Earth System Modeling Framework

ESMF Reference Manual for C

Version 8.2.0 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, 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

October 21, 2021

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

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.

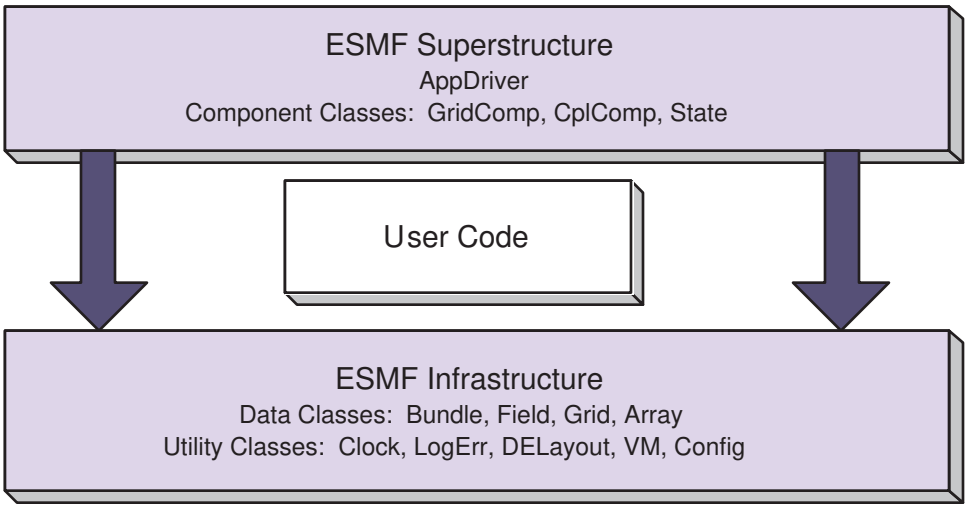
2 The ESMF Reference Manual for C

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

The 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 the method in which they were declared has returned.

An exception to this is when a shallow object, such as a Time, is stored in a deep object such as a Clock. The Clock then carries a copy of the Time in persistent memory. The Time is deallocated with the 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 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.

5.4 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.

5.5 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.

5.6 ESMF Maps

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

5.7 ESMF Specification Classes

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

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

5.8 ESMF Utility Classes

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

- **Attributes**, for storing metadata about Fields, FieldBundles, States, and other classes. (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 **??**.

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

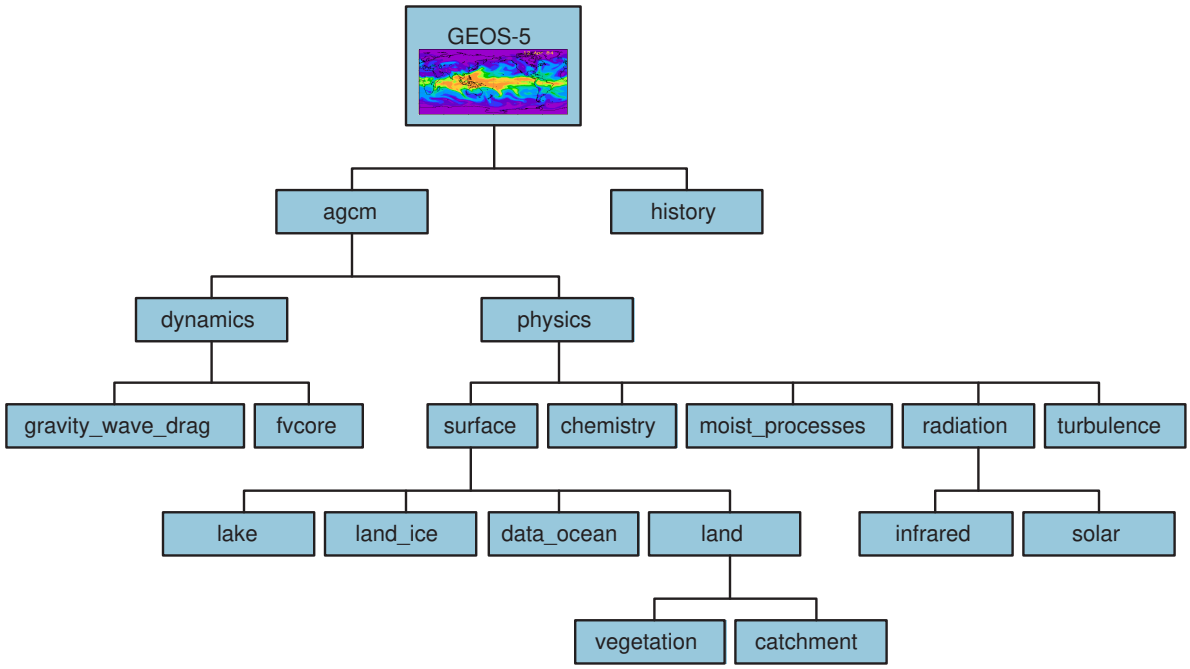
There are a small number of classes in the ESMF superstructure:

- **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.



