Earth System Modeling Framework

ESMF Reference Manual for C

Version 5.1

ESMF Joint Specification Team: V. Balaji, Byron Boville, Samson Cheung, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Rosalinda de Fainchtein, Brian Eaton, Bob Hallberg, Tom Henderson, Chris Hill, Mark Iredell, Rob Jacob, Phil Jones, Erik Kluzek, Brian Kauffman, Jay Larson, Peggy Li, Fei Liu, John Michalakes, Sylvia Murphy, David Neckels, Ryan O Kuinghttons, Bob Oehmke, Chuck Panaccione, Jim Rosinski, 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

February 28, 2011

Acknowledgements

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

- The Spherical Coordinate Remapping and Interpolation Package (SCRIP) from Los Alamos, which informed the design of our regridding functionality
- The Model Coupling Toolkit (MCT) from Argonne National Laboratory, on which we based our sparse matrix multiply approach to general regridding
- The Inpack configuration attributes package from NASA Goddard, which was adapted for use in ESMF by members of NASA Global Modeling and Assimilation group
- The Flexible Modeling System (FMS) package from GFDL and the Goddard Earth Modeling System (GEMS) from NASA Goddard, both of which provided inspiration for the overall ESMF architecture
- The Common Component Architecture (CCA) effort within the Department of Energy, from which we drew many ideas about how to design components
- The Vector Signal Image Processing Library (VSIPL) and its predecessors, which informed many aspects of our design, and the radar system software design group at Lincoln Laboratory
- The Portable, Extensible Toolkit for Scientific Computation (PETSc) package from Argonne National Laboratories, on which we based our initial makefile system
- The Community Climate System Model (CCSM) and Weather Research and Forecasting (WRF) modeling groups at NCAR, who have provided valuable feedback on the design and implementation of the framework

Contents

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 insitutional and national bounds.

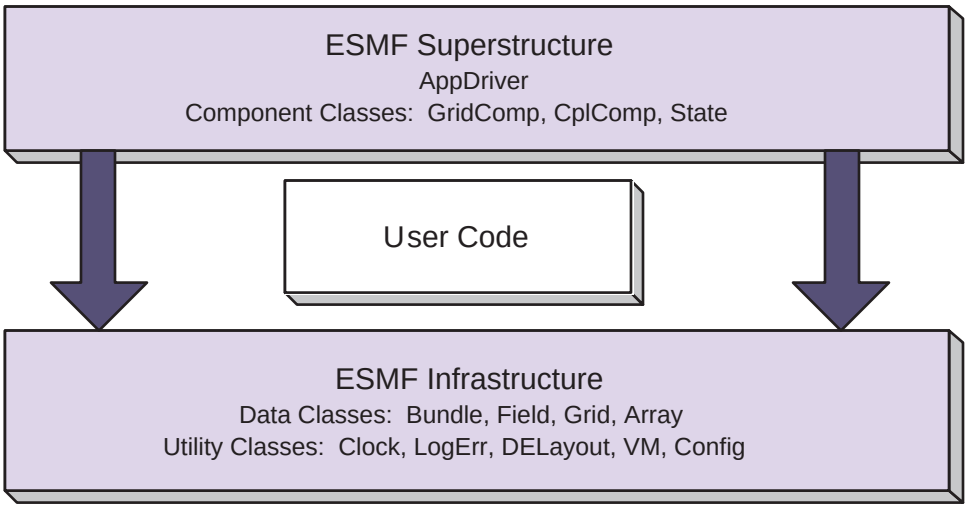
2 The ESMF Reference Manual for C

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 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.

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



3 How to Contact User Support and Find Additional Information

The ESMF team can answer questions about the interfaces presented in this document. For user support, please contact esmf_support@list.woc.noaa.gov.

More information on the ESMF project as a whole is available on the ESMF website, http://www.earthsystemmodeling.org. 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@list.woc.noaa.gov.

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

The 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 classes. The `ESMC_<Class>Create()` method allocates memory for the class structure itself and for internal variables, and initializes variables where appropriate. It is always written as a C function that returns a derived type instance of the class.
- `ESMC_<Class>Set()` and `ESMC_<Class>Get()`, for setting and retrieving a particular item or flag.
- `ESMC_<Class>Add()` and `ESMC_<Class>Remove()` for manipulating items that can be appended or inserted into a list of like items within a class. For example, the `ESMC_StateAdd()` method adds another Field to the list of Fields contained in the State class.
- `ESMC_<Class>Print()`, for printing the contents of a class 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` checks whether the Array and Grid associated with a Field are consistent.

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

```
#include "ESMC.h"

ESMC_Field field;

field = ESMC_FieldCreate("temp", &rc);
```

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 take significant time to set up and should not be created in a time-critical portion of code. Deep objects persist even after the method in which they were created has returned. Most classes in ESMF, including Fields, FieldBundles, Arrays, ArrayBundles, Grids, and Clocks, fall into this category.

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

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

See Section **??**, 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 Fortran subroutines within user code.

5.4 The ESMF Data Hierarchy

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

The main data classes 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 Fortran array.

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

- **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 an hierarchy of classes that represent different spaces associated with a computation. Each of these spaces can be manipulated, in order to give the user control over how a computation is executed. For Earth system models, this hierarchy starts with the address space associated with the computer and extends to the physical region described by the application. The main spatial classes in ESMF, from those closest to the machine to those closest to the application, are:

- The **Virtual Machine**, or **VM** The ESMF VM is an abstraction of a parallel computing environment that encompasses both shared and distributed memory, single and multi-core systems. Its primary purpose is resource allocation and management. Each Component runs in its own VM, using the resources it defines. The elements of a VM are **Persistent Execution Threads**, or **PETs**, that are executing in **Virtual Address Spaces**, or **VASs**. A simple case is one in which every PET is associated with a single MPI process. In this case every PET is executing in its own private VAS. If Components are nested, the parent Component allocates a subset of its PETs to its children. The children have some flexibility, subject to the constraints of the computing environment, to decide how they want to use the resources associated with the PETs they've received.

- **DELayout** A DELayout represents a data decomposition (we also refer to this as a distribution). Its basic elements are **Decomposition Elements**, or **DEs**. A DELayout associates a set of DEs with the PETs in a VM. DEs are not necessarily one-to-one with PETs. For cache blocking, or user-managed multi-threading, more DEs than PETs may be defined. Fewer DEs than PETs may also be defined if an application requires it.

  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 of a physical space. It associates a coordinate system, a set of coordinates, and a topology to a collection of grid cells. Grids in ESMF are comprised of DistGrids plus additional coordinate information.

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

- **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 Overall Rules and Behavior

6.1 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.

6.2 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_DATA_REF` flag, will not deallocate that buffer at the time the object is destroyed. The user must deallocate the buffer when all use of it is complete.

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

6.3 Equality and Copying Objects

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

7 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.

7.1 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.

Part II
Applications

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 applications that are of general interest to the community. These applications utilize the ESMF library to implement features such as printing general information about the ESMF installation, or generating regrid weight files. The provided ESMF applications are intended to be used as standard command line tools.

The bundled ESMF applications 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 the installation the applications 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 applications in more detail in section "Using Bundled ESMF Applications".

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

8 ESMF_Info

8.1 Description

The `ESMF_Info` application prints basic information about the ESMF installation to `stdout`.

The application usage is as follows:

```
ESMF_Info  [--help]

where
  --help     prints a brief usage message
```

9 ESMF_RegridWeightGen

9.1 Description

In addition to the online regridding functionality, the ESMF distribution also contains an exectuable for generating regridding weights. This tool reads in two grid files and outputs weights for interpolation between the two grids. The input and output files are all in netcdf format. The grid files are either in the same format as is used as an input to SCRIP [3], or in the ESMF unstructured grid format **??**. The weight file is the same format as is output by SCRIP. The interpolation weights can be generated with the bilinear, patch, or first order conservative methods decribed below. This application assumes that the source and destination grids are spherical and that the coordinates given in the files are latitude and longitude values. This file based regrid weight generation application is fully parallel. This application is used in the ESMF_RegridWeightGenCheck external demo, so that can serve as an example of its use.

Internally this application uses the ESMF public API to generate the interpolation weights. If a the source or destination grid is logically rectangular, then `ESMF_GridCreate()` **??** is used to create an ESMF_Grid object. The cell center coordinates of the input grid are put into the center stagger location (`ESMF_STAGGERLOC_CENTER`). In addition, for conservative regridding, the corner coordinates are also put into the corner stagger location (`ESMF_STAGGERLOC_CORNER`). The 2D coordinates are mapped into 3D Cartesian coordiantes by setting the `regridScheme` flag to `ESMF_REGRID_SCHEME_FULL3` while calling `ESMF_FieldRegridStore()`. The method `ESMF_MeshCreate()` **??** is used to create an ESMF_Mesh

object, if the source or destination grid is a cubed sphere grid or an unstructured grid. When making this call, the flag `convert3D` is set to `TRUE` to convert the 2D coordinates into 3D Cartesian coordinates. `ESMF_FieldRegridStore()` is used to generate the weight table and indicies table representing the interpolation matrix.

The regridding occurs in 3D to avoid problems with periodicity and with the pole singularity. This application supports four options for handling the pole region (i.e. the empty area above the top row of the source grid or below the bottom row of the source grid). The first option is to leave the pole region empty ("-p none"), in this case if a destination point lies above or below the top row of the source grid, it will fail to map, yielding an error. With the next two options, the pole region is handled by constructing an artificial pole in the center of the top and bottom row of grid points and then filling in the region from this pole to the edges of the source grid with triangles. The pole is located at the average of the position of the points surrounding it, but moved outward to be at the same radius as the rest of the points in the grid. The difference between these two artificial pole options is what value is used at the pole. The default pole option ("-p all") sets the value at the pole to be the average of the values of all of the grid points surrounding the pole. For the other option ("-p N"), the user chooses a number N from 1 to the number of source grid points around the pole. For each destination point, the value at the pole is then the average of the N source points surrounding that destination point. For the last pole option ("-p teeth") no artificial pole is constructed, instead the pole region is covered by connecting points across the top and bottom row of the source Grid into triangles. As this makes the top and bottom of the source sphere flat, for a big enough difference between the size of the source and destination pole regions, this can still result in unmapped destination points. Only pole option "none" is currently supported with the conservative interpolation method (i.e. "-m conserve").

This regridding application can be used to generate bilinear, patch, or first-order conservative interpolation weights. The default interpolation method is bilinear. The algorithm used by this application to generate the bilinear weights is the standard one found in many textbooks. Each destination point is mapped to a location in the source Mesh, the position of the destination point relative to the source points surrounding it is used to calculate the interpolation weights.

