynamicMask** The `ESMC_DynamicMask` to set.

**predefFlag** Predefined dynamic mask routine.

**handleAllElements** Make all elements available to the dynamic mask routine.

**dynamicSrcMaskValue** Source element mask value.

**dynamicDstMaskValue** Destination element mask value.

# 18 ArraySpec Class

## 18.1 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 information.

## 18.2 Class API

### 18.2.1 ESMC_ArraySpecGet - Get values from an ArraySpec

INTERFACE:

```
int ESMC_ArraySpecGet(
  ESMC_ArraySpec arrayspec,         // in
  int *rank,                        // out
  enum ESMC_TypeKind_Flag *typekind // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Returns information about the contents of an `ESMC_ArraySpec`.

The arguments are:

**arrayspec** The `ESMC_ArraySpec` to query.

**rank** Array rank (dimensionality - 1D, 2D, etc). Maximum allowed is 7D.

**typekind** Array typekind. See section 34.18 for valid values.

---

### 18.2.2 ESMC_ArraySpecSet - Set values for an ArraySpec

INTERFACE:

```
int ESMC_ArraySpecSet(
  ESMC_ArraySpec *arrayspec,         // inout
  int rank,                          // in
  enum ESMC_TypeKind_Flag typekind       // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Set an Array specification - typekind, and rank.

The arguments are:

**arrayspec** The `ESMC_ArraySpec` to set.

**rank** Array rank (dimensionality - 1D, 2D, etc). Maximum allowed is 7D.

**typekind** Array typekind. See section 34.18 for valid values.

# 19 Grid Class

## 19.1 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.

**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).
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.
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.

### 19.1.1 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. The DistGrid used in Grid creation describes the properties of the Grid cells. In addition to this one, the Grid internally creates DistGrids for each stagger location. These stagger DistGrids are related to the original DistGrid, but may contain extra padding to represent the extent of the index space of the stagger. These DistGrids are what are used when a Field is created on a Grid.

### 19.1.2 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 19.1.3.
- Types of distributions supported. ESMF supports regular, irregular, or arbitrary distributions of data. For more details see section 19.1.4.
- Types of coordinates supported. ESMF supports uniform, rectilinear, and curvilinear coordinates. For more details see section 19.1.5.

### 19.1.3 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. There are two sets of methods which allow the user to do this. These two sets of methods cover the same set of topologies, but allow the user to specify them in different ways.

The first set of these are a group of overloaded calls broken up by the number of periodic dimensions they specify. With these the user can pick the method which creates a Grid with the number of periodic dimensions they need, and then specify other connectivity options via arguments to the method. The following is a description of these methods:

**ESMF_GridCreateNoPeriDim()** Allows the user to create a Grid with no edge connections, for example, a regional Grid with closed boundaries.

**ESMF_GridCreate1PeriDim()** Allows the user to create a Grid with 1 periodic dimension and supports a range of options for what to do at the pole (see Section 19.2.4. Some examples of Grids which can be created here are tripole spheres, bipole spheres, cylinders with open poles.

**ESMF_GridCreate2PeriDim()** Allows the user to create a Grid with 2 periodic dimensions, for example a torus, or a regional Grid with doubly periodic boundaries.

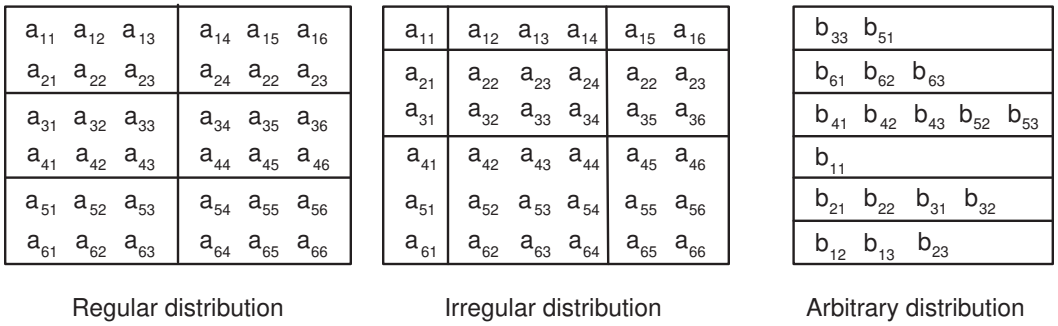



Figure 7: Examples of regular and irregular decomposition of a grid **a** that is 6x6, and an arbitrary decomposition of a grid **b** that is 6x3.

More detailed information can be found in the API description of each.

The second set of shortcut methods is a set of methods overloaded under the name `ESMF_GridCreate()`. These methods allow the user to specify the connectivites at the end of each dimension, by using the ESMF_GridConn_Flag flag. The table below shows the ESMF_GridConn_Flag settings used to create standard shapes in 2D using the ESMF_GridCreate() call. Two values are specified for each dimension, one for the low end and one for the high end of the dimension's index values.

| 2D Shape | **connflagDim1(1)** | **connflagDim1(2)** | **connflagDim2(1)** | **connflagDim2(2)** |
|---|---|---|---|---|
| **Rectangle** | NONE | NONE | NONE | NONE |
| **Bipole Sphere** | POLE | POLE | PERIODIC | PERIODIC |
| **Tripole Sphere** | POLE | BIPOLE | PERIODIC | PERIODIC |
| **Cylinder** | NONE | NONE | PERIODIC | PERIODIC |
| **Torus** | PERIODIC | PERIODIC | PERIODIC | PERIODIC |

If the user's grid shape is too complex for an ESMF shortcut routine, or involves more than three dimensions, a DistGrid can be created to specify the shape in detail. This DistGrid is then passed into a Grid create call.

### 19.1.4 Grid Distribution

ESMF Grids have several options for data distribution (also referred to as decomposition). As ESMF Grids are cell based, these options are all specified in terms of how the cells in the Grid are broken up between DEs.

The main distribution options are regular, irregular, and arbitrary. A **regular** distribution is one in which the same number of contiguous grid cells are assigned to each DE in the distributed dimension. An **irregular** distribution is one in which unequal numbers of contiguous grid cells are assigned to each DE in the distributed dimension. An **arbitrary** distribution is one in which any grid cell 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). Support for arbitrary distribution is limited in the current version of ESMF.

Figure 7 illustrates options for distribution.

A distribution can also be specified using the DistGrid, by passing object into a Grid create call.

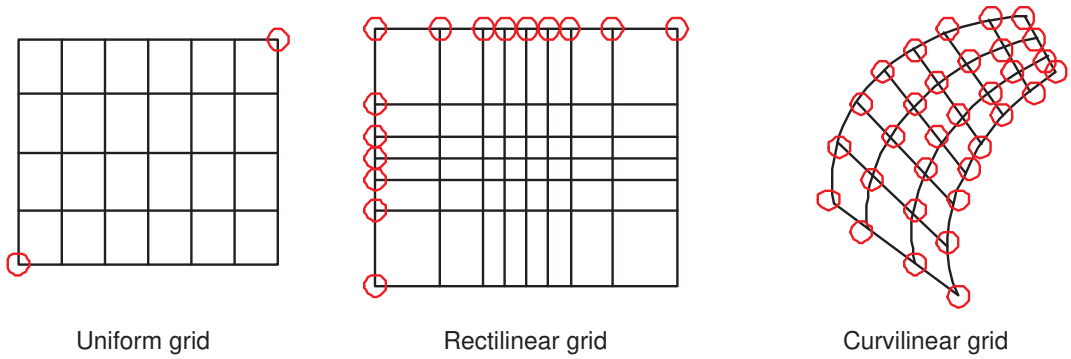



Figure 8: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

### 19.1.5 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 8 shows examples of each type of grid.

Each of these coordinate types can be set for each of the standard grid shapes described in section 19.1.3.

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.

| | **Uniform** | **Rectilinear** | **Curvilinear** |
|---|---|---|---|
| **Sphere** | Global uniform lat-lon grid | Gaussian grid | Displaced pole grid |
| **Rectangle** | Regional uniform lat-lon grid | Gaussian grid section | Polar stereographic grid section |

### 19.1.6 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.

### 19.1.7 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. The default stagger location in ESMF is the cell center, and cell counts in Grid are based on this assumption. Combinations

of the 2D ESMF stagger locations 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.

### 19.1.8 Masking

Masking is the process whereby parts of a grid can be marked to be ignored during an operation, such as regridding. Masking can be used on a source grid to indicate that certain portions of the grid should not be used to generate regridded data. This is useful, for example, if a portion of the source grid contains unusable values. Masking can also be used on a destination grid to indicate that the portion of the field built on that part of the Grid should not receive regridded data. This is useful, for example, when part of the grid isn't being used (e.g. the land portion of an ocean grid).

ESMF regrid currently supports masking for Fields built on structured Grids and element masking for Fields built on unstructured Meshes. The user may mask out points in the source Field or destination Field or both. To do masking the user sets mask information in the Grid or Mesh upon which the Fields passed into the `ESMC_FieldRegridStore()` call are built. The `srcMaskValues` and `dstMaskValues` arguments to that call can then be used to specify which values in that mask information indicate that a location should be masked out. For example, if `dstMaskValues` is set to (/1,2/), then any location that has a value of 1 or 2 in the mask information of the Grid or Mesh upon which the destination Field is built will be masked out.

Masking behavior differs slightly between regridding methods. For non-conservative regridding methods (e.g. bilinear or high-order patch), masking is done on points. For these methods, masking a destination point means that that point won't participate in regridding (e.g. won't be interpolated to). For these methods, masking a source point means that the entire source cell using that point is masked out. In other words, if any corner point making up a source cell is masked then the cell is masked. For conservative regridding methods (e.g. first-order conservative) masking is done on cells. Masking a destination cell means that the cell won't participate in regridding (e.g. won't be interpolated to). Similarly, masking a source cell means that the cell won't participate in regridding (e.g. won't be interpolated from). For any type of interpolation method (conservative or non-conservative) the masking is set on the location upon which the Fields passed into the regridding call are built. For example, if Fields built on `ESMC_STAGGERLOC_CENTER` are passed into the `ESMC_FieldRegridStore()` call then the masking should also be set on `ESMC_STAGGERLOC_CENTER`.

## 19.2 Constants

### 19.2.1 ESMC_COORDSYS

DESCRIPTION:
A set of values which indicates in which system the coordinates in the Grid are. This value is useful both to indicate to other users the type of the coordinates, but also to control how the coordinates are interpreted in regridding methods (e.g. `ESMC_FieldRegridStore()`).

The type of this flag is:

`type(ESMC_CoordSys_Flag)`

The valid values are:

**ESMC_COORDSYS_CART** Cartesian coordinate system. In this system, the cartesian coordinates are mapped to the Grid coordinate dimensions in the following order: x,y,z. (E.g. using `coordDim=2` in ESMC_GridGetCoord() references the y dimension)

**ESMC_COORDSYS_SPH_DEG** Spherical coordinates in degrees. In this system, the spherical coordinates are mapped to the Grid coordinate dimensions in the following order: longitude, latitude, radius. (E.g. using `coordDim=2` in ESMC_GridGetCoord() references the latitude dimension) Note, however, that ESMC_FieldRegridStore() currently just supports longitude and latitude (i.e. with this system, only Grids of dimension 2 are supported in the regridding).

**ESMC_COORDSYS_SPH_RAD** Spherical coordinates in radians. In this system, the spherical coordinates are mapped to the Grid coordinate dimensions in the following order: longitude, latitude, radius. (E.g. using `coordDim=2` in ESMC_GridGetCoord() references the latitude dimension) Note, however, that ESMC_FieldRegridStore() currently just supports longitude and latitude (i.e. with this system, only Grids of dimension 2 are supported in the regridding).

### 19.2.2 ESMC_GRIDITEM

DESCRIPTION:
The ESMC Grid can contain other kinds of data besides coordinates. This data is referred to as Grid "items". Some items may be used by ESMC for calculations involving the Grid. The following are the valid values of ESMC_GridItem_Flag.

The type of this flag is:

`type(ESMC_GridItem_Flag)`

The valid values are:

| Item Label | Type Restriction | Type Default | ESMC Uses | Controls |
|---|---|---|---|---|
| **ESMC_GRIDITEM_MASK** | ESMC_TYPEKIND_I4 | ESMC_TYPEKIND_I4 | YES | Masking in Regrid |
| **ESMC_GRIDITEM_AREA** | NONE | ESMC_TYPEKIND_R8 | YES | Conservation in Regrid |

### 19.2.3 ESMC_GRIDSTATUS

DESCRIPTION:
The ESMC Grid class can exist in two 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 ESMC_GRIDSTATUS.

The type of this flag is:

`type(ESMC_GridStatus_Flag)`

The valid values are:

**ESMC_GRIDSTATUS_EMPTY:** Status after a Grid has been created with `ESMC_GridEmptyCreate`. 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 `ESMC_GridEmptyComplete()` methods in which additional information is added to the Grid.

**ESMC_GRIDSTATUS_COMPLETE:** The Grid has a specific topology and distribution, but incomplete coordinate arrays. The Grid can be used as the basis for allocating a Field, and coordinates can be added via `ESMC_GridCoordAdd()` to allow other functionality.

### 19.2.4 ESMC_POLEKIND

DESCRIPTION:
This type describes the type of connection that occurs at the pole when a Grid is created with `ESMC_GridCreate1PeriodicDim()`.

The type of this flag is:

`type(ESMC_PoleKind_Flag)`

The valid values are:

**ESMC_POLEKIND_NONE** No connection at pole.

**ESMC_POLEKIND_MONOPOLE** This edge is connected to itself. Given that the edge is n elements long, then element i is connected to element i+n/2.

**ESMC_POLEKIND_BIPOLE** This edge is connected to itself. Given that the edge is n elements long, element i is connected to element n-i+1.