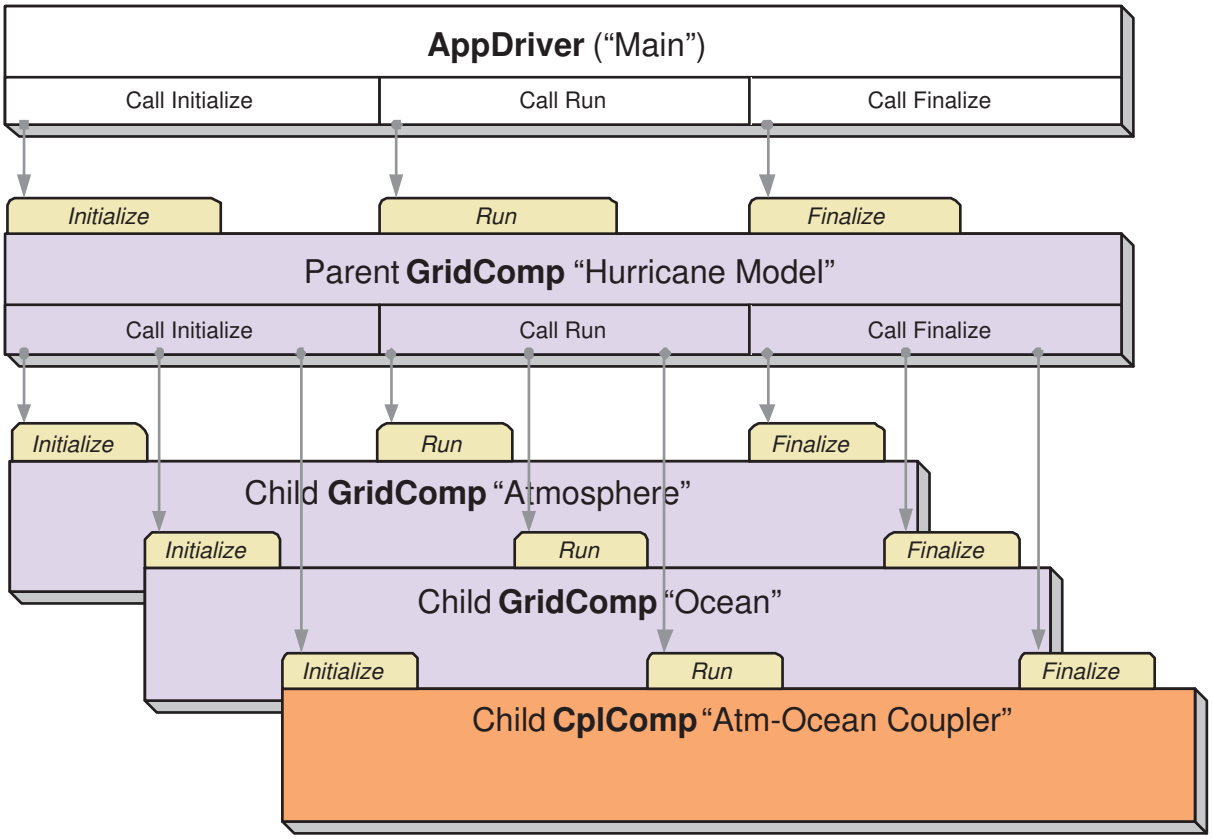
An ESMF coupled application typically involves a parent Gridded Component, two or more child Gridded Components and one or more Coupler Components.

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.