This application can also be used to generate patch interpolation weights. Patch interpolation is the ESMF version of a techique called "patch recovery" commonly used in finite element modeling [1] [2]. It typically results in better approximations to values and derivatives when compared to bilinear interpolation. Patch interpolation works by constructing multiple polynomial patches to represent the data in a source element. For 2D grids, these polynomials are currently 2nd degree 2D polynomials. The interpolated value at the destination point is the weighted average of the values of the patches at that point.

The patch interpolation process works as follows. For each source element containing a destination point we construct a patch for each corner node that makes up the element (e.g. 4 patches for quadrilateral elements, 3 for triangular elements). To construct a polynomial patch for a corner node we gather all the elements around that node. (Note that this means that the patch interpolation weights depends on the source element's nodes, and the nodes of all elements neighboring the source element.) We then use a least squares fitting algorithm to choose the set of coefficients for the polynomial that produces the best fit for the data in the elements. This polynomial will give a value at the destination point that fits the source data in the elements surrounding the corner node. We then repeat this process for each corner node of the source element generating a new polynomial for each set of elements. To calculate the value at the destination point we do a weighted average of the values of each of the corner polynomials evaluated at that point. The weight for a corner's polynomial is the bilinear weight of the destination point with regard to that corner. The patch method has a larger stencil than the bilinear, for this reason the patch weight matrix can be correspondingly larger than the bilinear matrix (e.g. for a quadrilateral grid the patch matrix is around 4x the size of the bilinear matrix). This can be an issue when performing a regrid weight generation operation close to the memory limit on a machine.

First-order conservative interpolation [4] is also available as a regridding method. This method will typically have a larger interpolation error than the previous two methods, but will do a much better job of preserving the value of the integral of data between the source and destination grid. In this method the value across each source cell is treated as a constant. The weights for a particular destination cell, are the area of intersection of each source cell with the destination cell divided by the area of the destination cell. Areas in this case are the great circle areas of the polygons which make up the cells (the cells around each center are defined by the corner coordinates in the grid file).

9.2 Usage

The command line arguments are all keyword based. Both the long keyward prefixed with '--' or the one character short keyword prefixed with '-' are supported. The format to run the application is as follows:

```
ESMF_RegridWeightGen  [--help]
                      [--source|-s] src_grid_filename
                      [--destination|-d] dst_grid_filename
                      [--weight|-w] out_weight_file
                      [--method|-m] [bilinear|patch|conserve]
                      [--pole|-p] [none|all|teeth|1|2|..]
                      --src_type [SCRIP|ESMF]
                      --dst_type [SCRIP|ESMF]
                      -t [SCRIP|ESMF]

where
  --help              - print the usage message
  --source or -s      - a required argument specifying the source grid file name

  --destination or -d - a required argument specifying the destination grid file name

  --weight or -w      - a required argument specifying the output regridding
                        weight file name

  --method or -m      - an optional argument specifying which interpolation method
                        is used.   The value can be one of the following:

                        bilinear     - for bilinear interpolation, also the default
                                       method if not specified.
                        patch        - for patch recovery interpolation
                        conserve     - for first-order conservative interpolation

  --pole or -p        - an optional argument indicating what to do with the pole.
                        The value can be one of the following:

                        none  - No pole, the source grid ends at the top (and bottom)
                                row of nodes specified in <source grid>.
                        all   - Construct an artificial pole placed in the center of the
                                top (or bottom) row of nodes, but projected onto the
                                sphere formed by the rest of the grid. The value at
                                this pole is the average of all the pole values. This
                                is the default option.

                        teeth - No new pole point is constructed, instead the holes at
                                the poles are filled by constructing triangles across
                                the top and bottom row of the source Grid. This can be
                                useful because no averaging occurs, however, because
                                the top and bottom of the sphere are now flat, for a
                                big enough mismatch between the size of the destination
                                and source pole regions, some destination points may
                                still not be able to be mapped to the source Grid.
```

<N> - Construct an artificial pole placed in the center of the top (or bottom) row of nodes, but projected onto the sphere formed by the rest of the grid. The value at this pole is the average of the N source nodes next to the pole and surrounding the destination point (i.e. the value may differ for each destination point. Here N ranges from 1 to the number of nodes around the pole.

--src_type - an optional argument specifying the source grid file type. The value could be either SCRIP or ESMF. Currently, the ESMF file type is only available for the unstructured grid. The default option is SCRIP.

--dst_type - an optional argument specifying the destination grid file type. The value could be either SCRIP or ESMF. Currently, the ESMF file type is only available for the unstructured grid. The default option is SCRIP.

-t - an optional argument specifying the file types for both the source and the destination grid files. The default option is SCRIP. If both -t and --src_type or --dst_type are given at the same time and they disagree with each other, an error message will be generated.

Part III
Superstructure

10 Overview of Superstructure

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

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

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

10.1 Superstructure Classes

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

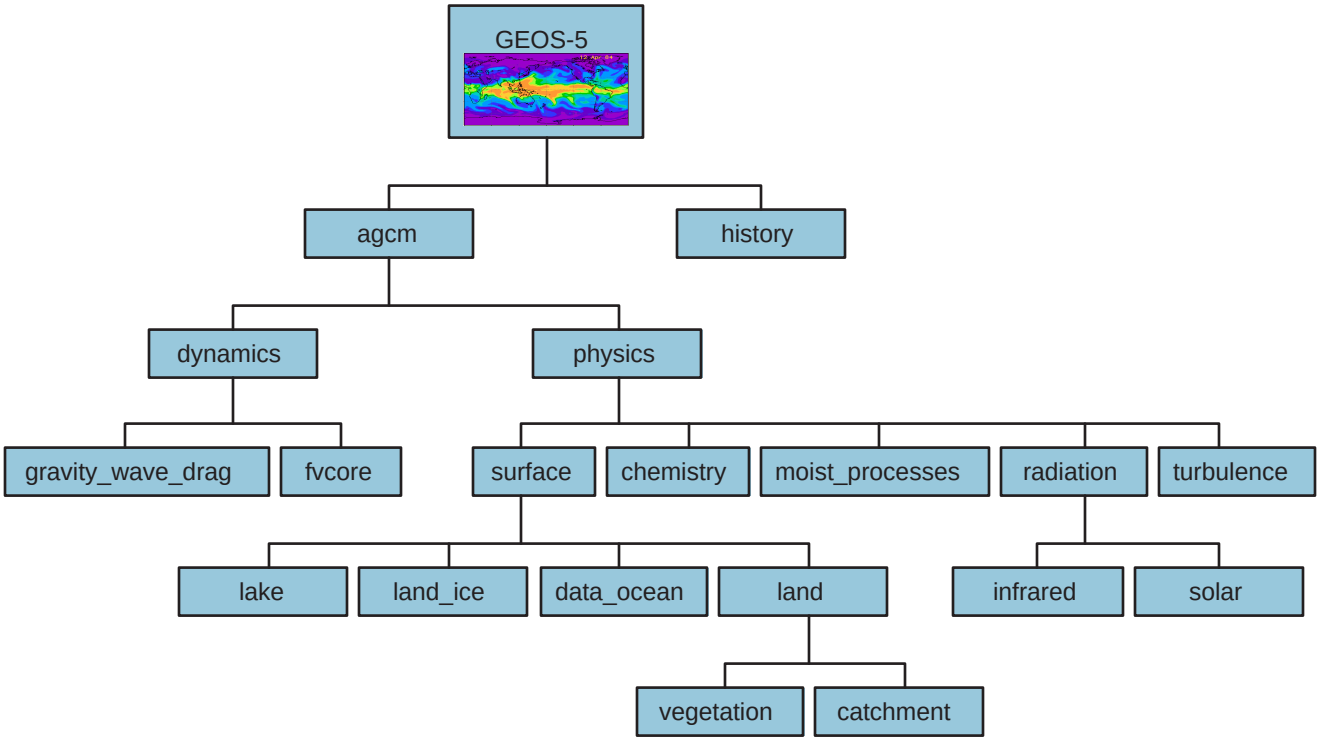
- **Component** An ESMF component has two parts, one that is supplied by the ESMF and one that is supplied by the user. The part that is supplied by the framework is an ESMF derived type that is either a Gridded Component (**GridComp**) or a Coupler Component (**CplComp**). A Gridded Component typically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

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

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

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

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.



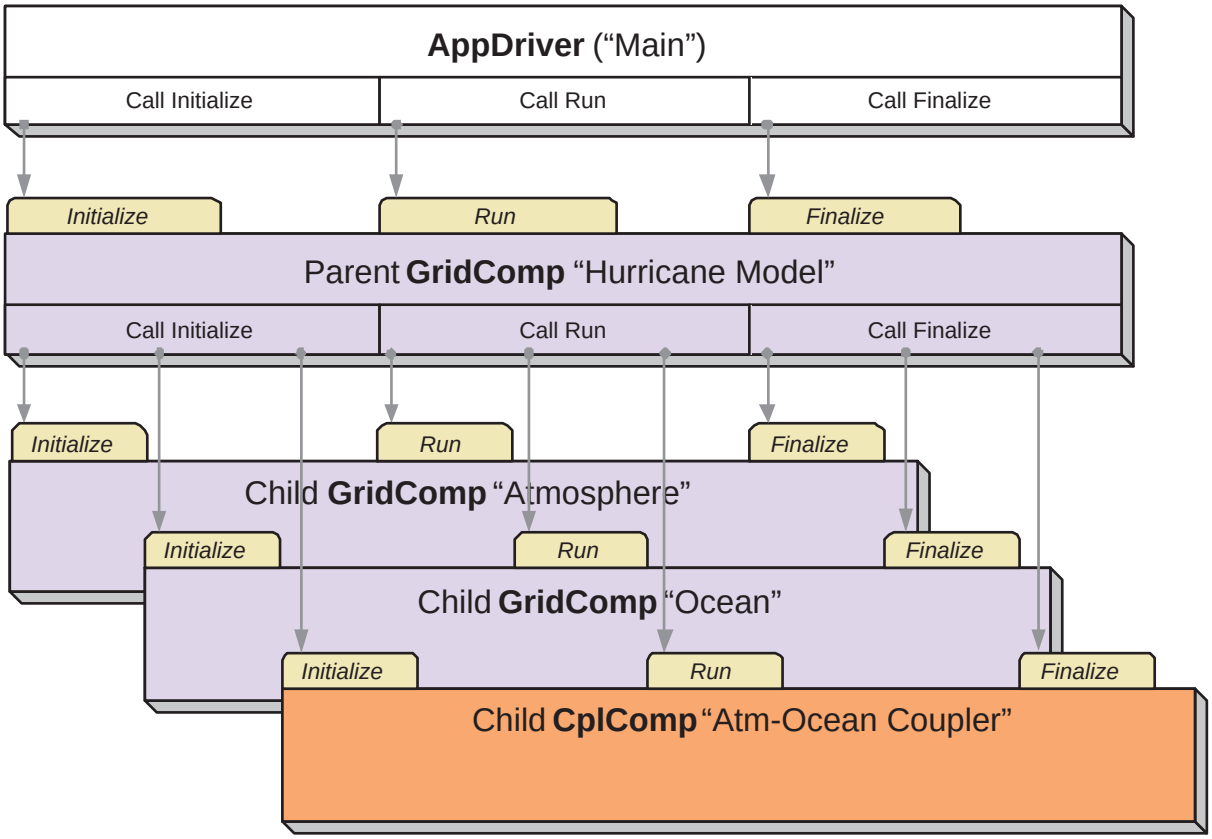
An ESMF coupled application typically involves an AppDriver, a parent Gridded Component, two or more child Gridded Components that require an inter-component data exchange, and one or more Coupler Components.