### 19.2.5 ESMC_STAGGERLOC

DESCRIPTION:
In the ESMC 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, ESMC 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 9) are:

**ESMC_STAGGERLOC_CENTER:** The center of the cell.

**ESMC_STAGGERLOC_CORNER:** The corners of the cell.

**ESMC_STAGGERLOC_EDGE1:** The edges offset from the center in the 1st dimension.

**ESMC_STAGGERLOC_EDGE2:** The edges offset from the center in the 2nd dimension.

The 3D predefined stagger locations (illustrated in figure 10) are:

**ESMC_STAGGERLOC_CENTER_VCENTER:** The center of the 3D cell.

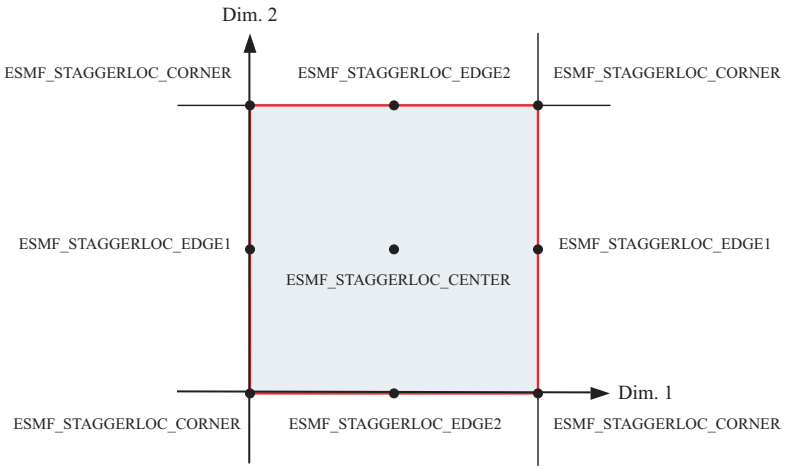



Figure 9: 2D Predefined Stagger Locations

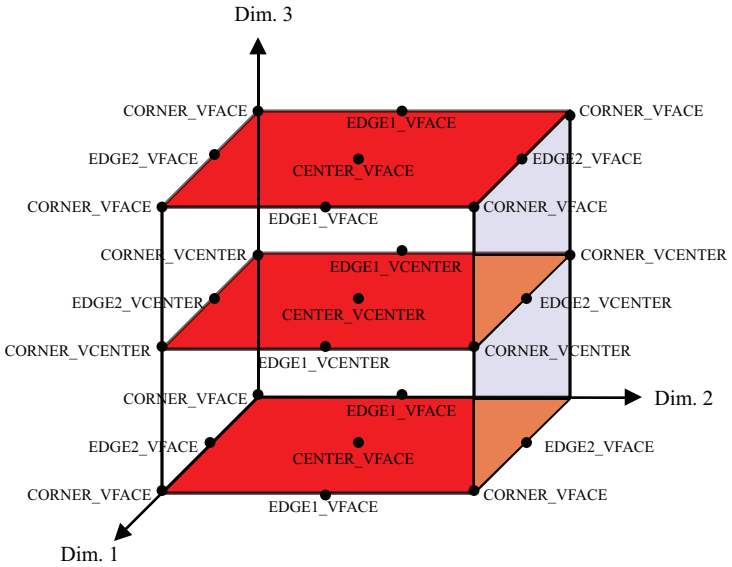



Figure 10: 3D Predefined Stagger Locations

**ESMC_STAGGERLOC_CORNER_VCENTER:** Half way up the vertical edges of the cell.

**ESMC_STAGGERLOC_EDGE1_VCENTER:** The center of the face bounded by edge 1 and the vertical dimension.

**ESMC_STAGGERLOC_EDGE2_VCENTER:** The center of the face bounded by edge 2 and the vertical dimension.

**ESMC_STAGGERLOC_CORNER_VFACE:** The corners of the 3D cell.

**ESMC_STAGGERLOC_EDGE1_VFACE:** The center of the edges of the 3D cell parallel offset from the center in the 1st dimension.

**ESMC_STAGGERLOC_EDGE2_VFACE:** The center of the edges of the 3D cell parallel offset from the center in the 2nd dimension.

**ESMC_STAGGERLOC_CENTER_VFACE:** The center of the top and bottom face. The face bounded by the 1st and 2nd dimensions.

#### 19.2.6 ESMC_FILEFORMAT

DESCRIPTION:
This option is used by `ESMC_GridCreateFromFile` to specify the type of the input grid file.

The type of this flag is:

```
type(ESMC_FileFormat_Flag)
```

The valid values are:

**ESMC_FILEFORMAT_SCRIP** SCRIP format grid file. The SCRIP format is the format accepted by the SCRIP regridding tool [1]. For Grid creation, files of this type only work when the `grid_rank` in the file is equal to 2.

**ESMC_FILEFORMAT_GRIDSPEC** a single tile grid file comforming with the proposed CF-GRIDSPEC conventions.

### 19.3 Restrictions and Future Work

- **Grids with factorized coordinates can only be redisted when they are 2D.** Using the ESMF_GridCreate() interface that allows the user to create a copy of an existing Grid with a new distribution will give incorrect results when used on a Grid with 3 or more dimensions and whose coordinate arrays are less than the full dimension of the Grid (i.e. it contains factorized coordinates).
- **7D limit.** Only grids up to 7D will be supported.
- **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 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.

## 19.4 Design and Implementation Notes

### 19.4.1 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 machinary 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.

## 19.5 Class API: General Grid Methods

### 19.5.1 ESMC_GridCreateNoPeriDim - Create a Grid with no periodic dimensions

INTERFACE:

```
ESMC_Grid ESMC_GridCreateNoPeriDim(
  ESMC_InterArrayInt *maxIndex,           // in
  enum ESMC_CoordSys_Flag *coordSys,      // in
  enum ESMC_TypeKind_Flag *coordTypeKind, // in
  enum ESMC_IndexFlag *indexflag,         // in
  int *rc                                 // out
);
```

*RETURN VALUE:*

```
type(ESMC_Grid)
```

DESCRIPTION:

This call creates an ESMC_Grid with no periodic dimensions.

The arguments are:

**maxIndex** The upper extent of the grid array.

**coordSys** The coordinated system of the grid coordinate data. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

**coordTypeKind** The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.

**indexflag** Indicates the indexing scheme to be used in the new Grid. If not present, defaults to ESMC_INDEX_DELOCAL.

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

### 19.5.2 ESMC_GridCreate1PeriDim - Create a Grid with 1 periodic dimension

INTERFACE:

```
ESMC_Grid ESMC_GridCreate1PeriDim(
  ESMC_InterArrayInt *maxIndex,           // in
  ESMC_InterArrayInt *polekindflag,       // in
  int *periodicDim,                       // in
  int *poleDim,                           // in
  enum ESMC_CoordSys_Flag *coordSys,      // in
  enum ESMC_TypeKind_Flag *coordTypeKind, // in
  enum ESMC_IndexFlag *indexflag,         // in
  int *rc                                 // out
);
```

*RETURN VALUE:*

```
    type(ESMC_Grid)
```

DESCRIPTION:

This call creates an ESMC_Grid with 1 periodic dimension.

The arguments are:

**maxIndex** The upper extent of the grid array.

**polekindflag** Two item array which specifies the type of connection which occurs at the pole. polekindflag(1) the connection that occurs at the minimum end of the index dimension. polekindflag(2) the connection that occurs at the maximum end of the index dimension. If not specified, the default is ESMF_POLETYPE_MONOPOLE for both.

**periodicDim** The periodic dimension. If not specified, defaults to 1.

**poleDim** The dimension at which the poles are located at the ends. If not specified, defaults to 2.

**coordSys** The coordinated system of the grid coordinate data. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

**coordTypeKind** The type/kind of the grid coordinate data. If not specified then the type/kind will be 8 byte reals.

**indexflag** Indicates the indexing scheme to be used in the new Grid. If not present, defaults to ESMC_INDEX_DELOCAL.

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

---

### 19.5.3 ESMC_GridCreateCubedSphere - Create a cubed sphere Grid

INTERFACE:

```
ESMC_Grid ESMC_GridCreateCubedSphere(
  int *tilesize,                       // in
  ESMC_InterArrayInt *regDecompPTile, // in
  //ESMC_InterArrayInt *decompFlagPTile,  // in
  //ESMC_InterArrayInt *deLabelList,    // in
  //ESMC_DELayout *delayout,            // in
  ESMC_InterArrayInt *staggerLocList,   // in
  const char *name,                    // in
  int *rc);                            // out
```

*RETURN VALUE:*

```
type(ESMC_Grid)
```

DESCRIPTION:

Create a six-tile `ESMC_Grid` for a cubed sphere grid using regular decomposition. Each tile can have different decomposition. The grid coordinates are generated based on the algorithm used by GEOS-5. The tile resolution is defined by tileSize.

The arguments are:

**tilesize** The number of elements on each side of the tile of the cubed sphere grid.

**regDecompPTile** List of DE counts for each dimension. The second index steps through the tiles. The total `deCount` is determined as the sum over the products of `regDecompPTile` elements for each tile. By default every tile is decomposed in the same way. If the total PET count is less than 6, one tile will be assigned to one DE and the DEs will be assigned to PETs sequentially, therefore, some PETs may have more than one DE. If the total PET count is greater than 6, the total number of DEs will be a multiple of 6 and less than or equal to the total PET count. For instance, if the total PET count is 16, the total DE count will be 12 with each tile decomposed into 1x2 blocks. The 12 DEs are mapped to the first 12 PETs and the remaining 4 PETs have no DEs locally, unless an optional `delayout` is provided.

**staggerLocList** The list of stagger locations to fill with coordinates. Only `ESMF_STAGGERLOC_CENTER` and `ESMF_STAGGERLOC_CORNER` are supported. If not present, no coordinates will be added or filled.

**name** The name of the `ESMC_Grid`.

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

---

### 19.5.4 ESMC_GridCreateFromFile - Create a Grid from a NetCDF file specification.

INTERFACE:

```
ESMC_Grid ESMC_GridCreateFromFile(const char *filename, int fileTypeFlag,
                  int *regDecomp, int *decompflag,
                  int *isSphere, ESMC_InterArrayInt *polekindflag,
                  int *addCornerStagger,
                  int *addUserArea, enum ESMC_IndexFlag *indexflag,
                  int *addMask, const char *varname,
                  const char **coordNames, int *rc);
```

*RETURN VALUE:*

`type(ESMC_Grid)`

DESCRIPTION:

This function creates a `ESMC_Grid` object from the specification in a NetCDF file.

The arguments are:

**filename** The NetCDF Grid filename.

**fileTypeFlag** The Grid file format, please see Section 19.2.6 for a list of valid options.

**regDecomp** A 2 element array specifying how the grid is decomposed. Each entry is the number of decounts for that dimension. The total decounts cannot exceed the total number of PETs. In other word, at most one DE is allowed per processor. If not specified, the default decomposition will be petCountx1.

**[decompflag]** List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_BALANCED` in all dimensions. Please see Section 34.3 for a full description of the possible options.

**[isSphere]** Set to 1 for a spherical grid, or 0 for regional. Defaults to 1.

**polekindflag** Two item array which specifies the type of connection which occurs at the pole. polekindflag(1) the connection that occurs at the minimum end of the index dimension. polekindflag(2) the connection that occurs at the maximum end of the index dimension. If not specified, the default is ESMF_POLETYPE_MONOPOLE for both.

**[addCornerStagger]** Set to 1 to use the information in the grid file to add the Corner stagger to the Grid. The coordinates for the corner stagger are required for conservative regridding. If not specified, defaults to 0.

**[addUserArea]** Set to 1 to read in the cell area from the Grid file; otherwise, ESMF will calculate it. This feature is only supported when the grid file is in the SCRIP format.

**indexflag** Indicates the indexing scheme to be used in the new Grid. If not present, defaults to ESMC_INDEX_DELOCAL.

**[addMask]** Set to 1 to generate the mask using the missing_value attribute defined in 'varname'. This flag is only needed when the grid file is in the GRIDSPEC format.

**[varname]** If addMask is non-zero, provide a variable name stored in the grid file and the mask will be generated using the missing value of the data value of this variable. The first two dimensions of the variable has to be the longitude and the latitude dimension and the mask is derived from the first 2D values of this variable even if this data is 3D, or 4D array.

**[coordNames]** A two-element array containing the longitude and latitude variable names in a GRIDSPEC file if there are multiple coordinates defined in the file.

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

---

### 19.5.5 ESMC_GridDestroy - Destroy a Grid

INTERFACE:

```
int ESMC_GridDestroy(
  ESMC_Grid *grid             // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Destroy the Grid.

The arguments are:

**grid** Grid object whose memory is to be freed.

---

### 19.5.6 ESMC_GridAddItem - Add items to a Grid

INTERFACE:

```
int ESMC_GridAddItem(
  ESMC_Grid grid,                    // in
  enum ESMC_GridItem_Flag itemflag, // in
  enum ESMC_StaggerLoc staggerloc   // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Add an item (e.g. a mask) to the Grid.

The arguments are:

**grid** Grid object to which the coordinates will be added

**itemflag** The grid item to add.

**staggerloc** The stagger location to add.

---

### 19.5.7 ESMC_GridGetItem - Get item from a Grid

INTERFACE:

```
void * ESMC_GridGetItem(
  ESMC_Grid grid,                          // in
  enum ESMC_GridItem_Flag itemflag,        // in
  enum ESMC_StaggerLoc staggerloc,         // in
  int *localDE,                            // in
  int *rc                                  // out
);
```

*RETURN VALUE:*

```
    A pointer to the item data.
```

DESCRIPTION:

Get a pointer to item data (e.g. mask data) in the Grid.

The arguments are:

**grid** Grid object from which to obtain the coordinates.

**itemflag** The grid item to add.

**staggerloc** The stagger location to add.

**localDE** The local decompositional element. If not present, defaults to 0.

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

---

### 19.5.8 ESMC_GridAddCoord - Add coordinates to a Grid

INTERFACE:

```
int ESMC_GridAddCoord(
  ESMC_Grid grid,                    // in
  enum ESMC_StaggerLoc staggerloc    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Add coordinates to the Grid.

The arguments are:

**grid** Grid object to which the coordinates will be added

**staggerloc** The stagger location to add.

---

### 19.5.9 ESMC_GridGetCoord - Get coordinates from a Grid

INTERFACE:

```
 void * ESMC_GridGetCoord(
   ESMC_Grid grid,                          // in
   int coordDim,                            // in
   enum ESMC_StaggerLoc staggerloc,         // in
   int *localDE,
   int *exclusiveLBound,                    // out
   int *exclusiveUBound,                    // out
   int *rc                                  // out
 );
```

*RETURN VALUE:*

```
    A pointer to coordinate data in the Grid.
```

DESCRIPTION:

Get a pointer to coordinate data in the Grid.

The arguments are:

**grid** Grid object from which to obtain the coordinates.

**coordDim** The coordinate dimension from which to get the data.

**staggerloc** The stagger location to add.

**localDE** The local decompositional element. If not present, defaults to 0.

**exclusiveLBound** Upon return this holds the lower bounds of the exclusive region. This bound must be allocated to be of size equal to the coord dimCount.

**exclusiveUBound** Upon return this holds the upper bounds of the exclusive region. This bound must be allocated to be of size equal to the coord dimCount.

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

---

### 19.5.10 ESMC_GridGetCoordBounds - Get coordinate bounds from a Grid

INTERFACE:

```
int ESMC_GridGetCoordBounds(
  ESMC_Grid grid,                          // in
  enum ESMC_StaggerLoc staggerloc,         // in
  int *localDE,                            // in
  int *exclusiveLBound,                    // out
  int *exclusiveUBound,                    // out
  int *rc                                  // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Get coordinates bounds from the Grid.

The arguments are:

**grid** Grid object from which to obtain the coordinates.

**staggerloc** The stagger location to add.

**localDE** The local decompositional element. If not present, defaults to 0.

**exclusiveLBound** Upon return this holds the lower bounds of the exclusive region. This bound must be allocated to be of size equal to the coord dimCount.

**exclusiveUBound** Upon return this holds the upper bounds of the exclusive region. This bound must be allocated to be of size equal to the coord dimCount.