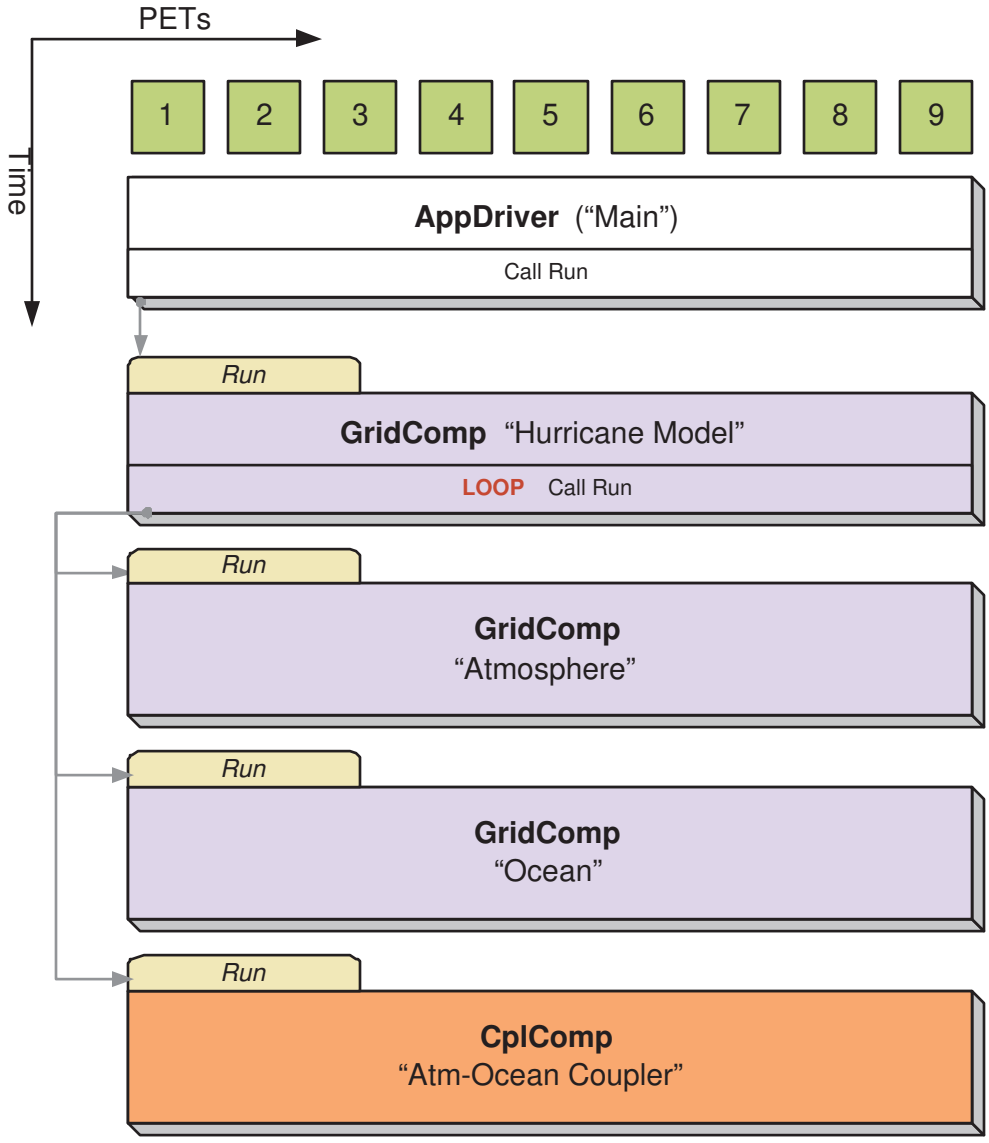


Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The application driver,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