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

10.2 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.

10.3 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.

10.4 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.

10.5 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.

10.6 Performance

The ESMF design enables the user to configure ESMF applications so that data is transferred directly from one component to another, without requiring that it be copied or sent to a different data buffer as an interim step. This is likely to be the most efficient way of performing inter-component coupling. However, if desired, an application can also be configured so that data from a source component is sent to a distinct set of Coupler Component PETs for processing before being sent to its destination.

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

Figure 4: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running sequentially with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The AppDriver, Coupler, and all Gridded Components are distributed over nine PETs.

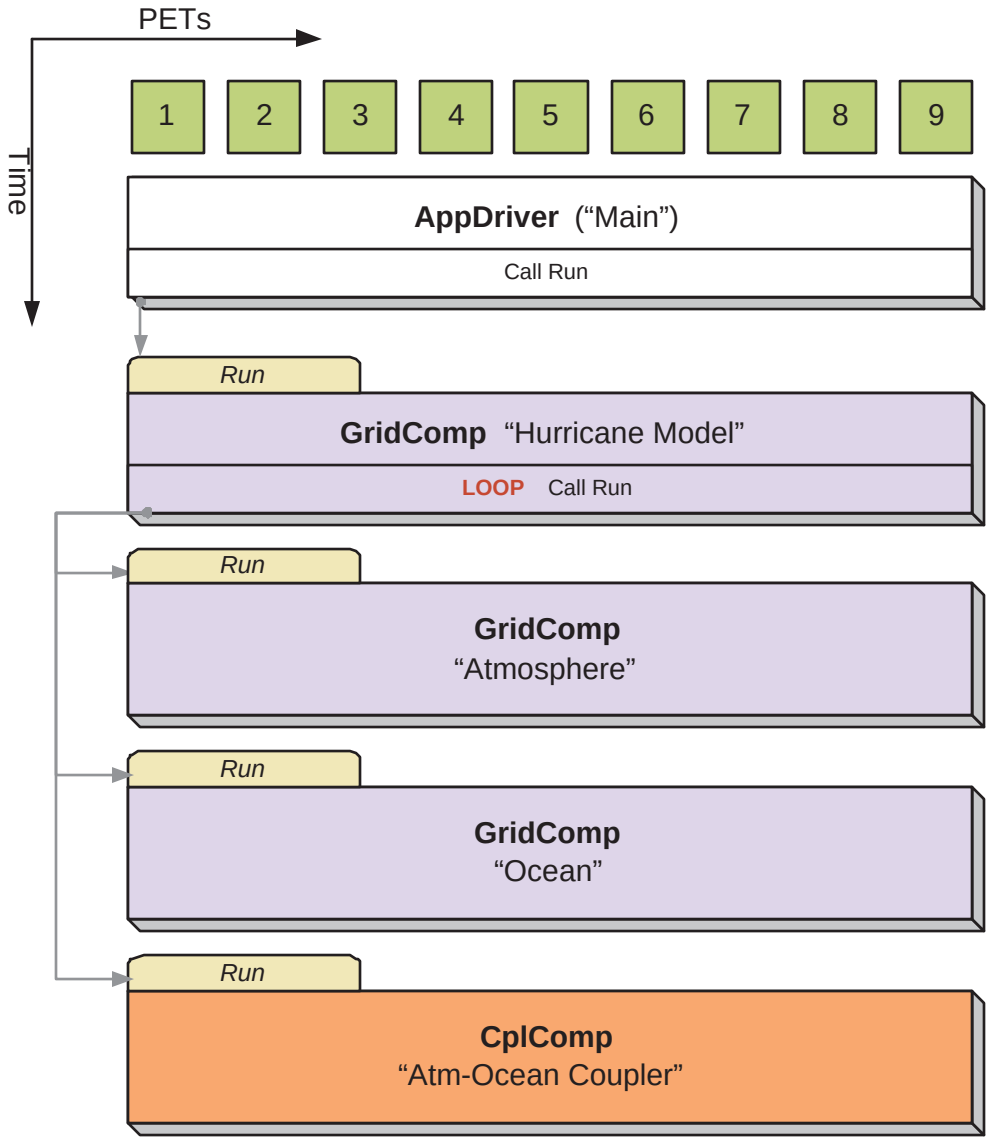


Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The AppDriver, Coupler, and top-level “Hurricane Model” Gridded Component are distributed over nine PETs. The “Atmosphere” Gridded Component is distributed over three PETs and the “Ocean” Gridded Component is distributed over six PETs.

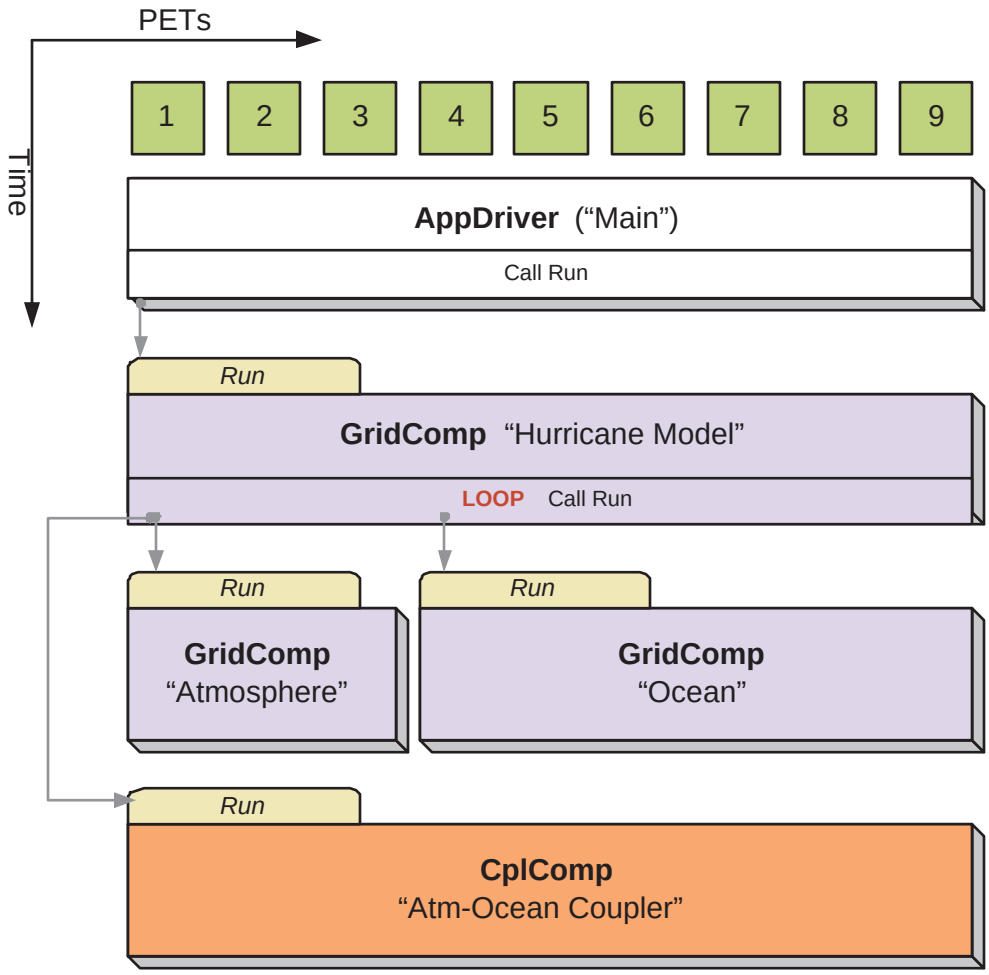
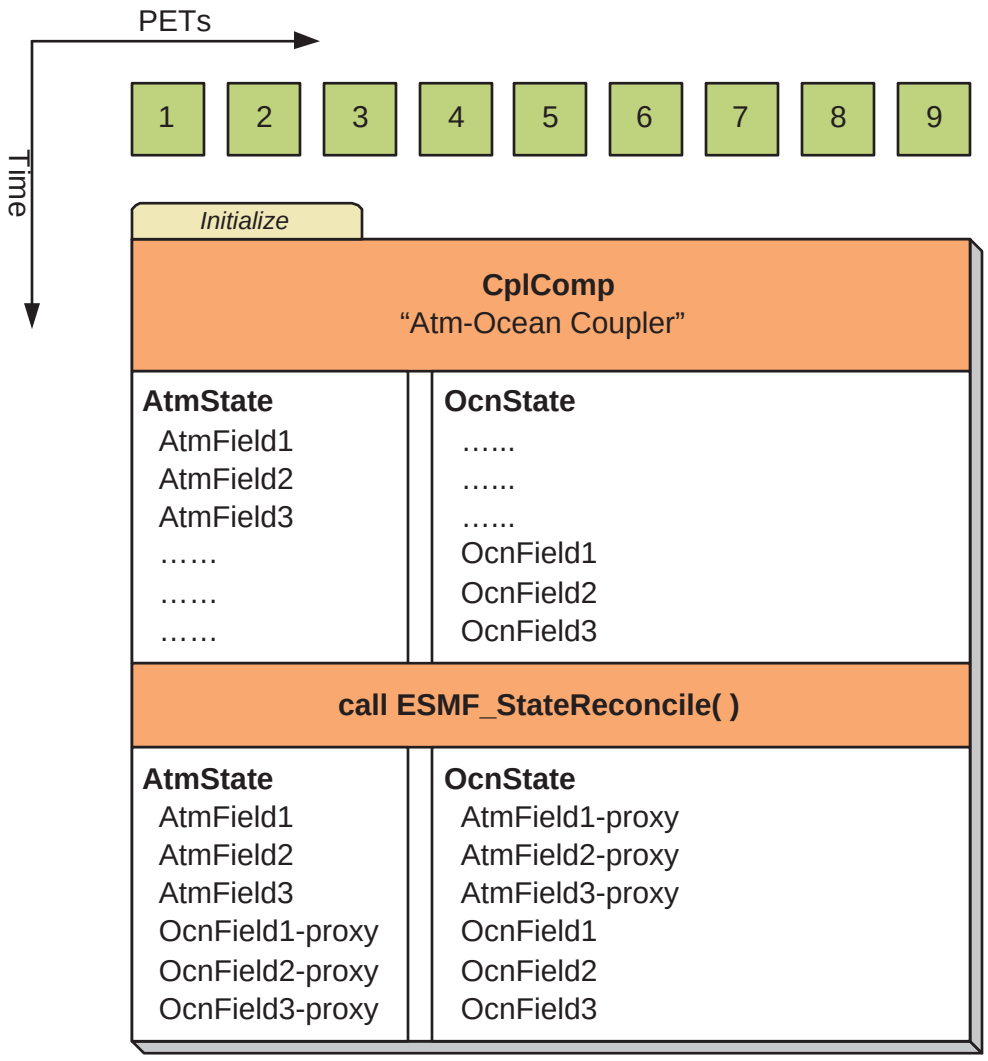
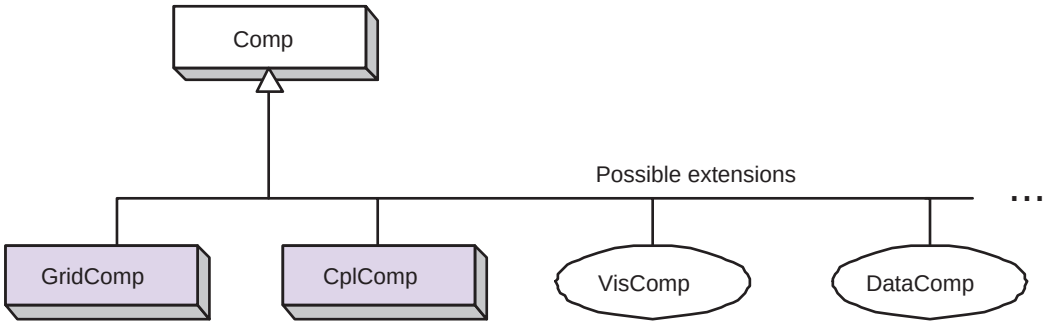