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

---

### 19.5.11 ESMC_GridGetLocalDECount - Get the number of DEs in this grid on this PET

INTERFACE:

```
int ESMC_GridGetLocalDECount(
  ESMC_Grid grid,                    // in
  int *localDECount                  // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Get the number of DEs in this grid on this PET.

The arguments are:

**grid** Grid object from which to obtain the local DE count.

**localDECount** The number of DEs in this grid on this PET.

# 20 Mesh Class

## 20.1 Description

Unstructured grids are commonly used in the computational solution of Partial Differential equations. These are especially useful for problems that involve complex geometry, where using the less flexible structured grids can result in grid representation of regions where no computation is needed. Finite element and finite volume methods map naturally to unstructured grids and are used commonly in hydrology, ocean modeling, and many other applications.

In order to provide support for application codes using unstructured grids, the ESMF library provides a class for representing unstructured grids called the **Mesh**. Fields can be created on a Mesh to hold data. In Fortran, Fields created on a Mesh can also be used as either the source or destination or both of an interpolation (i.e. an `ESMF_FieldRegridStore()` call). This capability is currently not supported with the C interface, however, if the C Field is passed via a State to a component written in Fortran then the regridding can be performed there. The rest of this section describes the Mesh class and how to create and use them in ESMF.

### 20.1.1 Mesh Representation in ESMF

A Mesh in ESMF is described in terms of **nodes** and **elements**. A node is a point in space which represents where the coordinate information in a Mesh is located. An element is a higher dimensional shape constructed of nodes. Elements give a Mesh its shape and define the relationship of the nodes to one another. Field data may be located on a Mesh's nodes.

### 20.1.2 Supported Meshes

The range of Meshes supported by ESMF are defined by several factors: dimension, element types, and distribution.

ESMF currently only supports Meshes whose number of coordinate dimensions (spatial dimension) is 2 or 3. The dimension of the elements in a Mesh (parametric dimension) must be less than or equal to the spatial dimension, but also must be either 2 or 3. This means that an ESMF mesh may be either 2D elements in 2D space, 3D elements in 3D space, or a manifold constructed of 2D elements embedded in 3D space.

ESMF currently supports two types of elements for each Mesh parametric dimension. For a parametric dimension of 2 the supported element types are triangles or quadrilaterals. For a parametric dimension of 3 the supported element types are tetrahedrons and hexahedrons. See Section 20.2.1 for diagrams of these. The Mesh supports any combination of element types within a particular dimension, but types from different dimensions may not be mixed, for example, a Mesh cannot be constructed of both quadrilaterals and tetrahedra.

ESMF currently only supports distributions where every node on a PET must be a part of an element on that PET. In other words, there must not be nodes without an element on a PET.

## 20.2 Constants

### 20.2.1 ESMC_MESHELEMTYPE

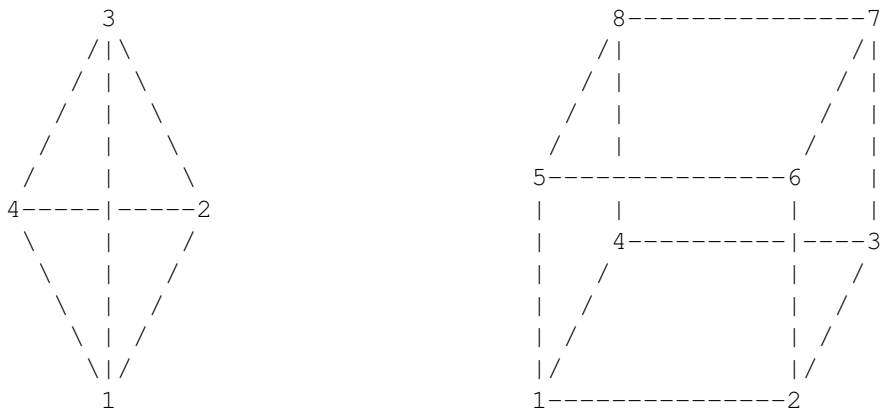
DESCRIPTION:
An ESMF Mesh can be constructed from a combination of different elements. The type of elements that can be used in a Mesh depends on the Mesh's parametric dimension, which is set during Mesh creation. The following are the valid Mesh element types for each valid Mesh parametric dimension (2D or 3D) .




```
2D element types (numbers are the order for elementConn during
                  Mesh create)
```

For a Mesh with parametric dimension of 2 the valid element types (illustrated above) are:

| Element Type | Number of Nodes | Description |
|---|---|---|
| ESMC_MESHELEMTYPE_TRI | 3 | A triangle |
| ESMC_MESHELEMTYPE_QUAD | 4 | A quadrilateral (e.g. a rectangle) |



```
       ESMC_MESHELEMTYPE_TETRA             ESMC_MESHELEMTYPE_HEX

3D element types (numbers are the order for elementConn during
                  Mesh create)
```

For a Mesh with parametric dimension of 3 the valid element types (illustrated above) are:

| Element Type | Number of Nodes | Description |
|---|---|---|
| ESMC_MESHELEMTYPE_TETRA | 4 | A tetrahedron (CAN'T BE USED IN REGRID) |
| ESMC_MESHELEMTYPE_HEX | 8 | A hexahedron (e.g. a cube) |

#### 20.2.2 ESMF_FILEFORMAT

DESCRIPTION:
This option is used by `ESMF_MeshCreate` to specify the type of the input grid file.

The type of this flag is:

```
type(ESMF_FileFormat_Flag)
```

The valid values are:

**ESMF_FILEFORMAT_SCRIP** SCRIP format grid file. The SCRIP format is the format accepted by the SCRIP regridding tool [1]. For Mesh creation, files of this type only work when the `grid_rank` in the file is equal to 1.

**ESMF_FILEFORMAT_ESMFMESH** ESMF unstructured grid file format. This format was developed by the ESMF team to match the capabilities of the Mesh class and to be efficient to convert to that class.

**ESMF_FILEFORMAT_UGRID** CF-convention unstructured grid file format. This format is a proposed extension to the CF-conventions for unstructured grid data model. Currently, only the 2D flexible mesh topology is supported in ESMF.

### 20.3 Class API

#### 20.3.1 ESMC_MeshGetMOAB - Query if the MOAB mesh backend is enabled

INTERFACE:

```
 void ESMC_MeshGetMOAB(
   bool *moabOn,
   int *rc
 );
```

*RETURN VALUE:*

```
    None
```

DESCRIPTION:

This call will query whether or not the MOAB mesh backend is enabled.

The arguments are:

**moabOn** This parameter will receive a boolean value indicating if MOAB is enabled.

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

---

### 20.3.2 ESMC_MeshSetMOAB - Toggle the MOAB mesh backend

INTERFACE:

```
void ESMC_MeshSetMOAB(
  bool moabOn,
  int *rc
);
```

*RETURN VALUE:*

```
    None
```

DESCRIPTION:

This call will toggle the MOAB mesh backend.

The arguments are:

**moabOn** This parameter will hold a boolean value to indicate the setting for the MOAB mesh back end (on or off).

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

---

### 20.3.3 ESMC_MeshAddElements - Add elements to a Mesh

INTERFACE:

```
int ESMC_MeshAddElements(
  ESMC_Mesh mesh,           // inout
  int  elementCount,        // in
  int *elementIds,          // in
  int *elementTypes,        // in
  int *elementConn,         // in
  int *elementMask,         // in
  double *elementArea,      // in
```

```
  double *elementCoords     // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

This call is the third and last part of the three part mesh create sequence and should be called after the mesh is created with `ESMF_MeshCreate()` (20.3.5) and after the nodes are added with `ESMF_MeshAddNodes()` (20.3.4). This call adds the elements to the mesh and finalizes the create. After this call the Mesh is usable, for example a Field may be built on the created Mesh object and this Field may be used in a `ESMF_FieldRegridStore()` call.

The parameters to this call `elementIds`, `elementTypes`, and `elementConn` describe the elements to be created. The description for a particular element lies at the same index location in `elementIds` and `elementTypes`. Each entry in `elementConn` consists of the list of nodes used to create that element, so the connections for element $e$ in the `elementIds` array will start at $number_of_nodes_in_element(1) + number_of_nodes_in_element(2) + \cdots + number_of_nodes_in_element(e-1) + 1$ in `elementConn`.

**mesh** Mesh object.

**elementCount** The number of elements on this PET.

**elementIds** An array containing the global ids of the elements to be created on this PET. This input consists of a 1D array of size `elementCount`.

**elementTypes** An array containing the types of the elements to be created on this PET. The types used must be appropriate for the parametric dimension of the Mesh. Please see Section 20.2.1 for the list of options. This input consists of a 1D array of size `elementCount`.

**elementConn** An array containing the indexes of the sets of nodes to be connected together to form the elements to be created on this PET. The entries in this list are NOT node global ids, but rather each entry is a local index (1 based) into the list of nodes which were created on this PET by the previous `ESMC_MeshAddNodes()` call. In other words, an entry of 1 indicates that this element contains the node described by `nodeIds(1)`, `nodeCoords(1)`, etc. passed into the `ESMC_MeshAddNodes()` call on this PET. It is also important to note that the order of the nodes in an element connectivity list matters. Please see Section 20.2.1 for diagrams illustrating the correct order of nodes in a element. This input consists of a 1D array with a total size equal to the sum of the number of nodes in each element on this PET. The number of nodes in each element is implied by its element type in `elementTypes`. The nodes for each element are in sequence in this array (e.g. the nodes for element 1 are elementConn(1), elementConn(2), etc.).

**[elementMask]** An array containing values which can be used for element masking. Which values indicate masking are chosen via the srcMaskValues or dstMaskValues arguments to ESMF_FieldRegridStore() call. This input consists of a 1D array the size of the number of elements on this PET. If not specified (i.e. NULL is passed in), then no masking will occur.

**[elementArea]** An array containing element areas. This input consists of a 1D array the size of the number of elements on this PET. If not specified (i.e. NULL is passed in), the element areas are internally calculated.

**[elementCoords]** An array containing the physical coordinates of the elements to be created on this PET. This input consists of a 1D array the size of the number of elements on this PET times the Mesh's spatial dimension (`spatialDim`). The coordinates in this array are ordered so that the coordinates for an element lie in sequence in memory. (e.g. for a Mesh with spatial dimension 2, the coordinates for element 1 are in elementCoords(1) and elementCoords(2), the coordinates for element 2 are in elementCoords(3) and elementCoords(4), etc.).

### 20.3.4 ESMC_MeshAddNodes - Add nodes to a Mesh

INTERFACE:

```
int ESMC_MeshAddNodes(
  ESMC_Mesh mesh,           // inout
  int nodeCount,            // in
  int *nodeIds,             // in
  double *nodeCoords,       // in
  int *nodeOwners,          // in
  int *nodeMask             // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

This call is the second part of the three part mesh create sequence and should be called after the mesh's dimensions are set using `ESMC_MeshCreate()`. This call adds the nodes to the mesh. The next step is to call `ESMC_MeshAddElements()` (20.3.5).

The parameters to this call `nodeIds`, `nodeCoords`, and `nodeOwners` describe the nodes to be created on this PET. The description for a particular node lies at the same index location in `nodeIds` and `nodeOwners`. Each entry in `nodeCoords` consists of spatial dimension coordinates, so the coordinates for node $n$ in the `nodeIds` array will start at $(n-1) * spatialDim + 1$.

**mesh** Mesh object.

**nodeCount** The number of nodes on this PET.

**nodeIds** An array containing the global ids of the nodes to be created on this PET. This input consists of a 1D array the size of the number of nodes on this PET (i.e. `nodeCount`).

**nodeCoords** An array containing the physical coordinates of the nodes to be created on this PET. The coordinates in this array are ordered so that the coordinates for a node lie in sequence in memory. (e.g. for a Mesh with spatial dimension 2, the coordinates for node 1 are in nodeCoords(0) and nodeCoords(1), the coordinates for node 2 are in nodeCoords(2) and nodeCoords(3), etc.). This input consists of a 1D array the size of `nodeCount` times the Mesh's spatial dimension (`spatialDim`).

**nodeOwners** An array containing the PETs that own the nodes to be created on this PET. If the node is shared with another PET, the value may be a PET other than the current one. Only nodes owned by this PET will have PET local entries in a Field created on the Mesh. This input consists of a 1D array the size of the number of nodes on this PET (i.e. `nodeCount`).

---

### 20.3.5 ESMC_MeshCreate - Create a Mesh as a 3 step process

INTERFACE:

```
ESMC_Mesh ESMC_MeshCreate(
  int parametricDim,         // in
  int spatialDim,            // in
  enum ESMC_CoordSys_Flag *coordSys, // in
  int *rc                    // out
);
```

*RETURN VALUE:*

```
    type(ESMC_Mesh)         :: ESMC_MeshCreate
```

DESCRIPTION:

This call is the first part of the three part mesh create sequence. This call sets the dimension of the elements in the mesh (`parametricDim`) and the number of coordinate dimensions in the mesh (`spatialDim`). The next step is to call `ESMC_MeshAddNodes()` (20.3.4) to add the nodes and then `ESMC_MeshAddElements()` (20.3.3) to add the elements and finalize the mesh.

The arguments are:

**parametricDim** Dimension of the topology of the Mesh. (E.g. a mesh constructed of squares would have a parametric dimension of 2, whereas a Mesh constructed of cubes would have one of 3.)

**spatialDim** The number of coordinate dimensions needed to describe the locations of the nodes making up the Mesh. For a manifold, the spatial dimension can be larger than the parametric dim (e.g. the 2D surface of a sphere in 3D space), but it can't be smaller.

**[coordSys]** Set the coordinate system of the mesh. If not specified, then defaults to ESMC_COORDSYS_SPH_DEG.

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

---

### 20.3.6 ESMC_MeshCreateFromFile - Create a Mesh from a NetCDF grid file

INTERFACE:

```
ESMC_Mesh ESMC_MeshCreateFromFile(
                                  const char *filename, // in (required)
                                  int fileTypeFlag,     // in (required)
                                  int *convertToDual,   // in (optional)
                                  int *addUserArea,     // in (optional)
                                  const char *meshname, // in (optional)
                                  int *maskFlag,        // in (optional)
                                  const char *varname,  // in (optional)
                                  int *rc               // out
);
```

*RETURN VALUE:*

```
    type(ESMC_Mesh)         :: ESMC_MeshCreateFromFile
```

DESCRIPTION:

Method to create a Mesh object from a NetCDF file in either SCRIP, UGRID, or ESMF file formats.

The required arguments are:

**filename** The name of the grid file

**filetypeflag** The file type of the grid file to be read, please see Section 20.2.2 for a list of valid options.

**[convertToDual]** if 1, the mesh will be converted to its dual. If not specified, defaults to 0. Converting to dual is only supported with file type `ESMF_FILEFORMAT_SCRIP`.

**[addUserArea]** if 1, the cell area will be read in from the GRID file. This feature is only supported when the grid file is in the SCRIP or ESMF format. If not specified, defaults to 0.

**[meshname]** The dummy variable for the mesh metadata in the UGRID file if the `filetypeflag` is `ESMF_FILEFORMAT_UGRID`. If not specified, defaults to empty string.

**[maskFlag]** An enumerated integer that, if specified, tells whether a mask in a UGRID file should be defined on the nodes (MeshLoc.NODE) or elements (MeshLoc.ELEMENT) of the mesh. If specified, generate the mask using the missing_value attribute defined in 'varname'. This flag is only supported when the grid file is in the UGRID format. If not specified, defaults to no mask.

**[varname]** If maskFlag is specified, provide a variable name stored in the UGRID file and the mask will be generated using the missing value of the data value of this variable. The first two dimensions of the variable has to be the the longitude and the latitude dimension and the mask is derived from the first 2D values of this variable even if this data is 3D, or 4D array. If not specified, defaults to empty string.

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

---

### 20.3.7 ESMC_MeshGetCoord - Get lat/lon coordinates from a Mesh

INTERFACE:

```
void ESMC_MeshGetCoord(
                         ESMC_Mesh mesh_in, // in (required)
                         double *nodeCoord, // out
                         int *num_nodes,    // out
                         int *num_dims,     // out
                         int *rc            // out
                         );
```

*RETURN VALUE:*