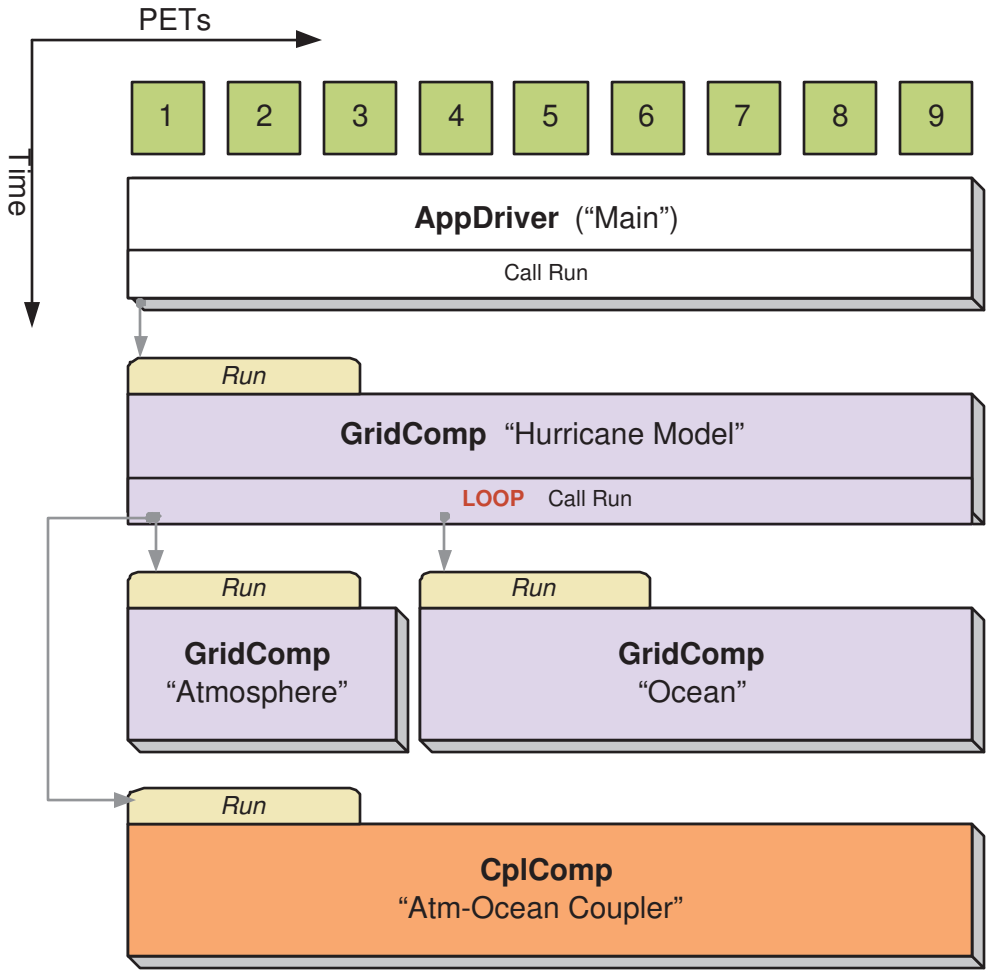
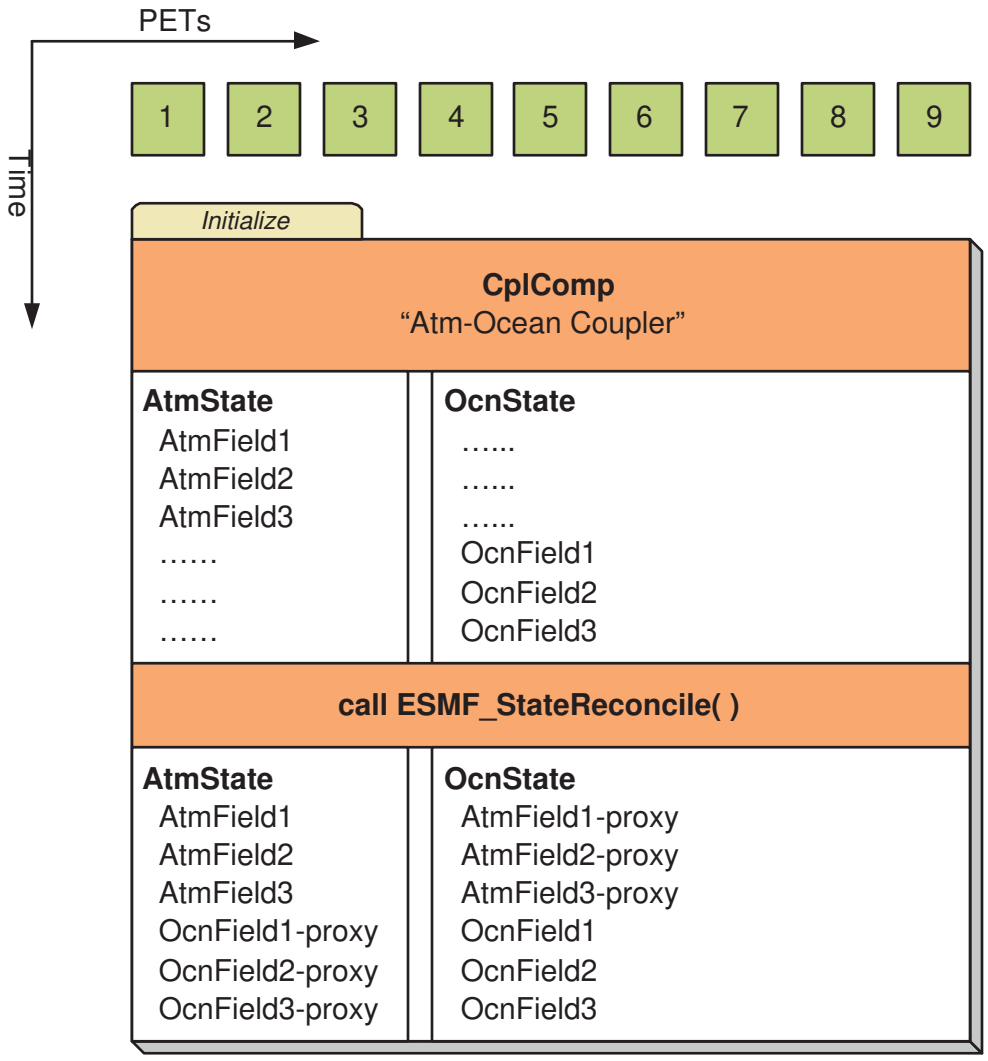
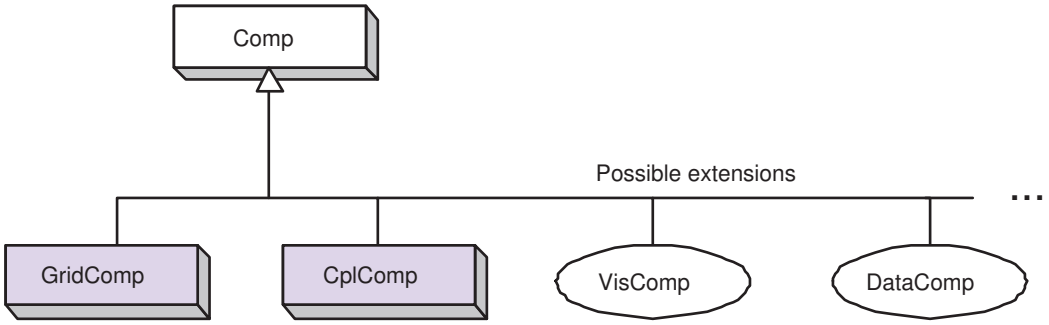