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



10.7 Object Model

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



11 Application Driver and Required ESMF Methods

11.1 Description

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

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

Sequential vs. Concurrent Execution In a sequential execution model every Component executes on all PETs, with each Component completing execution before the next Component begins. This has the appeal of simplicity of data consumption and production: when a Gridded Component starts all required data is available for use, and when a Gridded Component finishes all data produced is ready for consumption by the next Gridded Component. This approach also has the possibility of less data movement if the grid and data decomposition is done such that each processor's memory contains the data needed by the next Component.

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

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

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

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

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

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

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

The alternative is to start a different executable program on different PETs. This is a **MPMD** model, Multiple Program Multiple Data. There are complications with many job control systems on multiprocessor machines in getting the different executables started, and getting inter-process communcations 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.

11.2 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 library call `ESMC_<Grid/Cpl>CompSetVM()` can optionally be issues *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.

11.2.1 ESMC_Initialize - Initialize the ESMF Framework

INTERFACE:

```
   int ESMC_Initialize(
     int *rc,        // return code
     ...);           // optional arguments
#define ESMC_InitArgDefaultConfigFilename(ARG)  \
ESMCI_Arg(ESMCI_InitArgDefaultConfigFilenameID,ARG)
```

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 IO before the MPI environment is initialized. Please consult the documentation of your MPI implementation for details.
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.

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

11.2.2 ESMC_Finalize - Finanalize 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.

12 GridComp Class

12.1 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 **??**.
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.

12.2 Class API

12.2.1 ESMC_GridCompCreate - Create a Gridded Component

INTERFACE:

```
ESMC_GridComp ESMC_GridCompCreate(
  const char *name,                     // in
  enum ESMC_GridCompType mtype,         // 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.

mtype ESMC_GridComp model type, where models includes ESMF_ATM, ESMF_LAND, ESMF_OCEAN, ESMF_SEAICE, ESMF_RIVER, and ESMF_GRIDCOMPTYPE_UNKNOWN. Note that this has no meaning to the framework, it is an annotation for user code to query. See section **??** for a complete list of valid types.

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.

12.2.2 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.

12.2.3 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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_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.

12.2.5 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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_GridCompPrint - Print the contents of a GridComp

INTERFACE:

```
int ESMC_GridCompPrint(
  ESMC_GridComp comp,     // in
  const char *options     // 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.

[options] Print options are not yet supported, pass NULL.

12.2.7 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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_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_SETINIT, ESMF_SETRUN, ESMF_SETFINAL. 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_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.

12.2.10 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.

13 CplComp Class

13.1 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 12.1). 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.

13.2 Class API

13.2.1 ESMC_CplCompCreate - Create a Coupler Component

INTERFACE:

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

RETURN VALUE:

```
    Newly created ESMC_CplComp object.
```

DESCRIPTION:

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

name Name of the newly-created `ESMC_CplComp`.

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

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.

13.2.2 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.

13.2.3 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

13.2.4 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.

13.2.5 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

13.2.6 ESMC_CplCompPrint - Print a Coupler Component

INTERFACE:

```
int ESMC_CplCompPrint(
  ESMC_CplComp comp,      // in
  const char *options     // 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.

[options] Print options are not yet supported, pass NULL.

13.2.7 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

13.2.8 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_SETINIT`, `ESMF_SETRUN`, `ESMF_SETFINAL`. 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.

13.2.9 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.

13.2.10 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.

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 using during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time.

   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 identifiaction 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 are useful because they provide a standard, convenient way for developers to collect together information related to model or observational data. The information assembled in a data class includes a data pointer, a set of attributes (e.g. units, although attributes can also be user-defined), and a description of an associated grid. The same set of information within an ESMF data object can be used by the framework to arrange intercomponent data transfers, to perform I/O, for communications such as gathers and scatters, for simplification of interfaces within user code, for debugging, and for other functions. This unifies and organizes codes overall so that the user need not define different representations of metadata for the same field for I/O and for component coupling.

Since it is critical that users be able to introduce ESMF into their codes easily and incrementally, ESMF data classes can be created based on native Fortran pointers. Likewise, there are methods for retrieving native Fortran pointers from within ESMF data objects. This allows the user to perform allocations using ESMF, and to retrieve Fortran arrays later for optimized model calculations. The ESMF data classes do not have associated differential operators or other mathematical methods.

For flexibility, it is not necessary to build an ESMF data object all at once. For example, it's possible to create a field but to defer allocation of the associated field data until a later time.

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

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.
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.

16.1.1 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 depends 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.

16.2 Class API

16.2.1 ESMC_FieldCreate - Create a Field

INTERFACE:

```
ESMC_Field ESMC_FieldCreate(
  ESMC_Mesh mesh,                          // in
  ESMC_ArraySpec arrayspec,                // in
  ESMC_InterfaceInt gridToFieldMap,        // in
  ESMC_InterfaceInt ungriddedLBound,       // in
  ESMC_InterfaceInt 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.

16.2.2 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`.

16.2.3 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.

16.2.4 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.

16.2.5 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.

16.2.6 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`. Return code; equals `ESMF_SUCCESS` if there are no errors.
The arguments are:

field Print contents of this `ESMC_Field`.

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,         // inout
   int *rank,                        // in
   enum ESMC_TypeKind *typekind      // in
 );
```

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 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 Mesh Class

19.1 Description

Unstructured grids are commonly used in the computational solution of Partial Differential equations. These are especially useful for problems that involve complex geometry, where using the less flexible structured grids can result in grid representation of regions where no computation is needed. Finite element and finite volume methods map naturally to unstructured grids and are used commonly in hydrology, ocean modeling, and many other applications.
In order to provide support for application codes using unstructured grids, the ESMF library provides a class for representing unstructured grids called the **Mesh**. Fields can be created on a Mesh to hold data. In Fortran, Fields created on a Mesh can also be used as either the source or destination or both of an interpolaton (i.e. an `ESMF_FieldRegridStore()` call). This capability is currently not supported with the C interface, however, if the C Field is passed via a State to a component written in Fortran then the regridding can be performed there. The rest of this section describes the Mesh class and how to create and use them in ESMF.

19.1.1 Mesh Representation in ESMF

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

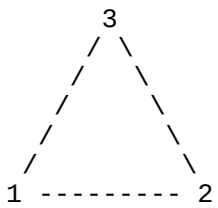
19.1.2 Supported Meshes

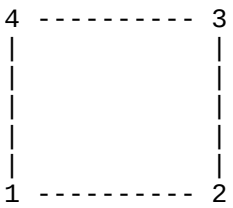
The range of Meshes supported by ESMF are defined by several factors: dimension, element types, and distribution.
ESMF currently only supports Meshes whose number of coordinate dimensions (spatial dimension) is 2 or 3. The dimension of the elements in a Mesh (parametric dimension) must be less than or equal to the spatial dimension, but also must be either 2 or 3. This means that an ESMF mesh may be either 2D elements in 2D space, 3D elements in 3D space, or a manifold constructed of 2D elements embedded in 3D space.
ESMF currently supports two types of elements for each Mesh parametric dimension. For a parametric dimension of 2 the supported element types are triangles or quadralaterals. For a parametric dimension of 3 the supported element types are tetrahedrons and hexahedrons. See Section 19.2.1 for diagrams of these. The Mesh supports any combination of element types within a particular dimension, but types from different dimensions may not be mixed, for example, a Mesh cannot be constructed of both quadralaterals and tetrahedra.
ESMF currently only supports distributions where every node on a PET must be a part of an element on that PET. In other words, there must not be nodes without an element on a PET.

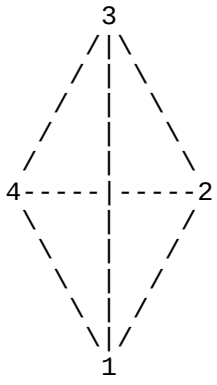
19.2 Mesh Options

19.2.1 ESMC_MeshElemType

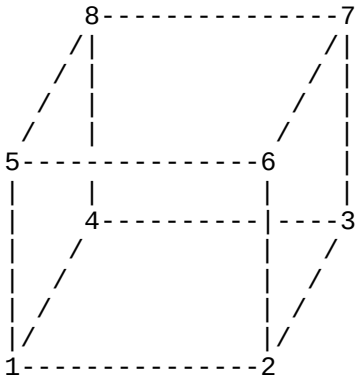
DESCRIPTION:
An ESMF Mesh can be constructed from a combination of different elements. The type of elements that can be used in a Mesh depends on the Mesh's parameteric dimension, which is set during Mesh creation. The following are the valid Mesh element types for each valid Mesh parametric dimension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







ESMC_MESHELEMTYPE_TRI



ESMC_MESHELEMTYPE_QUAD

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

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

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

ESMC_MESHELEMTYPE_TETRA

ESMC_MESHELEMTYPE_HEX

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

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

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

19.3 Class API

19.3.1 ESMC_MeshAddElements - Add elements to a Mesh

INTERFACE:

```
int ESMC_MeshAddElements(
  ESMC_Mesh mesh,           // inout
  int  elementCount,        // in
  int *elementIds,          // in
  int *elementTypes,        // in
  int *elementConn          // in
);
```

RETURN VALUE:

Return code; equals ESMF_SUCCESS if there are no errors.

DESCRIPTION:

This call is the third and last part of the three part mesh create sequence and should be called after the mesh is created with `ESMF_MeshCreate()` (19.3.3) and after the nodes are added with `ESMF_MeshAddNodes()` (19.3.2). This call adds the elements to the mesh and finalizes the create. After this call the Mesh is usable, for example a Field may be built on the created Mesh object and this Field may be used in a `ESMF_FieldRegridStore()` call.
The parameters to this call `elementIds`, `elementTypes`, and `elementConn` describe the elements to be created. The description for a particular element lies at the same index location in `elementIds` and `elementTypes`. Each entry in `elementConn` consists of the list of nodes used to create that element, so the connections for element e in the `elementIds` array will start at $number\_of\_nodes\_in\_element(1) + number\_of\_nodes\_in\_element(2) + \cdots + number\_of\_nodes\_in\_element(e-1) + 1$ in `elementConn`.

mesh Mesh object.

elementCount The number of elements on this PET.

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

elementTypes An array containing the types of the elements to be created on this PET. The types used must be appropriate for the parametric dimension of the Mesh. Please see Section 19.2.1 for the list of options. This input consists of a 1D array of size `elementCount`.

elementConn An array containing the indexes of the sets of nodes to be connected together to form the elements to be created on this PET. The entries in this list are NOT node global ids, but rather each entry is a local index (1 based) into the list of nodes which were created on this PET by the previous `ESMC_MeshAddNodes()` call. In other words, an entry of 1 indicates that this element contains the node described by `nodeIds(1)`, `nodeCoords(1)`, etc. passed into the `ESMC_MeshAddNodes()` call on this PET. It is also important to note that the order of the nodes in an element connectivity list matters. Please see Section 19.2.1 for diagrams illustrating the correct order of nodes in a element. This input consists of a 1D array with a total size equal to the sum of the number of nodes in each element on this PET. The number of nodes in each element is implied by its element type in `elementTypes`. The nodes for each element are in sequence in this array (e.g. the nodes for element 1 are elementConn(1), elementConn(2), etc.).

19.3.2 ESMC_MeshAddNodes - Add nodes to a Mesh

INTERFACE:

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

RETURN VALUE:

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

DESCRIPTION:

This call is the second part of the three part mesh create sequence and should be called after the mesh's dimensions are set using `ESMC_MeshCreate()`. This call adds the nodes to the mesh. The next step is to call `ESMC_MeshAddElements()` (19.3.3).

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

mesh Mesh object.

nodeCount The number of nodes on this PET.

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

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

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

19.3.3 ESMC_MeshCreate - Create a Mesh as a 3 step process

INTERFACE:

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

RETURN VALUE:

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

DESCRIPTION:

This call is the first part of the three part mesh create sequence. This call sets the dimension of the elements in the mesh (`parametricDim`) and the number of coordinate dimensions in the mesh (`spatialDim`). The next step is to call `ESMC_MeshAddNodes()` (19.3.2) to add the nodes and then `ESMC_MeshAddElements()` (19.3.1) to add the elements and finalize the mesh.
The arguments are:

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

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

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

19.3.4 ESMC_MeshDestroy - Destroy a Mesh

INTERFACE:

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

RETURN VALUE:

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

DESCRIPTION:

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

mesh Mesh object whose memory is to be freed.

19.3.5 ESMC_MeshFreeMemory - Remove a Mesh and its memory

INTERFACE:

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

RETURN VALUE:

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

DESCRIPTION:

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

mesh Mesh object whose memory is to be freed.

19.3.6 ESMC_MeshGetLocalElementCount - Get the number of elements in a Mesh owned by the current PET

INTERFACE:

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

RETURN VALUE:

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

DESCRIPTION:

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

mesh The mesh

elementCount The number of elements on this PET.

19.3.7 ESMC_MeshGetLocalNodeCount - Get the number of nodes in a Mesh owned by the current PET

INTERFACE:

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

RETURN VALUE:

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

DESCRIPTION:

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

mesh The mesh

nodeCount The number of nodes on this PET.

20 DistGrid Class

20.1 Description

The ESMF DistGrid class sits on top of the DELayout class (not currently directly accessible through the ESMF C API) and holds domain information in index space. A DistGrid object captures the index space topology and describes its decomposition in terms of DEs. Combined with DELayout and VM the DistGrid defines the data distribution of a domain decomposition across the computational resources of an ESMF Component.
The global domain is defined as the union or "patchwork" of logically rectangular (LR) sub-domains or *patches*. The DistGrid create methods allow the specification of such a patchwork global domain and its decomposition into exclusive, DE-local LR regions according to various degrees of user specified constraints. Complex index space topologies can be constructed by specifying connection relationships between patches during creation.
The DistGrid class holds domain information for all DEs. Each DE is associated with a local LR region. No overlap of the regions is allowed. The DistGrid offers query methods that allow DE-local topology information to be extracted, e.g. for the construction of halos by higher classes.
A DistGrid object only contains decomposable dimensions. The minimum rank for a DistGrid object is 1. A maximum rank does not exist for DistGrid objects, however, ranks greater than 7 may lead to difficulties with respect to the Fortran API of higher classes based on DistGrid. The rank of a DELayout object contained within a DistGrid object must be equal to the DistGrid rank. Higher class objects that use the DistGrid, such as an Array object, may be of different rank than the associated DistGrid object. The higher class object will hold the mapping information between its dimensions and the DistGrid dimensions.

20.2 Class API

20.2.1 ESMC_DistGridCreate - Create a DistGrid

INTERFACE:

```
 ESMC_DistGrid ESMC_DistGridCreate(
   ESMC_InterfaceInt minIndexInterfaceArg,   // in
   ESMC_InterfaceInt maxIndexInterfaceArg,   // in
   int *rc                                   // out
 );
```

RETURN VALUE:

```
    Newly created ESMC_DistGrid object.
```

DESCRIPTION:

Create an `ESMC_DistGrid` from a single logically rectangular (LR) patch with default decomposition. The default decomposition is `deCount` $\times 1 \times ... \times 1$, where `deCount` is the number of DEs in a default DELayout, equal to `petCount`. This means that the default decomposition will be into as many DEs as there are PETs, with 1 DE per PET.
The arguments are:

minIndex Global coordinate tuple of the lower corner of the patch.

maxIndex Global coordinate tuple of the upper corner of the patch.

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

20.2.2 ESMC_DistGridDestroy - Destroy a DistGrid

INTERFACE:

```
 int ESMC_DistGridDestroy(
   ESMC_DistGrid *distgrid         // inout
 );
```

RETURN VALUE:

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

DESCRIPTION:

Destroy an `ESMC_DistGrid` object.
The arguments are:

distgrid `ESMC_DistGrid` object to be destroyed.

20.2.3 ESMC_DistGridPrint - Print a DistGrid

INTERFACE:

```
int ESMC_DistGridPrint(
  ESMC_DistGrid distgrid          // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Print internal information of the specified `ESMC_DistGrid` object.
The arguments are:

distgrid `ESMC_DistGrid` object to be destroyed.

Part V

Infrastructure: Utilities

21 Overview of Infrastructure Utility Classes

The ESMF utilities are a set of tools for quickly assembling modeling applications.
The Time Management Library provides utilities for time and time interval representation, as well as a higher-level utility, a clock, that controls model time stepping.
The ESMF Config class provides configuration management based on NASA DAO's Inpak package, a collection of methods for accessing files containing input parameters stored in an ASCII format.
The ESMF LogErr class consists of a method for writing error, warning, and informational messages to a default Log file that is created during ESMF initialization.
The ESMF VM (Virtual Machine) class provides methods for querying information about a VM. A VM is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component. In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods.

22 Time Manager Utility

The ESMF Time Manager utility includes software for time and time interval representation, as well as model time advancement. Since multi-component geophysical applications often require synchronization across the time management schemes of the individual components, the Time Manager's standard calendars and consistent time representation promote component interoperability.

Key Features
Drift-free timekeeping through an integer-based internal time representation. Both integers and reals can be specified at the interface.
Support for many calendar types.
Support for both concurrent and sequential modes of component execution.

22.1 Time Manager Classes

There are four ESMF classes that represent time concepts:

- **Calendar** A Calendar can be used to keep track of the date as an ESMF Gridded Component advances in time. Standard calendars (such as Gregorian and 360-day) are supported.
- **Time** A Time represents a time instant in a particular calendar, such as November 28, 1964, at 7:00pm EST in the Gregorian calendar. The Time class can be used to represent the start and stop time of a time integration.
- **TimeInterval** TimeIntervals represent a period of time, such as 3 hours. Time steps can be represented using TimeIntervals.
- **Clock** Clocks collect the parameters and methods used for model time advancement into a convenient package. A Clock can be queried for quantities such as current simulation time and time step. Clock methods include incrementing the current time, and printing the its contents.

22.2 Calendar

The set of supported calendars includes:

Gregorian The standard Gregorian calendar.

no-leap The Gregorian calendar with no leap years.

Julian The standard Julian date calendar.

Julian Day The standard Julian days calendar.

Modified Julian Day The Modified Julian days calendar.

360-day A 30-day-per-month, 12-month-per-year calendar.

no calendar Tracks only elapsed model time in hours, minutes, seconds.

See Section 23.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

22.3 Time Instants and TimeIntervals

TimeIntervals and Time instants (simply called Times) are the computational building blocks of the Time Manager utility. Times support different queries for values of individual Time components such as year and hour. See Sections 24.1 and 25.1, respectively, for use of Times and TimeIntervals.

22.4 Clocks

It is useful to identify a higher-level concept to repeatedly step a Time forward by a TimeInterval. We refer to this capability as a Clock, and include in its required features the ability to store the start and stop times of a model run, and to query the value of quantities such as the current time and the number of time steps taken. Applications may contain temporary or multiple Clocks. Section 26.1 describes the use of Clocks in detail.

23 Calendar Class

23.1 Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, Julian Day, Modified Julian Day, no-leap, 360-day, and no-calendar. Brief descriptions are provided for each calendar below.

23.2 Calendar Options

23.2.1 ESMC_CalendarType

DESCRIPTION:
Supported calendar types.
Valid values are:

ESMC_CAL_360DAY *Valid range: machine limits*
In the 360-day calendar, there are 12 months, each of which has 30 days. Like the no-leap calendar, this is a simple approximation to the Gregorian calendar sometimes used by modelers.

ESMC_CAL_GREGORIAN *Valid range: 3/1/4801 BC to 10/29/292,277,019,914*
The Gregorian calendar is the calendar currently in use throughout Western countries. Named after Pope Gregory XIII, it is a minor correction to the older Julian calendar. In the Gregorian calendar every fourth year is a leap year in which February has 29 and not 28 days; however, years divisible by 100 are not leap years unless they are also divisible by 400. As in the Julian calendar, days begin at midnight.

ESMC_CAL_JULIAN *Valid range: 3/1/4713 BC to 4/24/292,271,018,333*
The Julian calendar was introduced by Julius Caesar in 46 B.C., and reached its final form in 4 A.D. The Julian calendar differs from the Gregorian only in the determination of leap years, lacking the correction for years divisible by 100 and 400 in the Gregorian calendar. In the Julian calendar, any year is a leap year if divisible by 4. Days are considered to begin at midnight.

ESMC_CAL_JULIANDAY *Valid range: +/- $1x10^{14}$*
Julian days simply enumerate the days and fraction of a day which have elapsed since the start of the Julian era, defined as beginning at noon on Monday, 1st January of year 4713 B.C. in the Julian calendar. Julian days, unlike the dates in the Julian and Gregorian calendars, begin at noon.

ESMC_CAL_MODJULIANDAY *Valid range: +/- $1x10^{14}$*
The Modified Julian Day (MJD) was introduced by space scientists in the late 1950's. It is defined as an offset from the Julian Day (JD):

MJD = JD - 2400000.5

The half day is subtracted so that the day starts at midnight.

ESMC_CAL_NOCALENDAR *Valid range: machine limits*
The no-calendar option simply tracks the elapsed model time in seconds.

ESMC_CAL_NOLEAP *Valid range: machine limits*
The no-leap calendar is the Gregorian calendar with no leap years - February is always assumed to have 28 days. Modelers sometimes use this calendar as a simple, close approximation to the Gregorian calendar.

23.3 Class API

23.3.1 ESMC_CalendarCreate - Create a Calendar

INTERFACE:

```
ESMC_Calendar ESMC_CalendarCreate(
  const char *name,                       // in
  enum ESMC_CalendarType calendartype,    // in
  int *rc                                 // out
);
```

RETURN VALUE:

```
Newly created ESMC_Calendar object.
```

DESCRIPTION:

Creates and sets a `ESMC_Calendar` object to the given built-in `ESMC_CalendarType`.
The arguments are:

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

calendartype The built-in `ESMC_CalendarType`. Valid values are: `ESMC_CAL_360DAY`, `ESMC_CAL_GREGORIAN`, `ESMC_CAL_JULIAN`, `ESMC_CAL_JULIANDAY`, `ESMC_CAL_MODJULIANDAY`, `ESMC_CAL_NOCALENDAR`, and `ESMC_CAL_NOLEAP`. See Section 23.2 for a description of each calendar type.

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

23.3.2 ESMC_CalendarDestroy - Destroy a Calendar

INTERFACE:

```
int ESMC_CalendarDestroy(
  ESMC_Calendar *calendar   // inout
);
```

RETURN VALUE:

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

DESCRIPTION:

Releases all resources associated with this `ESMC_Calendar`.
The arguments are:

calendar Destroy contents of this `ESMC_Calendar`.

23.3.3 ESMC_CalendarPrint - Print a Calendar

INTERFACE:

```
int ESMC_CalendarPrint(
  ESMC_Calendar calendar   // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Prints out an `ESMC_Calendar`'s properties to `stdio`, in support of testing and debugging. The arguments are:

calendar `ESMC_Calendar` object to be printed.

24 Time Class

24.1 Description

A Time represents a specific point in time.
There are Time methods defined for setting and getting a Time.
A Time that is specified in hours does not need to be associated with a standard calendar; use ESMC_CAL_NOCALENDAR.
A Time whose specification includes time units of a year must be associated with a standard calendar. The ESMF representation of a calendar, the Calendar class, is described in Section 23.1. The `ESMC_TimeSet` method is used to initialize a Time as well as associate it with a Calendar. If a Time method is invoked in which a Calendar is necessary and one has not been set, the ESMF method will return an error condition.
In the ESMF the TimeInterval class is used to represent time periods. This class is frequently used in combination with the Time class. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval.

24.2 Class API

24.2.1 ESMC_TimeGet - Get a Time value

INTERFACE:

```
int ESMC_TimeGet(
  ESMC_Time time,                          // in
  ESMC_I4 *yy,                             // out
  ESMC_I4 *h,                              // out
  ESMC_Calendar *calendar,                 // out
  enum ESMC_CalendarType *calendartype,    // out
  int *timeZone                            // out
);
```

RETURN VALUE:

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

DESCRIPTION:

Gets the value of an `ESMC_Time` in units specified by the user.
The arguments are:

time `ESMC_Time` object to be queried.

[yy] Integer year (>= 32-bit).

[h] Integer hours.

[calendar] Associated `ESMC_Calendar`.

[calendarType] Associated `ESMC_CalendarType`.

24.2.2 ESMC_TimePrint - Print a Time

INTERFACE:

```
int ESMC_TimePrint(
  ESMC_Time time   // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Prints out an `ESMC_Time`'s properties to `stdio`, in support of testing and debugging.
The arguments are:

time `ESMC_Time` object to be printed.

24.2.3 ESMC_TimeSet - Initialize or set a Time

INTERFACE:

```
int ESMC_TimeSet(
  ESMC_Time *time,                        // inout
  ESMC_I4 yy,                             // in
  ESMC_I4 h,                              // in
  ESMC_Calendar calendar,                 // in
  enum ESMC_CalendarType calendartype,    // in
  int timeZone                            // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Initializes an `ESMC_Time` with a set of user-specified units.
The arguments are:

time `ESMC_Time` object to initialize or set.

yy Integer year (>= 32-bit).

h Integer hours.

calendar Associated `ESMC_Calendar`. If not created, defaults to calendar `ESMC_CAL_NOCALENDAR` or default specified in `ESMC_Initialize()`. If created, has precedence over calendarType below.

calendarType Specifies associated `ESMC_Calendar` if calendar argument above not created. More convenient way of specifying a built-in calendar type.

25 TimeInterval Class

25.1 Description

A TimeInterval represents a period between time instants. It can be either positive or negative.
There are TimeInterval methods defined for setting and getting a TimeInterval, for printing the contents of a TimeInterval.
The class used to represent time instants in ESMF is Time, and this class is frequently used in operations along with TimeIntervals. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval. TimeIntervals are used by other parts of the ESMF timekeeping system, such as Clocks; see Section 26.1.

25.2 Class API

25.2.1 ESMC_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
int ESMC_TimeIntervalGet(
  ESMC_TimeInterval timeinterval,   // in
  ESMC_I8 *s_i8,                    // out
  ESMC_R8 *h_r8                     // out
);
```

RETURN VALUE:

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

DESCRIPTION:

Gets the value of an `ESMC_TimeInteval` in units specified by the user.
The arguments are:

timeinterval `ESMC_TimeInterval` object to be queried.

[s_i8] Integer seconds (large, >= 64-bit).

[h_r8] Double precision hours.

25.2.2 ESMC_TimeIntervalPrint - Print a TimeInterval

INTERFACE:

```
int ESMC_TimeIntervalPrint(
  ESMC_TimeInterval timeinterval   // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Prints out an `ESMC_TimeInterval`'s properties to `stdio`, in support of testing and debugging.
The arguments are:

timeinterval `ESMC_TimeInterval` object to be printed.

25.2.3 ESMC_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
int ESMC_TimeIntervalSet(
  ESMC_TimeInterval *timeinterval,   // inout
  ESMC_I4 h                          // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Sets the value of the `ESMC_TimeInterval` in units specified by the user.
The arguments are:

timeinterval `ESMC_TimeInterval` object to initialize or set.

h Integer hours.

26 Clock Class

26.1 Description

The Clock class advances model time and tracks its associated date on a specified Calendar. It stores start time, stop time, current time, and a time step.
There are methods for setting and getting the Times associated with a Clock. Methods are defined for advancing the Clock's current time and printing a Clock's contents.

26.2 Class API

26.2.1 ESMC_ClockAdvance - Advance a Clock's current time by one time step

INTERFACE:

```
int ESMC_ClockAdvance(
  ESMC_Clock clock   // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Advances the `ESMC_Clock`'s current time by one time step.
The arguments are:

clock `ESMC_Clock` object to be advanced.

26.2.2 ESMC_ClockCreate - Create a Clock

INTERFACE:

```
ESMC_Clock ESMC_ClockCreate(
  const char *name,             // in
  ESMC_TimeInterval timeStep,   // in
  ESMC_Time startTime,          // in
  ESMC_Time stopTime,           // in
  int *rc                       // out
);
```

RETURN VALUE:

```
    Newly created ESMC_Clock object.
```

DESCRIPTION:

Creates and sets the initial values in a new `ESMC_Clock` object.
The arguments are:

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

timeStep The ESMC_Clock's time step interval, which can be positive or negative.

startTime The ESMC_Clock's starting time. Can be less than or or greater than stopTime, depending on a positive or negative timeStep, respectively, and whether a stopTime is specified; see below.

stopTime The ESMC_Clock's stopping time. Can be greater than or less than the startTime, depending on a positive or negative timeStep, respectively.

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

26.2.3 ESMC_ClockDestroy - Destroy a Clock

INTERFACE:

```
int ESMC_ClockDestroy(
  ESMC_Clock *clock   // inout
);
```

RETURN VALUE:

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

DESCRIPTION:

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

clock Destroy contents of this ESMC_Clock.

26.2.4 ESMC_ClockGet - Get a Clock's properties

INTERFACE:

```
int ESMC_ClockGet(
  ESMC_Clock clock,                 // in
  ESMC_TimeInterval *currSimTime,   // out
  ESMC_I8 *advanceCount             // out
);
```

RETURN VALUE:

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

DESCRIPTION:

Gets one or more of the properties of an ESMC_Clock.
The arguments are:

clock ESMC_Clock object to be queried.

[currSimTime] The current simulation time.

[advanceCount] The number of times the ESMC_Clock has been advanced.

26.2.5 ESMC_ClockPrint - Print the contents of a Clock

INTERFACE:

```
int ESMC_ClockPrint(
  ESMC_Clock clock   // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Prints out an `ESMC_Clock`'s properties to `stdio`, in support of testing and debugging.
The arguments are:

clock `ESMC_Clock` object to be printed.

27 Config Class

27.1 Description

ESMF Configuration Management is based on NASA DAO's Inpak 90 package, a Fortran 90 collection of routines/functions for accessing *Resource Files* in ASCII format.The package is optimized for minimizing formatted I/O, performing all of its string operations in memory using Fortran intrinsic functions.

27.1.1 Package history

The ESMF Configuration Management Package was evolved by Leonid Zaslavsky and Arlindo da Silva from Ipack90 package created by Arlindo da Silva at NASA DAO.
Back in the 70's Eli Isaacson wrote IOPACK in Fortran 66. In June of 1987 Arlindo da Silva wrote Inpak77 using Fortran 77 string functions; Inpak 77 is a vastly simplified IOPACK, but has its own goodies not found in IOPACK. Inpak 90 removes some obsolete functionality in Inpak77, and parses the whole resource file in memory for performance.

27.2 Class API

27.2.1 ESMC_ConfigCreate - Create a Config object

INTERFACE:

```
ESMC_Config ESMC_ConfigCreate(
  int* rc                    // out
);
```

RETURN VALUE:

```
    ESMC_Config*  to newly allocated ESMC_Config
```

DESCRIPTION:

Creates an `ESMC_Config` for use in subsequent calls.
The arguments are:

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

27.2.2 ESMC_ConfigDestroy - Destroy a Config object

INTERFACE:

```
int ESMC_ConfigDestroy(
  ESMC_Config* config        // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Destroys the `config` object.
The arguments are:

config Already created `ESMC_Config` object to destroy.

27.2.3 ESMC_ConfigFindLabel - Find a label

INTERFACE:

```
int ESMC_ConfigFindLabel(
  ESMC_Config config,        // in
  const char* label          // in
);
```

RETURN VALUE:

```
    Return code; equals ESMF_SUCCESS if there are no errors.
    Equals -1 if buffer could not be loaded, -2 if label not found,
    and -3 if invalid operation with index.
```

DESCRIPTION:

Finds the `label` (key) in the `config` file.
Since the search is done by looking for a word in the whole resource file, it is important to use special conventions to distinguish labels from other words in the resource files. The DAO convention is to finish line labels by : and table labels by ::.
The arguments are:

config Already created `ESMC_Config` object.

label Identifying label.

27.2.4 ESMC_ConfigGetDim - Get table sizes

INTERFACE:

```
int ESMC_ConfigGetDim(
  ESMC_Config config,        // in
  int* lineCount,            // out
  int* columnCount,          // out
  ...                        // optional argument list
);
```

RETURN VALUE:

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

DESCRIPTION:

Returns the number of lines in the table in `lineCount` and the maximum number of words in a table line in `columnCount`.
The arguments are:

config Already created `ESMC_Config` object.

lineCount Returned number of lines in the table.

columnCount Returned maximum number of words in a table line.

[label] Identifying label (optional).

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

27.2.5 ESMC_ConfigGetLen - Get the length of the line in words

INTERFACE:

```
int ESMC_ConfigGetLen(
  ESMC_Config config,        // in
  int* wordCount,            // out
  ...                        // optional argument list
);
```

RETURN VALUE:

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

DESCRIPTION:

Gets the length of the line in words by counting words disregarding types. Returns the word count as an integer.
The arguments are:

config Already created `ESMC_Config` object.

wordCount Returned number of words in the line.

[label] Identifying label. If not specified, use the current line (optional).

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

27.2.6 ESMC_ConfigLoadFile - Load resource file into memory

INTERFACE:

```
int ESMC_ConfigLoadFile(
  ESMC_Config config,        // in
  const char* file,          // in
  ...                        // optional argument list
);
```

RETURN VALUE:

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

DESCRIPTION:

Resource file with `filename` is loaded into memory.
The arguments are:

config Already created `ESMC_Config` object.

file Configuration file name.

[delayout] `ESMC_DELayout` associated with this `config` object. **NOTE: This argument is not currently supported.

[unique] If specified as true, uniqueness of labels are checked and error code set if duplicates found (optional).

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

27.2.7 ESMC_ConfigNextLine - Find next line

INTERFACE:

```
int ESMC_ConfigNextLine(
  ESMC_Config config,       // in
  int *tableEnd             // out
);
```

RETURN VALUE:

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

DESCRIPTION:

Selects the next line (for tables).
The arguments are:

config Already created `ESMC_Config` object.

[tableEnd] End of table mark (::) found flag. Returns 1 when found, and 0 when not found.

27.2.8 ESMC_ConfigValidate - Validate a Config object

INTERFACE:

```
int ESMC_ConfigValidate(
  ESMC_Config config,        // in
  ...                        // optional argument list
);
```

RETURN VALUE:

```
Return code; equals ESMF_SUCCESS if there are no errors.
Equals ESMF_RC_ATTR_UNUSED if any unused attributes are found
with option "unusedAttributes" below.
```

DESCRIPTION:

Checks whether a `config` object is valid.
The arguments are:

config Already created `ESMC_Config` object.

[options] If none specified: simply check that the buffer is not full and the pointers are within range (optional). "unusedAttributes" - Report to the default logfile all attributes not retrieved via a call to `ESMC_ConfigGetAttribute()` or `ESMC_ConfigGetChar()`. The attribute name (label) will be logged via `ESMC_LogErr` with the WARNING log message type. For an array-valued attribute, retrieving at least one value via `ESMC_ConfigGetAttribute()` or `ESMC_ConfigGetChar()` constitutes being "used."

Due to this method accepting optional arguments, the final argument must be `ESMC_ArgLast`.

28 LogErr Class

28.1 Description

The Log class consists of a variety of methods for writing error, warning, and informational messages to files. A default Log is created at ESMF initialization.
When ESMF is started with `ESMC_Initialize()`, multiple Log files will be created by PET number. The PET number (in the format `PETx.`) will be prepended to each file name where x is the PET number. The `ESMC_LogWrite()` call is used to issue messages to the log. As part of the call, a message can be tagged as either an informational, warning, or error message.
The messages may be buffered within ESMF before appearing in the log. All messages will be properly flushed to the log files when `ESMC_Finalize()` is called.

28.2 Class API

28.2.1 ESMC_LogWrite - Write an entry into the Log file

INTERFACE:

```
 int ESMC_LogWrite(
   const char msg[], // in
   int msgtype       // in
 );
```

RETURN VALUE:

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

DESCRIPTION:

Write an entry into the Log file.
The arguments are:

msg The message to be written.

msgtype The message type. Can be one of ESMC_LOG_INFO, ESMC_LOG_WARNING, or ESMF_LOG_ERROR.

29 VM Class

29.1 Description

The ESMF VM (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The

VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component.
In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods. The VM communication calls are very similar to MPI. Data references in VM communication calls must be provided as raw, language specific, one-dimensional, contiguous data arrays. The similarity between VM and MPI communication calls is striking and there are many equivalent point-to-point and collective communication calls. However, unlike MPI, the VM communication calls support communication between threaded PETs in a completely transparent fashion.
Many ESMF applications do not interact with the VM class directly very much. The resource management aspect is wrapped completely transparent into the ESMF Component concept. Often the only reason that user code queries a Component object for the associated VM object is to inquire about resource information, such as the `localPet` or the `petCount`. Further, for most applications the use of higher level communication APIs, such as provided by Array and Field, are much more convenient than using the low level VM communication calls.
The basic elements of a VM are called PETs, which stands for Persistent Execution Threads. These are equivalent to OS threads with a lifetime of at least that of the associated component. All VM functionality is expressed in terms of PETs. In the simplest, and most common case, a PET is equivalent to an MPI process. However, ESMF also supports multi-threading, where multiple PETs run as Pthreads inside the same virtual address space (VAS).

29.2 Class API

29.2.1 ESMC_VMGet - Get VM internals

INTERFACE:

```
int ESMC_VMGet(
  ESMC_VM vm,                   // in
  int *localPet,                // out
  int *petCount,                // out
  int *peCount,                 // out
  MPI_Comm *mpiCommunicator,    // out
  int *pthreadsEnabledFlag,     // out
  int *openMPEnabledFlag        // out
);
```

RETURN VALUE:

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

DESCRIPTION:

Get internal information about the specified `ESMC_VM` object.
The arguments are:

vm Queried `ESMC_VM` object.

[localPet] Upon return this holds the id of the PET that issued this call.

[petCount] Upon return this holds the number of PETs in the specified `ESMC_VM` object.

[peCount] Upon return this holds the number of PEs referenced by the specified `ESMC_VM` object.

[mpiCommunicator] Upon return this holds the MPI intra-communicator used by the specified `ESMC_VM` object. This communicator may be used for user-level MPI communications. It is recommended that the user duplicates the communicator via `MPI_Comm_Dup()` in order to prevent any interference with ESMF communications.

[pthreadsEnabledFlag] A return value of '1' indicates that the ESMF library was compiled with Pthreads enabled. A return value of '0' indicates that Pthreads are disabled in the ESMF library.

[openMPEnabledFlag] A return value of '1' indicates that the ESMF library was compiled with OpenMP enabled. A return value of '0' indicates that OpenMP is disabled in the ESMF library.

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

29.2.2 ESMC_VMGetCurrent - Get current VM

INTERFACE:

```
ESMC_VM ESMC_VMGetCurrent(
  int *rc                     // out
);
```

RETURN VALUE:

```
    VM object of the current execution context.
```

DESCRIPTION:

Get the ESMC_VM object of the current execution context. Calling ESMC_VMGetCurrent() within an ESMF Component, will return the same VM object as ESMC_GridCompGet(..., vm=vm, ...) or ESMC_CplCompGet(..., vm=vm, ...).
The main purpose of providing ESMC_VMGetCurrent() is to simplify ESMF adoption in legacy code. Specifically, code that uses MPI_COMM_WORLD deep within its calling tree can easily be modified to use the correct MPI communicator of the current ESMF execution context. The advantage is that these modifications are very local, and do not require wide reaching interface changes in the legacy code to pass down the ESMF component object, or the MPI communicator.
The use of ESMC_VMGetCurrent() is strongly discouraged in newly written Component code. Instead, the ESMF Component object should be used as the appropriate container of ESMF context information. This object should be passed between the subroutines of a Component, and be queried for any Component specific information.
Outside of a Component context, i.e. within the driver context, the call to ESMC_VMGetCurrent() is identical to ESMC_VMGetGlobal().

The arguments are:

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

29.2.3 ESMC_VMGetGlobal - Get global VM

INTERFACE:

```
ESMC_VM ESMC_VMGetGlobal(
  int *rc                     // out
);
```

RETURN VALUE:

```
    VM object of the global execution context.
```

DESCRIPTION:

Get the global ESMC_VM object. This is the VM object that is created during ESMC_Initialize() and is the ultimate parent of all VM objects in an ESMF application. It is identical to the VM object returned by ESMC_Initialize(..., vm=vm, ...).

The ESMC_VMGetGlobal() call provides access to information about the global execution context via the global VM. This call is necessary because ESMF does not create a global ESMF Component during ESMC_Initialize() that could be queried for information about the global execution context of an ESMF application.
Usage of ESMC_VMGetGlobal() from within Component code is strongly discouraged. ESMF Components should only access their own VM objects through Component methods. Global information, if required by the Component user code, should be passed down to the Component from the driver through the Component calling interface.

The arguments are:

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

29.2.4 ESMC_VMPrint - Print a VM

INTERFACE:

```
int ESMC_VMPrint(
  ESMC_VM vm                // in
);
```

RETURN VALUE:

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

DESCRIPTION:

Print internal information of the specified ESMC_VM object.
The arguments are:

vm ESMC_VM object to be printed.

Part VI
References

References

[1] Khoei S.A. Gharehbaghi A, R. The superconvergent patch recovery technique and data transfer operators in 3d plasticity problems. *Finite Elements in Analysis and Design*, 43(8), 2007.

[2] K.C. Hung H. Gu, Z. Zong. A modified superconvergent patch recovery method and its application to large deformation problems. *Finite Elements in Analysis and Design*, 40(5-6), 2004.

[3] Jones, P.W. SCRIP: A Spherical Coordinate Remapping and Interpolation Package. http://www.acl.lanl.gov/climate/software/SCRIP/. Los Alamos National Laboratory Software Release LACC 98-45.

[4] D. Ramshaw. Conservative rezoning algorithm for generalized two-dimensional meshes. *Journal of Computational Physics*, 59, 1985.

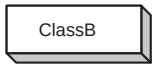
[5] Rumbaugh, J., I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

Part VII
Appendices

30 Appendix A: A Brief Introduction to UML

The schematic below shows the Unified Modeling Language (UML) notation for the class diagrams presented in this *Reference Manual*. For more on UML, see references such as *The Unified Modeling Language Reference Manual*, Rumbaugh et al, [5].

Public class. This is a class whose methods can be called by the user. In Fortran a public class is usually associated with a derived type and a corresponding module that contains class methods and flags.



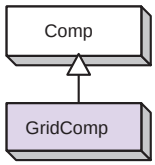
Private class. This type of class does not have methods that should be called by the user. Like a public class it is usually associated with a derived type and a corresponding module.

A line indicates some sort of association among classes.

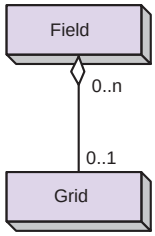
A hollow diamond at one end of a line drawn between classes represents an association called aggregation. Aggregation is a part-whole relationship that can be read as "the class at the end of the line without the diamond is part of the class at the end of the line with the diamond." The class that is the "part" can be created and destroyed separately, and it is usually implemented as a reference contained with the structure of the class that is the "whole."

A filled diamond at one end of a line drawn between classes represents an association called composition. Composition is a part-whole relationship that is similar to aggregation, but stronger. It implies that that class that is the "part" is created and destroyed by the class that is the "whole." It is often implemented as a structure within part of the contiguous memory of a larger structure.

Multiplicity indicators at association line ends show how many classes on the one end are associated with how many classes on the other end.



The triangle indicates an inheritance relationship. Inheritance means that a child class shares a set of characteristics (such as the same attributes or methods) with a parent class. The child can specialize and extend the behavior of the parent. This diagram shows a GridComp class that inherits from a more general Comp class.



This simple diagram shows that a public class called Field is associated with another public class, called Grid. The aggregation relationship indicated by the unfilled diamond means that a Field contains a Grid, but that a Grid can be created and destroyed outside of a Field. The diagram multiplicities show that a Field can be associated with no Grid or with one Grid, but that a single Grid can be associated with any number of Fields.

31 Appendix B: ESMF Error Return Codes

The tables below show the possible error return codes for Fortran and C++ methods.

```
 ====================================================
 Success/Failure Return codes for both Fortran and C++
 ====================================================

  ESMF_SUCCESS             0
  ESMF_FAILURE            -1

 ====================================
 Fortran Symmetric Return Codes 1-500
 ====================================

  ESMF_RC_OBJ_BAD           1
  ESMF_RC_OBJ_INIT          2
  ESMF_RC_OBJ_CREATE        3
  ESMF_RC_OBJ_COR           4
  ESMF_RC_OBJ_WRONG         5
  ESMF_RC_ARG_BAD           6
  ESMF_RC_ARG_RANK          7
  ESMF_RC_ARG_SIZE          8
  ESMF_RC_ARG_VALUE         9
  ESMF_RC_ARG_DUP          10
  ESMF_RC_ARG_SAMETYPE     11
  ESMF_RC_ARG_SAMECOMM     12
  ESMF_RC_ARG_INCOMP       13
  ESMF_RC_ARG_CORRUPT      14
  ESMF_RC_ARG_WRONG        15
  ESMF_RC_ARG_OUTOFRANGE   16
  ESMF_RC_ARG_OPT          17
  ESMF_RC_NOT_IMPL         18
  ESMF_RC_FILE_OPEN        19
  ESMF_RC_FILE_CREATE      20
  ESMF_RC_FILE_READ        21
  ESMF_RC_FILE_WRITE       22
  ESMF_RC_FILE_UNEXPECTED  23
  ESMF_RC_FILE_CLOSE       24
  ESMF_RC_FILE_ACTIVE      25
  ESMF_RC_PTR_NULL         26
  ESMF_RC_PTR_BAD          27
  ESMF_RC_PTR_NOTALLOC     28
  ESMF_RC_PTR_ISALLOC      29
  ESMF_RC_MEM              30
  ESMF_RC_MEM_ALLOCATE     31
  ESMF_RC_MEM_DEALLOCATE   32
  ESMF_RC_MEMC             33
  ESMF_RC_DUP_NAME         34
  ESMF_RC_LONG_NAME        35
  ESMF_RC_LONG_STR         36
  ESMF_RC_COPY_FAIL        37
  ESMF_RC_DIV_ZERO         38
  ESMF_RC_CANNOT_GET       39
  ESMF_RC_CANNOT_SET       40
  ESMF_RC_NOT_FOUND        41
```

```
 ESMF_RC_NOT_VALID         42
 ESMF_RC_INTNRL_LIST       43
 ESMF_RC_INTNRL_INCONS     44
 ESMF_RC_INTNRL_BAD        45
 ESMF_RC_SYS               46
 ESMF_RC_BUSY              47
 ESMF_RC_LIB               48
 ESMF_RC_LIB_NOT_PRESENT   49
 ESMF_RC_ATTR_UNUSED       50
 ESMF_RC_OBJ_NOT_CREATED   51
 ESMF_RC_OBJ_DELETED       52
 ESMF_RC_NOT_SET           53
 ESMF_RC_VAL_WRONG         54
 ESMF_RC_VAL_ERRBOUND      55
 ESMF_RC_VAL_OUTOFRANGE    56
 ESMF_RC_ATTR_NOTSET       57
 ESMF_RC_ATTR_WRONGTYPE    58
 ESMF_RC_ATTR_ITEMSOFF     59
 ESMF_RC_ATTR_LINK         60
 ESMF_RC_BUFFER_SHORT      61

62-499 reserved for future Fortran symmetric return code definitions

====================================
C++ Symmetric Return Codes 501-999
====================================

 ESMC_RC_OBJ_BAD          501
 ESMC_RC_OBJ_INIT         502
 ESMC_RC_OBJ_CREATE       503
 ESMC_RC_OBJ_COR          504
 ESMC_RC_OBJ_WRONG        505
 ESMC_RC_ARG_BAD          506
 ESMC_RC_ARG_RANK         507
 ESMC_RC_ARG_SIZE         508
 ESMC_RC_ARG_VALUE        509
 ESMC_RC_ARG_DUP          510
 ESMC_RC_ARG_SAMETYPE     511
 ESMC_RC_ARG_SAMECOMM     512
 ESMC_RC_ARG_INCOMP       513
 ESMC_RC_ARG_CORRUPT      514
 ESMC_RC_ARG_WRONG        515
 ESMC_RC_ARG_OUTOFRANGE   516
 ESMC_RC_ARG_OPT          517
 ESMC_RC_NOT_IMPL         518
 ESMC_RC_FILE_OPEN        519
 ESMC_RC_FILE_CREATE      520
 ESMC_RC_FILE_READ        521
 ESMC_RC_FILE_WRITE       522
 ESMC_RC_FILE_UNEXPECTED  523
 ESMC_RC_FILE_CLOSE       524
 ESMC_RC_FILE_ACTIVE      525
 ESMC_RC_PTR_NULL         526
 ESMC_RC_PTR_BAD          527
 ESMC_RC_PTR_NOTALLOC     528
```

```
 ESMC_RC_PTR_ISALLOC      529
 ESMC_RC_MEM              530
 ESMC_RC_MEM_ALLOCATE     531
 ESMC_RC_MEM_DEALLOCATE   532
 ESMC_RC_MEMC             533
 ESMC_RC_DUP_NAME         534
 ESMC_RC_LONG_NAME        535
 ESMC_RC_LONG_STR         536
 ESMC_RC_COPY_FAIL        537
 ESMC_RC_DIV_ZERO         538
 ESMC_RC_CANNOT_GET       539
 ESMC_RC_CANNOT_SET       540
 ESMC_RC_NOT_FOUND        541
 ESMC_RC_NOT_VALID        542
 ESMC_RC_INTNRL_LIST      543
 ESMC_RC_INTNRL_INCONS    544
 ESMC_RC_INTNRL_BAD       545
 ESMC_RC_SYS              546
 ESMC_RC_BUSY             547
 ESMC_RC_LIB              548
 ESMC_RC_LIB_NOT_PRESENT  549
 ESMC_RC_ATTR_UNUSED      550
 ESMC_RC_OBJ_NOT_CREATED  551
 ESMC_RC_OBJ_DELETED      552
 ESMC_RC_NOT_SET          553
 ESMC_RC_VAL_WRONG        554
 ESMC_RC_VAL_ERRBOUND     555
 ESMC_RC_VAL_OUTOFRANGE   556
 ESMC_RC_ATTR_NOTSET      557
 ESMC_RC_ATTR_WRONGTYPE   558
 ESMC_RC_ATTR_ITEMSOFF    559
 ESMC_RC_ATTR_LINK        560
 ESMC_RC_BUFFER_SHORT     561

562-999 reserved for future C++ symmetric return code definitions

====================================
C++ Non-symmetric Return Codes 1000
====================================

 ESMC_RC_OPTARG_BAD      1000
```