```
None
```

DESCRIPTION:

This call returns the node coordinates of the given `ESMC_Mesh` in the provided `nodeCoord` buffer of doubles. At completion, this buffer is a 1-D array with the coordinates for a given node in adjacent indices. For example, for

$d$-dimensional coordinates, the first $d$ values in the returned array are the coordinates for the first node, the second $d$ values are the coordinates for the second node, etc.

The arguments are:

**mesh_in** Mesh object.

**nodeCoord** Pointer to doubles. The node coordinates are returned here.

**num_nodes** Pointer to an integer. The number of nodes found in the input Mesh is returned here.

**num_dims** Pointer to an integer. The number of coordinate dimensions is returned here.

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

---

### 20.3.8 ESMC_MeshGetElemCoord - Get lat/lon element center coordinates from a Mesh

INTERFACE:

```
void ESMC_MeshGetElemCoord(
                          ESMC_Mesh mesh_in, // in (required)
                          double *elemCoord, // out
                          int *num_elems,    // out
                          int *num_dims,     // out
                          int *rc            // out
                          );
```

*RETURN VALUE:*

```
    None
```

DESCRIPTION:

This call returns the element coordinates of the given `ESMC_Mesh` in the provided `elemCoord` buffer of doubles. At completion, this buffer is a 1-D array with the coordinates for a given element in adjacent indices. For example, for $d$-dimensional coordinates, the first $d$ values in the returned array are the coordinates for the first element, the second $d$ values are the coordinates for the second element, etc.

The arguments are:

**mesh_in** Mesh object.

**elemCoord** Pointer to doubles. The element coordinates are returned here.

**num_elems** Pointer to an integer. The number of elements found in the input Mesh is returned here.

**num_dims** Pointer to an integer. The number of coordinate dimensions is returned here.

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

---

### 20.3.9 ESMC_MeshGetConnectivity - Get Mesh connectivity

INTERFACE:

```
void ESMC_MeshGetConnectivity(
                          ESMC_Mesh mesh_in,      // in (required)
                          double *connCoord,      // out
                          int *nodesPerElem,  // out
                          int *rc                 // out
                          );
```

*RETURN VALUE:*

```
    None
```

DESCRIPTION:

NOTE: At this time the connectivity that is returned from this call is not necessarily in the same format as how it was passed into the creation routine.

This call returns the connectivity of the given `ESMC_Mesh` in the provided `connCoord` buffer of doubles. At completion, this buffer is a 1-D array with the coordinates for the nodes of a given element in counterclockwise order. The /tt nodesPerElem buffer of integers contains the number of nodes to expect per element.

The arguments are:

**mesh_in** Mesh object.

**connCoord** Pointer to doubles. The connectivity is returned here.

**nodesPerElem** Pointer to integers. The number of nodes in each element.

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

---

### 20.3.10 ESMC_MeshDestroy - Destroy a Mesh

INTERFACE:

```
int ESMC_MeshDestroy(
  ESMC_Mesh *mesh             // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Destroy the Mesh. This call removes all internal memory associated with `mesh`. After this call mesh will no longer be usable.

The arguments are:

**mesh** Mesh object whose memory is to be freed.

---

### 20.3.11 ESMC_MeshFreeMemory - Remove a Mesh and its memory

INTERFACE:

```
int ESMC_MeshFreeMemory(
  ESMC_Mesh mesh            // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

This call removes the portions of `mesh` which contain connection and coordinate information. After this call, Fields build on `mesh` will no longer be usable as part of an `ESMF_FieldRegridStore()` operation. However, after this call Fields built on `mesh` can still be used in an `ESMF_FieldRegrid()` operation if the routehandle was generated beforehand. New Fields may also be built on `mesh` after this call.

The arguments are:

**mesh** Mesh object whose memory is to be freed.

---

### 20.3.12 ESMC_MeshGetElementCount - Get the number of elements in a Mesh on the current PET

INTERFACE:

```
int ESMC_MeshGetElementCount(
  ESMC_Mesh mesh,           // in
  int *elementCount         // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Query the number of elements in a mesh on the local PET. The arguments are:

**mesh** The mesh

**elementCount** The number of elements on this PET.

---

### 20.3.13 ESMC_MeshGetNodeCount - Get the number of nodes in a Mesh on the current PET

INTERFACE:

```
int ESMC_MeshGetNodeCount(
  ESMC_Mesh mesh,          // in
  int *nodeCount           // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Query the number of nodes in a mesh on the local PET. The arguments are:

**mesh** The mesh

**nodeCount** The number of nodes on this PET.

---

### 20.3.14 ESMC_MeshGetOwnedElementCount - Get the number of elements in a Mesh owned by the current PET

INTERFACE:

```
int ESMC_MeshGetOwnedElementCount(
  ESMC_Mesh mesh,           // in
  int *elementCount         // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Query the number of elements in a mesh owned by the local PET. This number will be equal or less than the local element count. The arguments are:

**mesh** The mesh

**elementCount** The number of elements owned by this PET.

---

#### 20.3.15 ESMC_MeshGetOwnedNodeCount - Get the number of nodes in a Mesh owned by the current PET

INTERFACE:

```
int ESMC_MeshGetOwnedNodeCount(
  ESMC_Mesh mesh,          // in
  int *nodeCount           // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Query the number of nodes in a mesh owned by the local PET. This number will be equal or less than the local node count. The arguments are:

**mesh** The mesh

**nodeCount** The number of nodes owned by this PET.

# 21 XGrid Class

## 21.1 Description

An exchange grid represents the 2D boundary layer usually between the atmosphere on one side and ocean and land on the other in an Earth system model. There are dynamical and thermodynamical processes on either side of the boundary layer and on the boundary layer itself. The boundary layer exchanges fluxes from either side and adjusts boundary conditions for the model components involved. For climate modeling, it is critical that the fluxes transferred by the boundary layer are conservative.

The ESMF exchange grid is implemented as the `ESMC_XGrid` class. Internally it's represented by a collection of the intersected cells between atmosphere and ocean/land[2] grids. These polygonal cells can have irregular shapes and can be broken down into triangles facilitating a finite element approach.

Through the C API there is one way to create an ESMC_XGrid object from user supplied information. The ESMC_XGrid takes two lists of ESMC_Grid or ESMC_Mesh that represent the model component grids on either side of the exchange grid. From the two lists of ESMC_Grid or ESMC_Mesh, information required for flux exchange calculation between any pair of the model components from either side of the exchange grid is computed. In addition, the internal representation of the ESMC_XGrid is computed and can be optionally stored as an ESMC_Mesh. This internal representation is the collection of the intersected polygonal cells as a result of merged ESMC_Meshes from both sides of the exchange grid. ESMC_Field can be created on the ESMC_XGrid and used for weight generation and regridding as the internal representation in the ESMC_XGrid has a complete geometrical description of the exchange grid.

Once an ESMC_XGrid has been created, information describing it (e.g. cell areas, mesh representation, etc.) can be retrieved from the object using a set of get calls (e.g. ESMC_XGridGetElementArea()). The full extent of these can be found in the API section below.

## 21.2 Restrictions and Future Work

### 21.2.1 Restrictions and Future Work

1. **CAUTION:** Any Grid or Mesh pair picked from the A side and B side of the XGrid cannot point to the same Grid or Mesh in memory on a local PET. This prevents Regrid from selecting the right source and destination grid automatically to calculate the regridding routehandle. It's okay for the Grid and Mesh to have identical topological and geographical properties as long as they are stored in different memory.

## 21.3 Design and Implementation Notes

1. The XGrid class is implemented in Fortran, and as such is defined inside the framework by a XGrid derived type and a set of subprograms (functions and subroutines) which operate on that derived type. The XGrid class contains information needed to create Grid, Field, and communication routehandle.

2. XGrid follows 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 XGrids at the same point in their execution. In addition to the unison rule, XGrid creation also performs inter-PET communication within the current executing VM.

## 21.4 Class API

### 21.4.1 ESMC_XGridCreate - Create an XGrid

INTERFACE:

```
ESMC_XGrid ESMC_XGridCreate(
                            int sideAGridCount,  ESMC_Grid *sideAGrid, // in
                            int sideAMeshCount,  ESMC_Mesh *sideAMesh, // in
                            int sideBGridCount,  ESMC_Grid *sideBGrid, // in
                            int sideBMeshCount,  ESMC_Mesh *sideBMesh, // in
                            ESMC_InterArrayInt *sideAGridPriority,     // in
                            ESMC_InterArrayInt *sideAMeshPriority,     // in
                            ESMC_InterArrayInt *sideBGridPriority,     // in
                            ESMC_InterArrayInt *sideBMeshPriority,     // in
```

```
                                    ESMC_InterArrayInt *sideAMaskValues,       // in
                                    ESMC_InterArrayInt *sideBMaskValues,       // in
                                    int storeOverlay,                          // in
                                    int *rc                                    // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_XGrid object.
```

DESCRIPTION:

Create an `ESMC_XGrid` object from user supplied input: the list of Grids or Meshes on side A and side B, and other optional arguments. A user can supply both Grids and Meshes on one side to create the XGrid. By default, the Grids have a higher priority over Meshes but the order of priority can be adjusted by the optional GridPriority and MeshPriority arguments. The priority order of Grids and Meshes can also be interleaved by rearranging the optional GridPriority and MeshPriority arguments accordingly.

Sparse matrix multiplication coefficients are internally computed and uniquely determined by the Grids or Meshes provided in `sideA` and `sideB`. User can supply a single `ESMC_Grid` or an array of `ESMC_Grid` on either side of the `ESMC_XGrid`. For an array of `ESMC_Grid` or `ESMC_Mesh` in `sideA` or `sideB`, a merging process concatenates all the `ESMC_Grid`s and `ESMC_Mesh`es into a super mesh represented by `ESMC_Mesh`. The super mesh is then used to compute the XGrid. Grid or Mesh objects in `sideA` and `sideB` arguments must have coordinates defined for the corners of a Grid or Mesh cell. XGrid creation can be potentially memory expensive given the size of the input Grid and Mesh objects. By default, the super mesh is not stored to reduce memory usage.

If `sideA` and `sideB` have a single Grid or Mesh object, it's erroneous if the two Grids or Meshes are spatially disjoint. It is also erroneous to specify a Grid or Mesh object in `sideA` or `sideB` that is spatially disjoint from the `ESMC_XGrid`.

This call is *collective* across the current VM. For more details please refer to the description 21.1 of the XGrid class.

The arguments are:

**sideAGridCount** The number of Grids in the `sideAGrid` array.

**[sideAGrid]** Parametric 2D Grids on side A, for example, these Grids can be either Cartesian 2D or Spherical.

**sideAMeshCount** The number of Meshes in the `sideAMesh` array.

**[sideAMesh]** Parametric 2D Meshes on side A, for example, these Meshes can be either Cartesian 2D or Spherical.

**sideBGridCount** The number of Grids in the `sideBGrid` array.

**[sideBGrid]** Parametric 2D Grids on side B, for example, these Grids can be either Cartesian 2D or Spherical.

**sideBMeshCount** The number of Meshes in the `sideBMesh` array.

**[sideBMesh]** Parametric 2D Meshes on side B, for example, these Meshes can be either Cartesian 2D or Spherical.

**[sideAGridPriority]** Priority array of Grids on `sideA` during overlay generation. The priority arrays describe the priorities of Grids at the overlapping region. Flux contributions at the overlapping region are computed in the order from the Grid of the highest priority to the lowest priority.

**[sideAMeshPriority]** Priority array of Meshes on `sideA` during overlay generation. The priority arrays describe the priorities of Meshes at the overlapping region. Flux contributions at the overlapping region are computed in the order from the Mesh of the highest priority to the lowest priority.

**[sideBGridPriority]** Priority of Grids on `sideB` during overlay generation The priority arrays describe the priorities of Grids at the overlapping region. Flux contributions at the overlapping region are computed in the order from the Grid of the highest priority to the lowest priority.

**[sideBMeshPriority]** Priority array of Meshes on `sideB` during overlay generation. The priority arrays describe the priorities of Meshes at the overlapping region. Flux contributions at the overlapping region are computed in the order from the Mesh of the highest priority to the lowest priority.

**[sideAMaskValues]** Mask information can be set in the Grid (see 19.1.8) or Mesh upon which the Field is built. The `sideAMaskValues` argument specifies the values in that mask information which indicate a point should be masked out. In other words, a location is masked if and only if the value for that location in the mask information matches one of the values listed in `sideAMaskValues`. If `sideAMaskValues` is not specified, no masking on side A will occur.

**[sideBMaskValues]** Mask information can be set in the Grid (see 19.1.8) or Mesh upon which the Field is built. The `sideBMaskValues` argument specifies the values in that mask information which indicate a point should be masked out. In other words, a location is masked if and only if the value for that location in the mask information matches one of the values listed in `sideBMaskValues`. If `sideBMaskValues` is not specified, no masking on side B will occur.

**storeOverlay** Setting the `storeOverlay` optional argument to 0. allows a user to bypass storage of the Mesh used to represent the XGrid. Only a DistGrid is stored to allow Field to be built on the XGrid. If the temporary mesh object is of interest, `storeOverlay` can be set to a value not equal to 0. so a user can retrieve it for future use.

**[rc]** Return code; equals `ESMF_SUCCESS` only if the `ESMF_XGrid` is created.

---

### 21.4.2 ESMC_XGridDestroy - Destroy a XGrid

INTERFACE:

```
int ESMC_XGridDestroy(
  ESMC_XGrid *xgrid     // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this `ESMC_XGrid`. Return code; equals `ESMF_SUCCESS` if there are no errors.

The arguments are:

**xgrid** Destroy contents of this `ESMC_XGrid`.

---

### 21.4.3 ESMC_XGridGetSideAGridCount - Get the number of Grids on side A.

INTERFACE:

```
int ESMC_XGridGetSideAGridCount(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The number of Grids on side A.
```

DESCRIPTION:

Get the number of Grids on side A.

The arguments are:

**xgrid** The XGrid from which to get the information.

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

---

### 21.4.4 ESMC_XGridGetSideAMeshCount - Get the number of Meshes on side A.

INTERFACE:

```
int ESMC_XGridGetSideAMeshCount(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The number of Meshes on side A.
```

DESCRIPTION:

Get the number of Meshes on side A.

The arguments are:

**xgrid** The XGrid from which to get the information.

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

---

### 21.4.5 ESMC_XGridGetSideBGridCount - Get the number of Grids on side B.

INTERFACE:

```
int ESMC_XGridGetSideBGridCount(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The number of Grids on side B.
```

DESCRIPTION:

Get the number of Grids on side B.

The arguments are:

**xgrid** The XGrid from which to get the information.

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

---

### 21.4.6 ESMC_XGridGetSideBMeshCount - Get the number of Meshes on side B.

INTERFACE:

```
int ESMC_XGridGetSideBMeshCount(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The number of Meshes on side B.
```

DESCRIPTION:

Get the number of Meshes on side B.

The arguments are:

**xgrid** The XGrid from which to get the information.

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

---

### 21.4.7 ESMC_XGridGetDimCount - Get the dimension of the XGrid.

INTERFACE:

```
int ESMC_XGridGetDimCount(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The dimension of the XGrid.
```

DESCRIPTION:

Get the dimension of the XGrid.

The arguments are:

**xgrid** The XGrid from which to get the information.

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

---

### 21.4.8 ESMC_XGridGetElementCount - Get the number of elements in the XGrid.

INTERFACE:

```
int ESMC_XGridGetElementCount(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
       The number of elements in the XGrid.
```

DESCRIPTION:

Get the number of elements in the XGrid.

The arguments are:

**xgrid** The XGrid from which to get the information.

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

---

### 21.4.9 ESMC_XGridGetMesh - Get the Mesh representation of the XGrid.

INTERFACE:

```
ESMC_Mesh ESMC_XGridGetMesh(
  ESMC_XGrid xgrid,     // in
  int *rc               // out
);
```

*RETURN VALUE:*

```
    The ESMC_Mesh object representing the XGrid.
```

DESCRIPTION:

Get the ESMC_Mesh object representing the XGrid.

The arguments are:

**xgrid** The xgrid from which to get the information.

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

---

### 21.4.10 ESMC_XGridGetElementArea - Get the area of elements in the XGrid.

INTERFACE:

```
void ESMC_XGridGetElementArea(
  ESMC_XGrid xgrid,     // in
  ESMC_R8 *area,        // out
  int *rc               // out
);
```

*RETURN VALUE:*

```
       The number of elements in the XGrid.
```

DESCRIPTION:

Get the number of elements in the XGrid.

The arguments are:

**xgrid** The XGrid from which to get the information.

**area** An array to fill with element areas. The array must be allocated to size elementCount.

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

---

### 21.4.11 ESMC_XGridGetElementCentroid - Get the centroid of elements in the XGrid.

INTERFACE:

```
void ESMC_XGridGetElementCentroid(
  ESMC_XGrid xgrid,      // in
  ESMC_R8 *centroid,     // out
  int *rc                // out
);
```

*RETURN VALUE:*

```
The number of elements in the XGrid.
```

DESCRIPTION:

Get the centroid for each element in the exchange grid.

The arguments are:

**xgrid** The XGrid from which to get the information.

**centroid** An array to fill with element centroids. The array must be allocated to size elementCount*dimCount.

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

---

### 21.4.12 ESMC_XGridGetSparseMatA2X - Get the sparse matrix that goes from a side A grid to the exchange grid.

INTERFACE:

```
void ESMC_XGridGetSparseMatA2X(
                              ESMC_XGrid xgrid,      // in
                              int sideAIndex,        // in
                              int *factorListCount,  // out
                              double **factorList,   // out
                              int **factorIndexList, // out
                              int *rc);
```

*RETURN VALUE:*

```
N/A
```

DESCRIPTION:

Get the sparse matrix that goes from a side A grid to the exchange grid.

The arguments are:

**xgrid** The XGrid from which to get the information.

**sideAIndex** The 0 based index of the Grid/Mesh on side A to get the sparse matrix for. If a priority has been specified for Grids and Meshes, then this index is in priority order. If no priority was specified, then the all the Grids are first followed by all the Meshes in the order they were passed into the XGrid create call.

**factorListCount** The size of the sparse matrix.

**factorList** A pointer to the list of factors for the requested sparse matrix. The list is of size `factorListCount`. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

**factorIndexList** A pointer to the list of indices for the requested sparse matrix. The list is of size 2*`factorListCount`. For each pair of entries in this array: entry 0 is the sequence index in the source grid, and entry 1 is the sequence index in the destination grid. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

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

---

### 21.4.13 ESMC_XGridGetSparseMatX2A - Get the sparse matrix that goes from the exchange grid to a side A grid.

INTERFACE:

```
void ESMC_XGridGetSparseMatX2A(
                                ESMC_XGrid xgrid,      // in
                                int sideAIndex,        // in
                                int *factorListCount,  // out
                                double **factorList,   // out
                                int **factorIndexList, // out
                                int *rc);
```

*RETURN VALUE:*

`N/A`

DESCRIPTION:

Get the sparse matrix that goes from the exchange grid to a side A grid.

The arguments are:

**xgrid** The XGrid from which to get the information.

**sideAIndex** The 0 based index of the Grid/Mesh on side A to get the sparse matrix for. If a priority has been specified for Grids and Meshes, then this index is in priority order. If no priority was specified, then the all the Grids are first followed by all the Meshes in the order they were passed into the XGrid create call.

**factorListCount** The size of the sparse matrix.

**factorList** A pointer to the list of factors for the requested sparse matrix. The list is of size `factorListCount`. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

**factorIndexList** A pointer to the list of indices for the requested sparse matrix. The list is of size 2*factorListCount. For each pair of entries in this array: entry 0 is the sequence index in the source grid, and entry 1 is the sequence index in the destination grid. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

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

---

### 21.4.14 ESMC_XGridGetSparseMatB2X - Get the sparse matrix that goes from a side B grid to the exchange grid.

INTERFACE:

```
void ESMC_XGridGetSparseMatB2X(
                                ESMC_XGrid xgrid,      // in
                                int sideBIndex,        // in
                                int *factorListCount,  // out
                                double **factorList,   // out
                                int **factorIndexList, // out
                                int *rc);
```

*RETURN VALUE:*

N/A

DESCRIPTION:

Get the sparse matrix that goes from a side B grid to the exchange grid.

The arguments are:

**xgrid** The XGrid from which to get the information.

**sideBIndex** The 0 based index of the Grid/Mesh on side B to get the sparse matrix for. If a priority has been specified for Grids and Meshes, then this index is in priority order. If no priority was specified, then the all the Grids are first followed by all the Meshes in the order they were passed into the XGrid create call.

**factorListCount** The size of the sparse matrix.

**factorList** A pointer to the list of factors for the requested sparse matrix. The list is of size factorListCount. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

**factorIndexList** A pointer to the list of indices for the requested sparse matrix. The list is of size 2*factorListCount. For each pair of entries in this array: entry 0 is the sequence index in the source grid, and entry 1 is the sequence index in the destination grid. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

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

---

#### 21.4.15 ESMC_XGridGetSparseMatX2B - Get the sparse matrix that goes from the exchange grid to a side B grid.

INTERFACE:

```
void ESMC_XGridGetSparseMatX2B(
                                ESMC_XGrid xgrid,      // in
                                int sideBIndex,        // in
                                int *factorListCount,  // out
                                double **factorList,   // out
                                int **factorIndexList, // out
                                int *rc);
```

*RETURN VALUE:*

```
N/A
```

DESCRIPTION:

Get the sparse matrix that goes from the exchange grid to a side B grid.

The arguments are:

**xgrid** The XGrid from which to get the information.

**sideBIndex** The 0 based index of the Grid/Mesh on side B to get the sparse matrix for. If a priority has been specified for Grids and Meshes, then this index is in priority order. If no priority was specified, then the all the Grids are first followed by all the Meshes in the order they were passed into the XGrid create call.

**factorListCount** The size of the sparse matrix.

**factorList** A pointer to the list of factors for the requested sparse matrix. The list is of size `factorListCount`. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

**factorIndexList** A pointer to the list of indices for the requested sparse matrix. The list is of size 2*`factorListCount`. For each pair of entries in this array: entry 0 is the sequence index in the source grid, and entry 1 is the sequence index in the destination grid. To save space this is a pointer to the internal xgrid memory for this list. Don't deallocate it.

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

# 22 DistGrid Class

## 22.1 Description

The ESMF DistGrid class sits on top of the DELayout class (not currently directly accessible through the ESMF C API) 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 DELayout 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 of logically rectangular (LR) sub-domains or *tiles*. The DistGrid create methods allow the specification of such a multi-tile 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 tiles 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.

## 22.2 Class API

### 22.2.1 ESMC_DistGridCreate - Create a DistGrid

INTERFACE:

```
ESMC_DistGrid ESMC_DistGridCreate(
  ESMC_InterArrayInt minIndexInterfaceArg,  // in
  ESMC_InterArrayInt maxIndexInterfaceArg,  // in
  int *rc                                   // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_DistGrid object.
```

DESCRIPTION:

Create an `ESMC_DistGrid` from a single logically rectangular (LR) tile with default decomposition. The default decomposition is `deCount`$\times 1 \times ... \times 1$, where `deCount` is the number of DEs in a default DELayout, equal to `petCount`. This means that the default decomposition will be into as many DEs as there are PETs, with 1 DE per PET.

The arguments are:

**minIndex** Global coordinate tuple of the lower corner of the tile.

**maxIndex** Global coordinate tuple of the upper corner of the tile.

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

---

#### 22.2.2 ESMC_DistGridDestroy - Destroy a DistGrid

INTERFACE:

```
int ESMC_DistGridDestroy(
  ESMC_DistGrid *distgrid         // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Destroy an `ESMC_DistGrid` object.

The arguments are:

**distgrid** `ESMC_DistGrid` object to be destroyed.

---

#### 22.2.3 ESMC_DistGridPrint - Print a DistGrid

INTERFACE:

```
int ESMC_DistGridPrint(
  ESMC_DistGrid distgrid          // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Print internal information of the specified `ESMC_DistGrid` object.

The arguments are:

**distgrid** `ESMC_DistGrid` object to be destroyed.

# 23 RouteHandle Class

## 23.1 Description

The ESMF RouteHandle class provides a unified interface for all route-based communication methods across the Field, FieldBundle, Array, and ArrayBundle classes. All route-based communication methods implement a pre-computation

step, returning a RouteHandle, an execution step, and a release step. Typically the pre-computation, or Store() step will be a lot more expensive (both in memory and time) than the execution step. The idea is that once precomputed, a RouteHandle will be executed many times over during a model run, making the execution time a very performance critical piece of code. In ESMF, Regridding, Redisting, and Haloing are implemented as route-based communication methods. The following sections discuss the RouteHandle concepts that apply uniformly to all route-based communication methods, across all of the above mentioned classes.

## 23.2 Use and Examples

The user interacts with the RouteHandle class through the route-based communication methods of Field, FieldBundle, Array, and ArrayBundle. The usage of these methods are described in detail under their respective class documentation section. The following examples focus on the RouteHandle aspects common across classes and methods.

## 23.3 Restrictions and Future Work

- **Non-blocking** communication via the `routesyncflag` option is implemented for Fields and Arrays. It is *not* available for FieldBundles and ArrayBundles. The user is advised to use the VMEpoch approach for all cases to achive asynchronicity.
- The **dynamic masking** feature currently has the following limitations:
  - Only available for `ESMF_TYPEKIND_R8` and `ESMF_TYPEKIND_R4` Fields and Arrays.
  - Only available through the `ESMF_FieldRegrid()` and `ESMF_ArraySMM()` methods.
  - Destination objects that have undistributed dimensions *after* any distributed dimension are not supported.
  - No check is implemented that ensure the user-provided RouteHandle object is suitable for dynamic masking.

## 23.4 Design and Implementation Notes

Internally all route-based communication calls are implemented as sparse matrix multiplications. The precompute step for all of the supported communication methods can be broken up into three steps:

1. Construction of the sparse matrix for the specific communication method.
2. Generation of the communication pattern according to the sparse matrix.
3. Encoding of the communication pattern for each participating PET in form of an XXE stream.

## 23.5 Class API

### 23.5.1 ESMC_RouteHandleCreateFromFile - Create a RouteHandle

INTERFACE:

```
ESMC_RouteHandle ESMC_RouteHandleCreateFromFile(
  char *filename,
  int *rc
);
```

*RETURN VALUE:*

```
    ESMC_RouteHandle
```

DESCRIPTION:

Create an `ESMC_RouteHandle` object.

The arguments are:

**filename** The file that describes the `ESMC_RouteHandle` object.

---

### 23.5.2 ESMC_RouteHandlePrint - Print a RouteHandle

INTERFACE:

```
int ESMC_RouteHandlePrint(
  ESMC_RouteHandle rh            // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Print internal information of the specified `ESMC_RouteHandle` object.

The arguments are:

**rh** `ESMC_RouteHandle` object to be printed.

---

### 23.5.3 ESMC_RouteHandleWrite - Write a RouteHandle to file

INTERFACE:

```
int ESMC_RouteHandleWrite(
  ESMC_RouteHandle rh,          // in
  char *filename                // in
);
```

*RETURN VALUE:*

Return code; equals ESMF_SUCCESS if there are no errors.

## DESCRIPTION:

Write ESMC_RouteHandle object to file to save regrid information for fast input for regridding operations during runtime.

The arguments are:

**rh** ESMC_RouteHandle object to be printed.

**filename** The name of the file for writing the ESMC_RouteHandle object.

Part V

# Infrastructure: Utilities

# 24 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 time interval representation, as well as a higher-level utility, a clock, that controls model time stepping.

The ESMF Config class provides configuration management based on NASA DAO's Inpak package, a collection of methods for accessing files containing input parameters stored in an ASCII format.

The ESMF LogErr class consists of a method for writing error, warning, and informational messages to a default Log file that is created during ESMF initialization.

The ESMF VM (Virtual Machine) class provides methods for querying information about a VM. A VM 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 class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component. In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods.

# 25 Time Manager Utility

The ESMF Time Manager utility includes software for time and time interval representation, as well as model time advancement. Since multi-component geophysical applications often require synchronization across the time management schemes of the individual components, the Time Manager's standard calendars and consistent time representation promote component interoperability.

**Key Features**
Drift-free timekeeping through an integer-based internal time representation. Both integers and reals can be specified at the interface.
Support for many calendar kinds.
Support for both concurrent and sequential modes of component execution.

## 25.1 Time Manager Classes

There are four 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) are supported.
- **Time** A Time represents a time instant in a particular calendar, such as November 28, 1964, at 7:00pm 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 3 hours. 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 current simulation time and time step. Clock methods include incrementing the current time, and printing the its contents.

## 25.2 Calendar

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.

**Modified Julian Day** The Modified 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 26.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

## 25.3 Time Instants and TimeIntervals

TimeIntervals and Time instants (simply called Times) are the computational building blocks of the Time Manager utility. Times support different queries for values of individual Time components such as year and hour. See Sections 27.1 and 28.1, respectively, for use of Times and TimeIntervals.

## 25.4 Clocks

It is useful to identify a higher-level concept to repeatedly step a Time forward by a TimeInterval. 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, and to query the value of quantities such as the current time and the number of time steps taken. Applications may contain temporary or multiple Clocks. Section 29.1 describes the use of Clocks in detail.

# 26 Calendar Class

## 26.1 Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, Julian Day, Modified Julian Day, no-leap, 360-day, and no-calendar. Brief descriptions are provided for each calendar below.

## 26.2 Constants

### 26.2.1 ESMC_CALKIND

DESCRIPTION:
Supported calendar kinds.

The type of this flag is:

`type(ESMF_CalKind_Flag)`

The valid values are:

**ESMC_CALKIND_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.

**ESMC_CALKIND_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 does *not* handle leap seconds.)

**ESMC_CALKIND_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.

**ESMC_CALKIND_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.

**ESMC_CALKIND_MODJULIANDAY** *Valid range: +/- 1x10*$^{14}$
The Modified Julian Day (MJD) was introduced by space scientists in the late 1950's. It is defined as an offset from the Julian Day (JD):

MJD = JD - 2400000.5

The half day is subtracted so that the day starts at midnight.

**ESMC_CALKIND_NOCALENDAR** *Valid range: machine limits*
The no-calendar option simply tracks the elapsed model time in seconds.

**ESMC_CALKIND_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.

## 26.3 Class API

### 26.3.1 ESMC_CalendarCreate - Create a Calendar

INTERFACE:

```
ESMC_Calendar ESMC_CalendarCreate(
  const char *name,                       // in
  enum ESMC_CalKind_Flag calkindflag,     // in
  int *rc                                 // out
);
```

*RETURN VALUE:*

```
    Newly created ESMC_Calendar object.
```

DESCRIPTION:

Creates and sets a `ESMC_Calendar` object to the given built-in `ESMC_CalKind_Flag`.

The arguments are:

**[name]** The name for the newly created Calendar. If not specified, i.e. NULL, a default unique name will be generated: "CalendarNNN" where NNN is a unique sequence number from 001 to 999.

**calkindflag** The built-in `ESMC_CalKind_Flag`. Valid values are:
`ESMC_CALKIND_360DAY`,
`ESMC_CALKIND_GREGORIAN`,
`ESMC_CALKIND_JULIAN`,
`ESMC_CALKIND_JULIANDAY`,
`ESMC_CALKIND_MODJULIANDAY`,
`ESMC_CALKIND_NOCALENDAR`,
and `ESMC_CALKIND_NOLEAP`.
See Section 26.2 for a description of each calendar kind.

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

---

### 26.3.2 ESMC_CalendarDestroy - Destroy a Calendar

INTERFACE:

```
int ESMC_CalendarDestroy(
  ESMC_Calendar *calendar   // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this `ESMC_Calendar`.

The arguments are:

**calendar** Destroy contents of this `ESMC_Calendar`.

---

### 26.3.3 ESMC_CalendarPrint - Print a Calendar

INTERFACE:

```
int ESMC_CalendarPrint(
  ESMC_Calendar calendar   // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints out an `ESMC_Calendar`'s properties to `stdio`, in support of testing and debugging.

The arguments are:

**calendar** `ESMC_Calendar` object to be printed.

# 27 Time Class

## 27.1 Description

A Time represents a specific point in time.

There are Time methods defined for setting and getting a Time.

A Time that is specified in hours does not need to be associated with a standard calendar; use ESMC_CALKIND_NOCALENDAR. A Time whose specification includes time units of a year must be associated with a standard calendar. The ESMF representation of a calendar, the Calendar class, is described in Section 26.1. The `ESMC_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.

## 27.2 Class API

### 27.2.1 ESMC_TimeGet - Get a Time value

INTERFACE:

```
int ESMC_TimeGet(
  ESMC_Time time,                           // in
  ESMC_I4 *yy,                              // out
  ESMC_I4 *h,                               // out
  ESMC_Calendar *calendar,                  // out
  enum ESMC_CalKind_Flag *calkindflag,      // out
  int *timeZone                             // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Gets the value of an `ESMC_Time` in units specified by the user.

The arguments are:

**time** `ESMC_Time` object to be queried.

**[yy]** Integer year (>= 32-bit).

**[h]** Integer hours.

**[calendar]** Associated `ESMC_Calendar`.

**[calkindflag]** Associated `ESMC_CalKind_Flag`.

### 27.2.2 ESMC_TimePrint - Print a Time

INTERFACE:

```
int ESMC_TimePrint(
  ESMC_Time time   // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints out an `ESMC_Time`'s properties to `stdio`, in support of testing and debugging.

The arguments are:

**time** `ESMC_Time` object to be printed.

---

### 27.2.3 ESMC_TimeSet - Initialize or set a Time

INTERFACE:

```
int ESMC_TimeSet(
  ESMC_Time *time,                        // inout
  ESMC_I4 yy,                             // in
  ESMC_I4 h,                              // in
  ESMC_Calendar calendar,                 // in
  enum ESMC_CalKind_Flag calkindflag,     // in
  int timeZone                            // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Initializes an `ESMC_Time` with a set of user-specified units.

The arguments are:

**time** `ESMC_Time` object to initialize or set.

**yy** Integer year (>= 32-bit).

**h** Integer hours.

**calendar** Associated `ESMC_Calendar`. If not created, defaults to calendar `ESMC_CALKIND_NOCALENDAR` or default specified in `ESMC_Initialize()`. If created, has precedence over calkindflag below.

**calkindflag** Specifies associated `ESMC_Calendar` if calendar argument above not created. More convenient way of specifying a built-in calendar kind.

# 28 TimeInterval Class

## 28.1 Description

A TimeInterval represents a period between time instants. It can be either positive or negative.

There are TimeInterval methods defined for setting and getting a TimeInterval, for printing the contents of a TimeInterval.

The class used to represent time instants in ESMF is Time, and this class is frequently used in operations along with TimeIntervals. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval.

TimeIntervals are used by other parts of the ESMF timekeeping system, such as Clocks; see Section 29.1.

## 28.2 Class API

### 28.2.1 ESMC_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
int ESMC_TimeIntervalGet(
  ESMC_TimeInterval timeinterval,   // in
  ESMC_I8 *s_i8,                    // out
  ESMC_R8 *h_r8                     // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Gets the value of an `ESMC_TimeInteval` in units specified by the user.

The arguments are:

**timeinterval** `ESMC_TimeInterval` object to be queried.

**[s_i8]** Integer seconds (large, >= 64-bit).

**[h_r8]** Double precision hours.

---

### 28.2.2 ESMC_TimeIntervalPrint - Print a TimeInterval

INTERFACE:

```
int ESMC_TimeIntervalPrint(
  ESMC_TimeInterval timeinterval   // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints out an `ESMC_TimeInterval`'s properties to `stdio`, in support of testing and debugging.

The arguments are:

**timeinterval** `ESMC_TimeInterval` object to be printed.

---

### 28.2.3 ESMC_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
int ESMC_TimeIntervalSet(
  ESMC_TimeInterval *timeinterval,   // inout
  ESMC_I4 h                          // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Sets the value of the `ESMC_TimeInterval` in units specified by the user.

The arguments are:

**timeinterval** `ESMC_TimeInterval` object to initialize or set.

**h** Integer hours.

# 29 Clock Class

## 29.1 Description

The Clock class advances model time and tracks its associated date on a specified Calendar. It stores start time, stop time, current time, and a time step.

There are methods for setting and getting the Times associated with a Clock. Methods are defined for advancing the Clock's current time and printing a Clock's contents.

## 29.2 Class API

### 29.2.1 ESMC_ClockAdvance - Advance a Clock's current time by one time step

INTERFACE:

```
int ESMC_ClockAdvance(
  ESMC_Clock clock   // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Advances the `ESMC_Clock`'s current time by one time step.

The arguments are:

**clock** `ESMC_Clock` object to be advanced.

---

### 29.2.2 ESMC_ClockCreate - Create a Clock

INTERFACE:

```
ESMC_Clock ESMC_ClockCreate(
  const char *name,             // in
  ESMC_TimeInterval timeStep,   // in
  ESMC_Time startTime,          // in
  ESMC_Time stopTime,           // in
  int *rc                       // out
);
```

*RETURN VALUE:*

```
Newly created ESMC_Clock object.
```

DESCRIPTION:

Creates and sets the initial values in a new `ESMC_Clock` object.

The arguments are:

**[name]** The name for the newly created Clock. If not specified, i.e. NULL, a default unique name will be generated: "ClockNNN" where NNN is a unique sequence number from 001 to 999.

**timeStep** The `ESMC_Clock`'s time step interval, which can be positive or negative.

**startTime** The `ESMC_Clock`'s starting time. Can be less than or or greater than stopTime, depending on a positive or negative timeStep, respectively, and whether a stopTime is specified; see below.

**stopTime** The `ESMC_Clock`'s stopping time. Can be greater than or less than the startTime, depending on a positive or negative timeStep, respectively.

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

---

### 29.2.3 ESMC_ClockDestroy - Destroy a Clock

INTERFACE:

```
int ESMC_ClockDestroy(
  ESMC_Clock *clock   // inout
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Releases all resources associated with this `ESMC_Clock`.

The arguments are:

**clock** Destroy contents of this `ESMC_Clock`.

---

### 29.2.4 ESMC_ClockGet - Get a Clock's properties

INTERFACE:

```
int ESMC_ClockGet(
  ESMC_Clock clock,                  // in
  ESMC_TimeInterval *currSimTime,    // out
  ESMC_I8 *advanceCount              // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Gets one or more of the properties of an `ESMC_Clock`.

The arguments are:

**clock** `ESMC_Clock` object to be queried.

**[currSimTime]** The current simulation time.

**[advanceCount]** The number of times the `ESMC_Clock` has been advanced.

---

### 29.2.5 ESMC_ClockPrint - Print the contents of a Clock

INTERFACE:

```
int ESMC_ClockPrint(
  ESMC_Clock clock   // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Prints out an `ESMC_Clock`'s properties to `stdio`, in support of testing and debugging.

The arguments are:

**clock** `ESMC_Clock` object to be printed.

# 30 Config Class

## 30.1 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.

### 30.1.1 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.

## 30.2 Class API

### 30.2.1 ESMC_ConfigCreate - Create a Config object

INTERFACE:

```
ESMC_Config ESMC_ConfigCreate(
  int* rc                     // out
);
```

*RETURN VALUE:*

```
   ESMC_Config*  to newly allocated ESMC_Config
```

DESCRIPTION:

Creates an `ESMC_Config` for use in subsequent calls.

The arguments are:

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

---

### 30.2.2 ESMC_ConfigDestroy - Destroy a Config object

INTERFACE:

```
int ESMC_ConfigDestroy(
  ESMC_Config* config        // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Destroys the `config` object.

The arguments are:

**config** Already created `ESMC_Config` object to destroy.

---

### 30.2.3 ESMC_ConfigCreateFromSection - Create a Config object from a section of an existing Config object

INTERFACE:

```
ESMC_Config ESMC_ConfigCreateFromSection(
  ESMC_Config config,       // in
  const char* olabel,       // in
  const char* clabel,       // in
  int* rc                   // out
);
```

*RETURN VALUE:*

```
    ESMC_Config*  to newly allocated ESMC_Config
```

DESCRIPTION:

Instantiates an `ESMC_Config` object from a section of an existing `ESMC_Config` object delimited by `openlabel` and `closelabel`. An error is returned if neither of the input labels is found in input `config`.

Note that a section is intended as the content of a given `ESMC_Config` object delimited by two distinct labels. Such content, as well as each of the surrounding labels, are still within the scope of the parent `ESMC_Config` object. Therefore, including in a section labels used outside that section should be done carefully to prevent parsing conflicts.

The arguments are:

**config** The input `ESMC_Config` object.

**openlabel** Label marking the beginning of a section in `config`.

**closelabel** Label marking the end of a section in `config`.

---

### 30.2.4 ESMC_ConfigFindLabel - Find a label

INTERFACE:

```
int ESMC_ConfigFindLabel(
  ESMC_Config config,        // in
  const char* label,         // in
  int *isPresent             // out
);
```

*RETURN VALUE:*

```
    Return code; equals ESMF_SUCCESS if there are no errors.
    If label not found, and the {\tt isPresent} pointer is {\tt NULL},
    an error will be returned.
```

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 `ESMC_Config` object.

**label** Identifying label.

**[isPresent]** Label presence flag. (optional). If non-NULL, the target is set to 1 when the label is found; otherwise set to 0.

---

### 30.2.5 ESMC_ConfigFindNextLabel - Find a label

INTERFACE:

```
int ESMC_ConfigFindNextLabel(
  ESMC_Config config,        // in
  const char* label,         // in
  int *isPresent             // out
);
```

*RETURN VALUE:*

```
    Return code; equals ESMF_SUCCESS if there are no errors.
    If label not found, and the {\tt isPresent} pointer is {\tt NULL},
    an error will be returned.
```

DESCRIPTION:

Finds the `label` (key) string in the `config` object, starting from the current position pointer.

This method is equivalent to `ESMC_ConfigFindLabel`, but the search is performed starting from the current position pointer.

The arguments are:

**config** Already created `ESMC_Config` object.

**label** Identifying label.

**isPresent** If non-NULL, the address specified is given a value of 1 if the label is found, and 0 when the label is not found.

---

### 30.2.6 ESMC_ConfigGetDim - Get table sizes

INTERFACE:

```
int ESMC_ConfigGetDim(
  ESMC_Config config,         // in
  int* lineCount,             // out
  int* columnCount,           // out
  ...                         // optional argument list
);
```

*RETURN VALUE:*

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

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 `ESMC_Config` object.

**lineCount** Returned number of lines in the table.

**columnCount** Returned maximum number of words in a table line.

**[label]** Identifying label (optional).

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

---

### 30.2.7 ESMC_ConfigGetLen - Get the length of the line in words

INTERFACE:

```
int ESMC_ConfigGetLen(
  ESMC_Config config,         // in
  int* wordCount,             // out
  ...                         // optional argument list
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Gets the length of the line in words by counting words disregarding types. Returns the word count as an integer.

The arguments are:

**config** Already created `ESMC_Config` object.

**wordCount** Returned number of words in the line.

**[label]** Identifying label. If not specified, use the current line (optional).

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

---

### 30.2.8 ESMC_ConfigLoadFile - Load resource file into memory

INTERFACE:

```
int ESMC_ConfigLoadFile(
  ESMC_Config config,         // in
  const char* file,           // in
  ...                         // optional argument list
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Resource file with `filename` is loaded into memory.

The arguments are:

**config** Already created `ESMC_Config` object.

**file** Configuration file name.

**[delayout]** `ESMC_DELayout` associated with this `config` object. **NOTE: This argument is not currently supported.

**[unique]** If specified as true, uniqueness of labels are checked and error code set if duplicates found (optional).

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

---

#### 30.2.9 ESMC_ConfigNextLine - Find next line

INTERFACE:

```
int ESMC_ConfigNextLine(
  ESMC_Config config,       // in
  int *tableEnd             // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Selects the next line (for tables).

The arguments are:

**config** Already created `ESMC_Config` object.

**[tableEnd]** End of table mark (::) found flag. Returns 1 when found, and 0 when not found.

---

#### 30.2.10 ESMC_ConfigPrint - Write content of config object to standard output

INTERFACE:

```
int ESMC_ConfigPrint(
  ESMC_Config config        // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Write content of a ESMC_Config object to standard output.

The arguments are:

**config** Already created ESMC_Config object.

---

### 30.2.11 ESMC_ConfigValidate - Validate a Config object

INTERFACE:

```
int ESMC_ConfigValidate(
  ESMC_Config config,         // in
  ...                         // optional argument list
);
```

*RETURN VALUE:*

```
    Return code; equals ESMF_SUCCESS if there are no errors.
    Equals ESMF_RC_ATTR_UNUSED if any unused attributes are found
    with option "unusedAttributes" below.
```

DESCRIPTION:

Checks whether a config object is valid.

The arguments are:

**config** Already created ESMC_Config object.

**[options]** If none specified: simply check that the buffer is not full and the pointers are within range (optional). "unusedAttributes" - Report to the default logfile all attributes not retrieved via a call to ESMC_ConfigGetAttribute() or ESMC_ConfigGetChar(). The attribute name (label) will be logged via ESMC_LogErr with the WARNING log message type. For an array-valued attribute, retrieving at least one value via ESMC_ConfigGetAttribute() or ESMC_ConfigGetChar() constitutes being "used."

Due to this method accepting optional arguments, the final argument must be ESMC_ArgLast.

# 31 Log Class

## 31.1 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.

When ESMF is started with `ESMC_Initialize()`, multiple Log files will be created by PET number. The PET number (in the format `PETx.`) will be prepended to each file name where x is the PET number. The `ESMC_LogWrite()` call is used to issue messages to the log. As part of the call, a message can be tagged as either an informational, warning, or error message.

The messages may be buffered within ESMF before appearing in the log. All messages will be properly flushed to the log files when `ESMC_Finalize()` is called.

## 31.2 Constants

### 31.2.1 ESMC_LOGKIND

DESCRIPTION:
Specifies a single log file, multiple log files (one per PET), or no log files.

The valid values are:

**ESMC_LOGKIND_SINGLE** Use a single log file, combining messages from all of the PETs. Not supported on some platforms.

**ESMC_LOGKIND_MULTI** Use multiple log files — one per PET.

**ESMC_LOGKIND_NONE** Do not issue messages to a log file.

### 31.2.2 ESMC_LOGMSG

DESCRIPTION:

Specifies a message level.

The valid values are:

**ESMC_LOGMSG_INFO** Informational messages

**ESMC_LOGMSG_WARNING** Warning messages

**ESMC_LOGMSG_ERROR** Error messages

**ESMC_LOGMSG_TRACE** Trace messages

**ESMC_LOGMSG_DEBUG** Debug messages

**ESMC_LOGMSG_JSON** JSON format messages

## 31.3 Class API

### 31.3.1 ESMC_LogMsgFoundError - Test for error and write message

INTERFACE:

```
int ESMC_LogMsgFoundError(
  int rcToCheck,         // in
  const char msg[],      // in
  int LINE,              // in
  const char FILE[],     // in
  const char method[],   // in
  int *rcToReturn        // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Test for rcToCheck != ESMF_SUCCESS. Return 1 if error found, 0 otherwise. Write message to log in case of error.

The arguments are:

**msg** The return code to check.

**msg** The message to be written in case of error.

**LINE** The source line.

**FILE** The source file.

**method** The source method.

**rcToReturn** Propagated return code.

---

#### 31.3.2 ESMC_LogSet - Set Log properties

INTERFACE:

```
int ESMC_LogSet(
  int flush       // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Set Log properties.

The arguments are:

**flush** If set to ESMF_TRUE, flush log messages immediately, rather than buffering them. Default is to flush after 10 messages.

---

### 31.3.3 ESMC_LogWrite - Write an entry into the Log file

INTERFACE:

```
int ESMC_LogWrite(
  const char msg[], // in
  int msgtype       // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Write an entry into the Log file.

The arguments are:

**msg** The message to be written.

**msgtype** The message type. This flag is documented in section 31.2.2

# 32 VM Class

## 32.1 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 class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component.

In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods. The VM communication calls are very similar to MPI. Data references in VM communication calls must be provided as raw, language-specific, one-dimensional, contiguous data arrays. 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, the VM communication calls support communication between threaded PETs in a completely transparent fashion.

Many ESMF applications do not interact with the VM class directly very much. The resource management aspect is wrapped completely transparent into the ESMF Component concept. Often the only reason that user code queries a Component object for the associated VM object is to inquire about resource information, such as the `localPet` or the `petCount`. Further, for most applications the use of higher level communication APIs, such as provided by Array and Field, are much more convenient than using the low level VM communication calls.

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 simplest, and most common case, a PET is equivalent to an MPI process. However, ESMF also supports multi-threading, where multiple PETs run as Pthreads inside the same virtual address space (VAS).

## 32.2 Class API

### 32.2.1 ESMC_VMBarrier - block calling PETs until all PETS called

INTERFACE:

```
int ESMC_VMBarrier(
  ESMC_VM vm                    // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Collective ESMC_VM communication call that blocks calling PET until all PETs of the VM context have issued the call.

The arguments are:

**vm** Queried ESMC_VM object.

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

---

### 32.2.2 ESMC_VMBroadcast - Broadcast data across the VM

INTERFACE:

```
int ESMC_VMBroadcast(ESMC_VM vm,
                  void *bcstData,
                  int count,
                  enum ESMC_TypeKind_Flag *typekind,
                  int rootPet);
```

*RETURN VALUE:*

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

DESCRIPTION:

Collective ESMC_VM communication call that broadcasts a contiguous data array from rootPet to all other PETs of the ESMC_VM object.

This method is overloaded for: ESMC_TYPEKIND_I4, ESMC_TYPEKIND_I8, ESMC_TYPEKIND_R4, ESMC_TYPEKIND_R8, ESMC_TYPEKIND_LOGICAL.

The arguments are:

**vm** ESMC_VM object.

**bcstData** Contiguous data array. On rootPet 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 bcstData. Must be the same on all PETs.

**typekind** The typekind of the data to be reduced. See section 34.18 for a list of valid typekind options.

**rootPet** PET that holds data that is being broadcast.

---

### 32.2.3 ESMC_VMGet - Get VM internals

INTERFACE:

```
int ESMC_VMGet(
  ESMC_VM vm,                   // in
  int *localPet,                // out
  int *petCount,                // out
  int *peCount,                 // out
  MPI_Comm *mpiCommunicator,    // out
  int *pthreadsEnabledFlag,     // out
  int *openMPEnabledFlag        // out
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Get internal information about the specified ESMC_VM object.

The arguments are:

**vm** Queried ESMC_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 ESMC_VM object.

**[peCount]** Upon return this holds the number of PEs referenced by the specified ESMC_VM object.

**[mpiCommunicator]** Upon return this holds the MPI intra-communicator used by the specified ESMC_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 ESMC communications.

**[pthreadsEnabledFlag]** A return value of '1' indicates that the ESMF library was compiled with Pthreads enabled. A return value of '0' indicates that Pthreads are disabled in the ESMF library.

**[openMPEnabledFlag]** A return value of '1' indicates that the ESMF library was compiled with OpenMP enabled. A return value of '0' indicates that OpenMP is disabled in the ESMF library.

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

---

### 32.2.4 ESMC_VMGetCurrent - Get current VM

INTERFACE:

```
ESMC_VM ESMC_VMGetCurrent(
  int *rc                       // out
);
```

*RETURN VALUE:*

```
    VM object of the current execution context.
```

DESCRIPTION:

Get the `ESMC_VM` object of the current execution context. Calling `ESMC_VMGetCurrent()` within an ESMC Component, will return the same VM object as `ESMC_GridCompGet(..., vm=vm, ...)` or `ESMC_CplCompGet(..., vm=vm, ...)`.

The main purpose of providing `ESMC_VMGetCurrent()` is to simplify ESMF adoption in legacy code. Specifically, code that uses `MPI_COMM_WORLD` deep within its calling tree can easily be modified to use the correct MPI communicator of the current ESMF execution context. The advantage is that these modifications are very local, and do not require wide reaching interface changes in the legacy code to pass down the ESMF component object, or the MPI communicator.

The use of `ESMC_VMGetCurrent()` is strongly discouraged in newly written Component code. Instead, the ESMF Component object should be used as the appropriate container of ESMF context information. This object should be passed between the subroutines of a Component, and be queried for any Component specific information.

Outside of a Component context, i.e. within the driver context, the call to `ESMC_VMGetCurrent()` is identical to `ESMC_VMGetGlobal()`.

The arguments are:

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

---

### 32.2.5 ESMC_VMGetGlobal - Get global VM

INTERFACE:

```
ESMC_VM ESMC_VMGetGlobal(
  int *rc                       // out
);
```

*RETURN VALUE:*

```
    VM object of the global execution context.
```

DESCRIPTION:

Get the global ESMC_VM object. This is the VM object that is created during ESMC_Initialize() and is the ultimate parent of all VM objects in an ESMF application. It is identical to the VM object returned by ESMC_Initialize(..., vm=vm, ...).

The ESMC_VMGetGlobal() call provides access to information about the global execution context via the global VM. This call is necessary because ESMF does not create a global ESMF Component during ESMC_Initialize() that could be queried for information about the global execution context of an ESMF application.

Usage of ESMC_VMGetGlobal() from within Component code is strongly discouraged. ESMF Components should only access their own VM objects through Component methods. Global information, if required by the Component user code, should be passed down to the Component from the driver through the Component calling interface.

The arguments are:

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

---

### 32.2.6 ESMC_VMLogMemInfo - Log memory measurements to file

INTERFACE:

```
int ESMC_VMLogMemInfo(
  const char *prefix                // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Write memory info to the log file.

The arguments are:

**prefix** String value to indicate the location of the memory measurement

---

### 32.2.7 ESMC_VMPrint - Print a VM

INTERFACE:

```
int ESMC_VMPrint(
  ESMC_VM vm                // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Print internal information of the specified `ESMC_VM` object.

The arguments are:

**vm** `ESMC_VM` object to be printed.

---

### 32.2.8 ESMC_VMReduce - Reduce data from across the VM

INTERFACE:

```
int ESMC_VMReduce(ESMC_VM vm,
                  void *sendData,
                  void *recvData,
                  int count,
                  enum ESMC_TypeKind_Flag *typekind,
                  enum ESMC_Reduce_Flag *reduceflag,
                  int rootPet);
```

*RETURN VALUE:*

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

DESCRIPTION:

Collective `ESMC_VM` communication call that reduces a contiguous data array across the `ESMC_VM` object into a contiguous data array of the same <type><kind>. The result array is returned on `rootPet`. Different reduction operations can be specified.

This method is overloaded for: `ESMC_TYPEKIND_I4`, `ESMC_TYPEKIND_I8`, `ESMC_TYPEKIND_R4`, `ESMC_TYPEKIND_R8`.

The arguments are:

**vm** `ESMC_VM` object.

**sendData** Contiguous data array holding data to be sent. All PETs must specify a valid source array.

**recvData** Contiguous data array for data to be received. Only the `recvData` array specified by the `rootPet` will be used by this method.

**count** Number of elements in sendData and recvData. Must be the same on all PETs.

**typekind** The typekind of the data to be reduced. See section 34.18 for a list of valid typekind options.

**reduceflag** Reduction operation. See section 34.14 for a list of valid reduce operations.

**rootPet** PET on which reduced data is returned.

# 33 Profiling and Tracing

## 33.1 Description

### 33.1.1 Profiling

ESMF's built in *profiling* capability collects runtime statistics of an executing ESMF application through both automatic and manual code instrumentation. Timing information for all phases of all ESMF components executing in an application can be automatically collected using the `ESMF_RUNTIME_PROFILE` environment variable (see below for settings). Additionally, arbitrary user-defined code regions can be timed by manually instrumenting code with special API calls. Timing profiles of component phases and user-defined regions can be output in several different formats:

- in text at the end of ESMF Log files
- in separate text file, one per PET (if the ESMF Logs are turned off)
- in a single summary text file that aggregates timings over multiple PETs
- in a binary format for import into the esmf-profiler for profile visualization

The following table lists important environment variables that control aspects of ESMF profiling.

| Environment Variable | Description | Example Values |
|---|---|---|
| `ESMF_RUNTIME_PROFILE` | Enable/disables all profiling functions | `ON` or `OFF` |
| `ESMF_RUNTIME_PROFILE_PETLIST` | Limits profiling to an explicit list of PETs | "`0-9 50 99`" |
| `ESMF_RUNTIME_PROFILE_OUTPUT` | Controls output format of profiles; multiple can be specified in a space separated list | `TEXT`, `SUMMARY`, `BINARY` |

### 33.1.2 Tracing

Whereas profiling collects summary information from an application, *tracing* records a more detailed set of events for later analysis. Trace analysis can be used to understand what happened during a program's execution and is often used for diagnosing problems, debugging, and performance analysis.

ESMF has a built-in tracing capability that records events into special binary log files. Unlike log files written by the `ESMF_Log` class, which are primarily for human consumption (see Section 31.1), the trace output files are recorded in a compact binary representation and are processed by tools to produce various analyses. ESMF event streams are recorded in the Common Trace Format (CTF). CTF traces include one or more event streams, as well as a metadata file describing the events in the streams.

Several tools are available for reading in the CTF traces output by ESMF. Of the tools listed below, the first one is designed specifically for analyzing ESMF applications and the second two are general purpose tools for working with all CTF traces.

- esmf-profiler is a tool that ingests traces from an ESMF application and generates performance profile plots.
- TraceCompass is a general purpose tool for reading, analyzing, and visualizing traces.
- Babeltrace is a command-line tool and library for trace conversion that can read and write CTF traces. Python bindings are available to open CTF traces are iterate through events.

Events that can be captured by the ESMF tracer include the following. Events are recorded with a high-precision timestamp to allow timing analyses.

**phase_enter** indicates entry into an initialize, run, or finalize ESMF component routine

**phase_exit** indicates exit from an initialize, run, or finalize ESMF component routine

**region_enter** indicates entry into a user-defined code region

**region_exit** indicates exit from a user-defined code region

The following table lists important environment variables that control aspects of ESMF tracing.

| Environment Variable | Description | Example Values |
|---|---|---|
| ESMF_RUNTIME_TRACE | Enable/disables all tracing functions | ON or OFF |
| ESMF_RUNTIME_TRACE_CLOCK | Sets the type of clock for timestamping events (see Section **??**). | REALTIME or MONO MONOTONIC_SYNC |
| ESMF_RUNTIME_TRACE_PETLIST | Limits tracing to an explicit list of PETs | "0-9 50 99" |
| ESMF_RUNTIME_TRACE_COMPONENT | Enables/disable tracing of Component phase_enter and phase_exit events | ON or OFF |
| ESMF_RUNTIME_TRACE_FLUSH | Controls frequency of event stream flushing to file | DEFAULT or EAGER |

## 33.2 Restrictions and Future Work

1. **Limited types of trace events.** Currently only a few trace event types are available. The tracer may be extended in the future to record additional types of events.
2. **MPI call profing not available for statically linked executables on Darwin.** Currently the linker on Darwin systems does not support the wrapping of symbols during static linking. In order to access MPI call profiling on Darwin, executables should be linked dynamically in combination with the procedure described in section **??**.

## 33.3 Class API

### 33.3.1 ESMC_TraceRegionEnter - Enter a trace region

INTERFACE:

```
int ESMC_TraceRegionEnter(
  const char* name  // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Enter a named trace region.

The arguments are:

**name** Name of the trace region.

---

### 33.3.2 ESMC_TraceRegionExit - Exit a trace region

INTERFACE:

```
int ESMC_TraceRegionExit(
  const char* name  // in
);
```

*RETURN VALUE:*

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

DESCRIPTION:

Exit a named trace region.

The arguments are:

**name** Name of the trace region.

# Part VI

# References

## References


[1] SCRIP: A Spherical Coordinate Remapping and Interpolation Package. http://oceans11.lanl.gov/trac/SCRIP, last accessed on Dec 4, 2015. Los Alamos Software Release LACC 98-45.

[2] V. Balaji, Jeff Anderson, Isaac Held, Michael Winton, Jeff Durachta, Sergey Malyshev, and Ronald J. Stouffer. The exchange grid: a mechanism for data exchange between earth system components on independent grids. *Parallel Computational Fluid Dynamics: Theory and Applications, Proceedings of the 2005 International Conference on Parallel Computational Fluid Dynamics*, 2006.

[3] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

# Part VII

# Appendices

## 34 Appendix A: Master List of Constants

### 34.1 ESMC_CALKIND

This flag is documented in section 26.2.1.

### 34.2 ESMC_COORDSYS

This flag is documented in section 19.2.1.

### 34.3 ESMC_DECOMP

DESCRIPTION:
Indicates how DistGrid elements are decomposed over DEs.

The type of this flag is:

`type(ESMC_Decomp_Flag)`

The valid values are:

**ESMC_DECOMP_BALANCED** Decompose elements as balanced as possible across DEs. The maximum difference in number of elements per DE is 1, with the extra elements on the lower DEs.

**ESMC_DECOMP_CYCLIC** Decompose elements cyclically across DEs.

**ESMC_DECOMP_RESTFIRST** Divide elements over DEs. Assign the rest of this division to the first DE.

**ESMC_DECOMP_RESTLAST** Divide elements over DEs. Assign the rest of this division to the last DE.

**ESMC_DECOMP_SYMMEDGEMAX** Decompose elements across the DEs in a symmetric fashion. Start with the maximum number of elements at the two edge DEs, and assign a decending number of elements to the DEs as the center of the decomposition is approached from both sides.

`subsection(ESMC_ExtrapMethod_Flag)` DESCRIPTION:
Specify which extrapolation method to use on unmapped destination points after regridding.

The type of this flag is:

`type(ESMC_ExtrapMethod_Flag)`

The valid values are:

**ESMC_EXTRAPMETHOD_NONE** Indicates that no extrapolation should be done.

**ESMC_EXTRAPMETHOD_NEAREST_STOD** Inverse distance weighted average. Here the value of a destination point is the weighted average of the closest N source points. The weight is the reciprocal of the distance of the source point from the destination point raised to a power P. All the weights contributing to one destination point are normalized so that they sum to 1.0. The user can choose N and P when using this method, but defaults are also provided.

**ESMC_EXTRAPMETHOD_NEAREST_IDAVG** Nearest source to destination. Here each destination point is mapped to the closest source point. A given source point may go to multiple destination points, but no destination point will receive input from more than one source point.

## 34.4 ESMC_FILEFORMAT

This flag is documented in section 19.2.6.

`subsection(ESMC_FileMode_Flag)` DESCRIPTION:
Specify which mode to use when writing a weight file.

The type of this flag is:

`type(ESMC_FileMode_Flag)`

The valid values are:

**ESMC_FILEMODE_BASIC** Indicates that only the factorList and factorIndexList should be written.

**ESMC_FILEMODE_WITHAUX** Indicates that grid center coordinates should also be written.

## 34.5 ESMC_GRIDITEM

This flag is documented in section 19.2.2.

## 34.6 ESMC_GRIDSTATUS

This flag is documented in section 19.2.3.

## 34.7 ESMC_INDEX

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

The type of this flag is:

`type(ESMC_IndexFlag)`

The valid values are:

**ESMC_INDEX_DELOCAL** Indicates that DE-local index space starts at lower bound 1 for each DE.

**ESMC_INDEX_GLOBAL** Indicates that global indices are used. This means that DE-local index space starts at the global lower bound for each DE.

**ESMC_INDEX_USER** Indicates that the DE-local index bounds are explicitly set by the user.

## 34.8 ESMC_LINETYPE

DESCRIPTION:
This argument allows the user to select the path of the line which connects two points on the surface of a sphere. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell.

The type of this flag is:

`type(ESMC_LineType_Flag)`

The valid values are:

**ESMC_LINETYPE_CART** Cartesian line. When this option is specified distances are calculated in a straight line through the 3D Cartesian space in which the sphere is embedded. Cells are approximated by 3D planes bounded by 3D Cartesian lines between their corner vertices. When calculating regrid weights, this line type is currently the default for the following regrid methods: ESMC_REGRIDMETHOD_BILINEAR, ESMC_REGRIDMETHOD_PATCH, ESMC_REGRIDMETHOD_NEAREST_STOD, and ESMC_REGRIDMETHOD_NEAREST_DTOS.

**ESMC_LINETYPE_GREAT_CIRCLE** Great circle line. When this option is specified distances are calculated along a great circle path (the shortest distance between two points on a sphere surface). Cells are bounded by great circle paths between their corner vertices. When calculating regrid weights, this line type is currently the default for the following regrid method: ESMC_REGRIDMETHOD_CONSERVE.

## 34.9 ESMC_LOGKIND

This flag is documented in section 31.2.1.

## 34.10 ESMC_LOGMSG

This flag is documented in section 31.2.2.

## 34.11 ESMC_MESHELEMTYPE

This flag is documented in section 20.2.1.

## 34.12 ESMF_METHOD

DESCRIPTION:
Specify standard ESMF Component method.

The type of this flag is:

`type(ESMF_Method_Flag)`

The valid values are:

**ESMF_METHOD_FINALIZE** Finalize method.

**ESMF_METHOD_INITIALIZE** Initialize method.

**ESMF_METHOD_READRESTART** ReadRestart method.

**ESMF_METHOD_RUN** Run method.

**ESMF_METHOD_WRITERESTART** WriteRestart method.

## 34.13 ESMC_POLEKIND

This flag is documented in section 19.2.4.

## 34.14 ESMC_REDUCE

DESCRIPTION:
Indicates reduce operation.

The type of this flag is:

```
type(ESMC_Reduce_Flag)
```

The valid values are:

**ESMC_REDUCE_SUM** Use arithmetic sum to add all data elements.

**ESMC_REDUCE_MIN** Determine the minimum of all data elements.

**ESMC_REDUCE_MAX** Determine the maximum of all data elements.

## 34.15 ESMC_REGION

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

The type of this flag is:

```
type(ESMC_Region_Flag)
```

The valid values are:

**ESMC_REGION_TOTAL** Total allocated memory.

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

**ESMC_REGION_EMPTY** The empty region contains no elements.

## 34.16 ESMC_REGRIDMETHOD

This flag is documented in section 16.2.1.

## 34.17 ESMC_STAGGERLOC

This flag is documented in section 19.2.5.

### 34.18 ESMC_TYPEKIND

DESCRIPTION:
Named constants used to indicate type and kind combinations supported by the overloaded ESMC interfaces. The corresponding Fortran kind-parameter constants are described in the ESMF_TYPEKIND section of Appendices of the ESMF Fortran reference manual.

The type of these named constants is:

```
type(ESMC_TypeKind_Flag)
```

The named constants are:

**ESMC_TYPEKIND_I1** Indicates 1 byte integer.
(Only available if ESMF was built with `ESMF_NO_INTEGER_1_BYTE = FALSE`. This is *not* the default.)

**ESMC_TYPEKIND_I2** Indicates 2 byte integer.
(Only available if ESMF was built with `ESMF_NO_INTEGER_2_BYTE = FALSE`. This is *not* the default.)

**ESMC_TYPEKIND_I4** Indicates 4 byte integer.

**ESMC_TYPEKIND_I8** Indicates 8 byte integer.

**ESMC_TYPEKIND_R4** Indicates 4 byte real.

**ESMC_TYPEKIND_R8** Indicates 8 byte real.

### 34.19 ESMC_UNMAPPEDACTION

DESCRIPTION:
Indicates what action to take with respect to unmapped destination points and the entries of the sparse matrix that correspond to these points.

The type of this flag is:

```
type(ESMC_UnmappedAction_Flag)
```

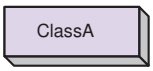
The valid values are:

**ESMC_UNMAPPEDACTION_ERROR** An error is issued when there exist destination points in a regridding operation that are not mapped by corresponding source points.

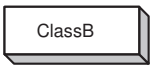
**ESMC_UNMAPPEDACTION_IGNORE** Destination points which do not have corresponding source points are ignored and zeros are used for the entries of the sparse matrix that is generated.

## 35 Appendix B: A Brief Introduction to UML

The schematic below shows the Unified Modeling Language (UML) notation for the class diagrams presented in this *Reference Manual*. For more on UML, see references such as *The Unified Modeling Language Reference Manual*, Rumbaugh et al, [3].




Public class. This is a class whose methods can be called by the user. In Fortran a public class is usually associated with a derived type and a corresponding module that contains class methods and flags.




Private class. This type of class does not have methods that should be called by the user. Like a public class it is usually associated with a derived type and a corresponding module.



A line indicates some sort of association among classes.



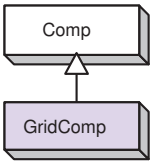
A hollow diamond at one end of a line drawn between classes represents an association called aggregation. Aggregation is a part-whole relationship that can be read as "the class at the end of the line without the diamond is part of the class at the end of the line with the diamond." The class that is the "part" can be created and destroyed separately, and it is usually implemented as a reference contained with the structure of the class that is the "whole."



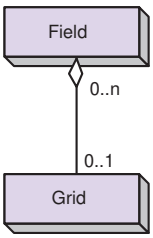
A filled diamond at one end of a line drawn between classes represents an association called composition. Composition is a part-whole relationship that is similar to aggregation, but stronger. It implies that that class that is the "part" is created and destroyed by the class that is the "whole." It is often implemented as a structure within part of the contiguous memory of a larger structure.




Multiplicity indicators at association line ends show how many classes on the one end are associated with how many classes on the other end.




The triangle indicates an inheritance relationship. Inheritance means that a child class shares a set of characteristics (such as the same attributes or methods) with a parent class. The child can specialize and extend the behavior of the parent. This diagram shows a GridComp class that inherits from a more general Comp class.




This simple diagram shows that a public class called Field is associated with another public class, called Grid. The aggregation relationship indicated by the unfilled diamond means that a Field contains a Grid, but that a Grid can be created and destroyed outside of a Field. The diagram multiplicities show that a Field can be associated with no Grid or with one Grid, but that a single Grid can be associated with any number of Fields.

# 36 Appendix C: ESMF Error Return Codes

The tables below show the possible error return codes for Fortran and C methods.

```
==================================
Fortran Symmetric Return Codes 1-500
==================================
```

```
ESMF_SUCCESS                0
ESMF_RC_OBJ_BAD             1
ESMF_RC_OBJ_INIT            2
ESMF_RC_OBJ_CREATE          3
ESMF_RC_OBJ_COR             4
ESMF_RC_OBJ_WRONG           5
ESMF_RC_ARG_BAD             6
ESMF_RC_ARG_RANK            7
ESMF_RC_ARG_SIZE            8
ESMF_RC_ARG_VALUE           9
ESMF_RC_ARG_DUP            10
ESMF_RC_ARG_SAMETYPE       11
ESMF_RC_ARG_SAMECOMM       12
ESMF_RC_ARG_INCOMP         13
ESMF_RC_ARG_CORRUPT        14
ESMF_RC_ARG_WRONG          15
ESMF_RC_ARG_OUTOFRANGE     16
ESMF_RC_ARG_OPT            17
ESMF_RC_NOT_IMPL           18
ESMF_RC_FILE_OPEN          19
ESMF_RC_FILE_CREATE        20
ESMF_RC_FILE_READ          21
ESMF_RC_FILE_WRITE         22
ESMF_RC_FILE_UNEXPECTED    23
ESMF_RC_FILE_CLOSE         24
ESMF_RC_FILE_ACTIVE        25
ESMF_RC_PTR_NULL           26
ESMF_RC_PTR_BAD            27
ESMF_RC_PTR_NOTALLOC       28
ESMF_RC_PTR_ISALLOC        29
ESMF_RC_MEM                30
ESMF_RC_MEM_ALLOCATE       31
ESMF_RC_MEM_DEALLOCATE     32
ESMF_RC_MEMC               33
ESMF_RC_DUP_NAME           34
ESMF_RC_LONG_NAME          35
ESMF_RC_LONG_STR           36
ESMF_RC_COPY_FAIL          37
ESMF_RC_DIV_ZERO           38
ESMF_RC_CANNOT_GET         39
ESMF_RC_CANNOT_SET         40
ESMF_RC_NOT_FOUND          41
ESMF_RC_NOT_VALID          42
ESMF_RC_INTNRL_LIST        43
ESMF_RC_INTNRL_INCONS      44
ESMF_RC_INTNRL_BAD         45
ESMF_RC_SYS                46
ESMF_RC_BUSY               47
ESMF_RC_LIB                48
ESMF_RC_LIB_NOT_PRESENT    49
ESMF_RC_ATTR_UNUSED        50
ESMF_RC_OBJ_NOT_CREATED    51
ESMF_RC_OBJ_DELETED        52
ESMF_RC_NOT_SET            53
```

```
 ESMF_RC_VAL_WRONG         54
 ESMF_RC_VAL_ERRBOUND      55
 ESMF_RC_VAL_OUTOFRANGE    56
 ESMF_RC_ATTR_NOTSET       57
 ESMF_RC_ATTR_WRONGTYPE    58
 ESMF_RC_ATTR_ITEMSOFF     59
 ESMF_RC_ATTR_LINK         60
 ESMF_RC_BUFFER_SHORT      61
 ESMF_RC_TIMEOUT           62
 ESMF_RC_FILE_EXISTS       63
 ESMF_RC_FILE_NOTDIR       64
 ESMF_RC_MOAB_ERROR        65
 ESMF_RC_NOOP              66
 ESMF_RC_NETCDF_ERROR      67

68-499 reserved for future Fortran symmetric return code definitions

====================================
C/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_INCOMP       513
 ESMC_RC_ARG_CORRUPT      514
 ESMC_RC_ARG_WRONG        515
 ESMC_RC_ARG_OUTOFRANGE   516
 ESMC_RC_ARG_OPT          517
 ESMC_RC_NOT_IMPL         518
 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_FILE_ACTIVE      525
 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_MEM_ALLOCATE     531
 ESMC_RC_MEM_DEALLOCATE   532
 ESMC_RC_MEMC             533
 ESMC_RC_DUP_NAME         534
```

```
 ESMC_RC_LONG_NAME        535
 ESMC_RC_LONG_STR         536
 ESMC_RC_COPY_FAIL        537
 ESMC_RC_DIV_ZERO         538
 ESMC_RC_CANNOT_GET       539
 ESMC_RC_CANNOT_SET       540
 ESMC_RC_NOT_FOUND        541
 ESMC_RC_NOT_VALID        542
 ESMC_RC_INTNRL_LIST      543
 ESMC_RC_INTNRL_INCONS    544
 ESMC_RC_INTNRL_BAD       545
 ESMC_RC_SYS              546
 ESMC_RC_BUSY             547
 ESMC_RC_LIB              548
 ESMC_RC_LIB_NOT_PRESENT  549
 ESMC_RC_ATTR_UNUSED      550
 ESMC_RC_OBJ_NOT_CREATED  551
 ESMC_RC_OBJ_DELETED      552
 ESMC_RC_NOT_SET          553
 ESMC_RC_VAL_WRONG        554
 ESMC_RC_VAL_ERRBOUND     555
 ESMC_RC_VAL_OUTOFRANGE   556
 ESMC_RC_ATTR_NOTSET      557
 ESMC_RC_ATTR_WRONGTYPE   558
 ESMC_RC_ATTR_ITEMSOFF    559
 ESMC_RC_ATTR_LINK        560
 ESMC_RC_BUFFER_SHORT     561
 ESMC_RC_TIMEOUT          562
 ESMC_RC_FILE_EXISTS      563
 ESMC_RC_FILE_NOTDIR      564
 ESMC_RC_MOAB_ERROR       565
 ESMC_RC_NOOP             566
 ESMC_RC_NETCDF_ERROR     567

568-999 reserved for future C/C++ symmetric return code definitions

====================================
C/C++ Non-symmetric Return Codes 1000
====================================

 ESMC_RC_OPTARG_BAD      1000
```