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



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 MPICH, 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 **??**. 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 **??**. 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_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 **??** 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]

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:

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

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 **??** 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_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.

16.4.13 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.

16.4.14 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_ExtrapMethod_Flag *extrapMethod,      // in
    int *extrapNumSrcPnts,                          // in
    float *extrapDistExponent,                      // in
    int *extrapNumLevels,                           // in
    enum ESMC_UnmappedAction_Flag *unmappedaction, // in
    enum ESMC_Logical *ignoreDegenerate,            // 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 **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** 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`.

[extrapMethod] The type of extrapolation. Please see Section **??** 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.

16.4.15 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_UnmappedAction_Flag *unmappedaction, // in
    enum ESMC_Logical *ignoreDegenerate,            // in
    enum ESMC_Logical *create_rh,                   // in
    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 **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** 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`.

[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.

16.4.16 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
```

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.

16.4.17 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

16.4.18 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_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.

17.2.5 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.

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 **??** 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 **??** 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.

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

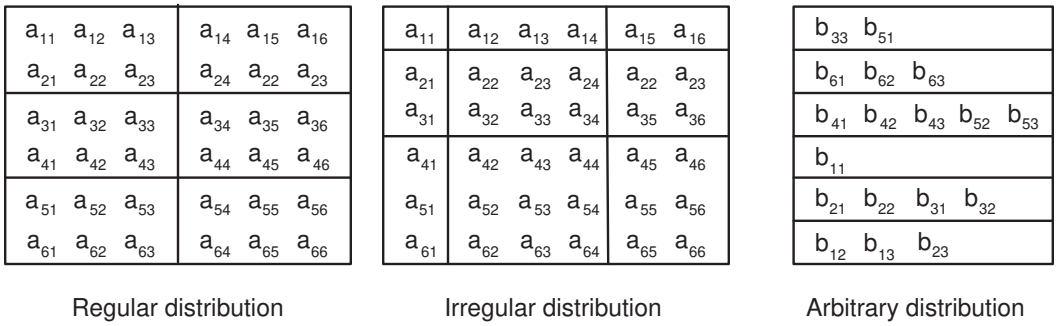


Figure 7: Examples of regular and irregular decomposition of a grid **a** that is 6x6, and an arbitrary decomposition of a grid **b** that is 6x3.

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.

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.

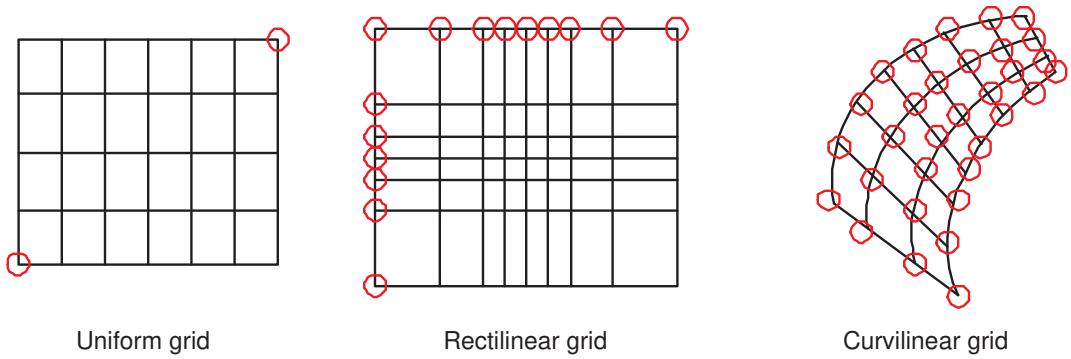


Figure 8: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

	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.

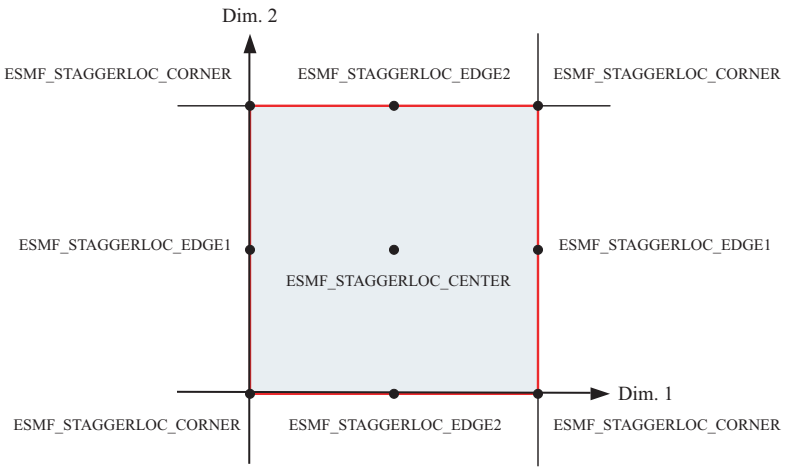


Figure 9: 2D Predefined Stagger Locations

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:

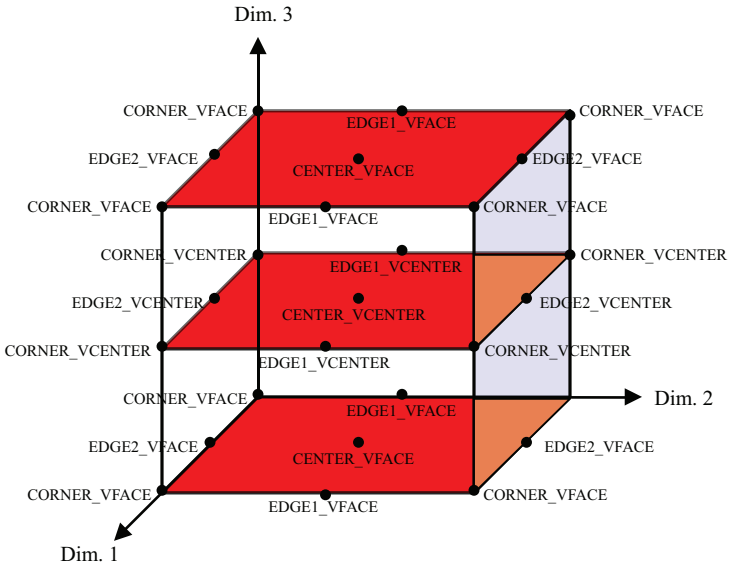


Figure 10: 3D Predefined Stagger Locations

ESMC_STAGGERLOC_CENTER_VCENTER: The center of the 3D cell.

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: