

Earth System Modeling Framework  
**ESMF Reference Manual for Fortran**

**Version 2.0**

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

24th September 2004

## Acknowledgements

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

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

## **Contents**

**Part I**  
**ESMF Overview**

## 1 Release Notes

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

## 2 What is the Earth System Modeling Framework?

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

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

## 3 The ESMF Reference Manual for Fortran

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

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

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

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

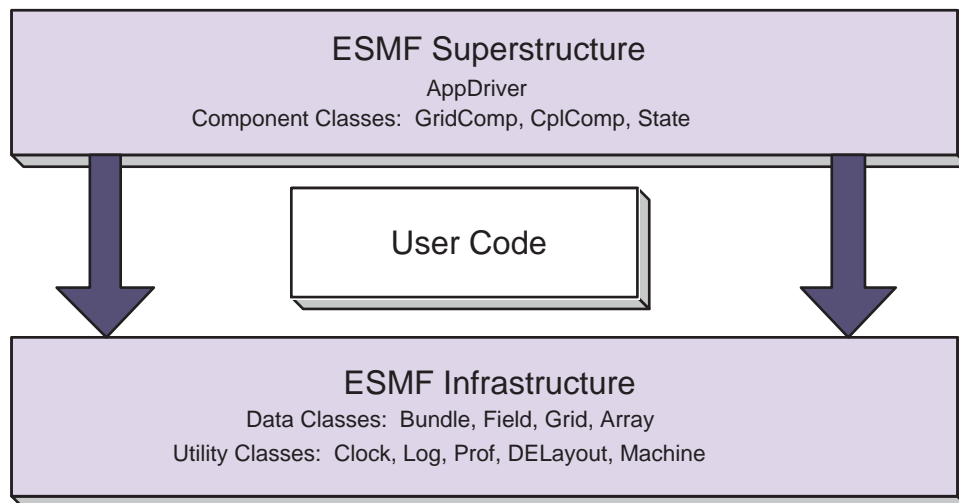
## 4 How to Contact User Support and Find Additional Information

The ESMF team can answer questions about the interfaces presented in this document. For user support, please contact [esmf\\_support@ucar.edu](mailto:esmf_support@ucar.edu).

---

<sup>1</sup>Since the audience for it is small, we have not yet prepared a comprehensive reference manual for C++.

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



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

## 5 How to Submit New Requirements

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

## 6 Conventions

### 6.1 Document Conventions

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

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

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

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

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

## 6.2 Method Name and Argument Conventions

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

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

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

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

A typical ESMF call thus looks like this:

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

where

<ClassName> is the class name,

<method> is the name of the action to be performed,

classname is a variable of the derived type associated with the class,

the `arg*` arguments are whatever other variables are required for the operation,

and `rc` is a return code.

## 6.3 Locating Methods in this Manual

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

## 7 The ESMF Application Programming Interface

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

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

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

### 7.1 Standard Methods and Interface Rules

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

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

#### EXAMPLE

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

```
USE ESMF_Mod

type ESMF_Field :: field

field = ESMF_FieldCreate('temp')
```

## 7.2 Deep and Shallow Classes

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

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

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

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

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

## 7.3 Special Methods

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

- `ESMF_Initialize()` and `ESMF_Finalize()` are required methods that must bracket the use of ESMF within an application. They manage the resources required to run ESMF and shut it down gracefully.
- `ESMF_<Type>CompInitialize()`, `ESMF_<Type>CompRun()`, and `ESMF_<Type>CompFinalize()` are component methods that are used at the highest level within ESMF. `<Type>` may be `<Grid>`, for Gridded Components such as oceans or atmospheres, or `<Cpl>`, for Coupler Components that are used to connect them. The content of these methods is not part of the ESMF. Instead the methods call into associated Fortran subroutines within user code.

## 7.4 The ESMF Data Hierarchy

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

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

- **Array** An ESMF Array is a distributed, multi-dimensional array that can carry information such as its type, kind, rank, and associated halo widths. It contains a reference to a native Fortran array.
- **Field** A Field represents a physical scalar or vector field. It contains a reference to an Array along with grid information and metadata.
- **Bundle** A Bundle is a collection of Fields discretized on the same grid. The staggering of data points may be different for different Fields within a Bundle.
- **State** A State represents the collection of data that a Component either requires to run (an Import State) or can make available to other Components (an Export State). States may contain references to Bundles, Fields, or Arrays.

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

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

## 7.5 ESMF Spatial Classes

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

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

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

## 7.6 ESMF DataMap Classes

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

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

## 7.7 ESMF Specification Classes

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

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

## 7.8 ESMF Utility Classes

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

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

# 8 Overall Rules and Behavior

## 8.1 Allocation Rules

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

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

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

## 8.2 Attributes

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

## 9 Integrating ESMF into Applications

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

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

### 9.1 Using the ESMF Superstructure

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

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

### 9.2 Using the ESMF Infrastructure

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

---

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

## 10 Global Options and Parameters

### 10.1 Flags

#### 10.1.1 ESMF\_AllocFlag

**DESCRIPTION:**

Indicates whether to allocate data or not.

Valid values are:

**ESMF\_ALLOC** Allocate data.

**ESMF\_NO\_ALLOC** Do not allocate data at this time.

#### 10.1.2 ESMF\_BlockingFlag

**DESCRIPTION:**

Indicates whether to block during a communication call.

Valid values are:

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

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

#### 10.1.3 ESMF\_CopyFlag

**DESCRIPTION:**

Indicates whether to reference a data item or make a copy of it.

Valid values are:

**ESMF\_DATA\_COPY** Copy the data item to another buffer.

**ESMF\_DATA\_REF** Reference the data item.

#### 10.1.4 ESMF\_IndexFlag

**DESCRIPTION:**

Indicates whether index is local (per DE) or global (per object).

Valid values are:

**ESMF\_INDEX\_DELOCAL** Refers to indices on the local DE.

**ESMF\_INDEX\_GLOBAL** Refers to object-wide indices.

#### 10.1.5 ESMF\_InterleaveFlag

**DESCRIPTION:**

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

Valid values are:

**ESMF\_INTERLEAVE\_BY\_BLOCK** Items are listed in blocks, all items of one type followed by all items of the next type.

**ESMF\_INTERLEAVE\_BY\_ITEM** Items are interleaved item by item.

### 10.1.6 ESMF\_NeededFlag

**DESCRIPTION:**

Specifies whether or not a data item is needed for a particular application configuration. Used in ESMF\_State.

Valid values are:

**ESMF\_NEEDED** Data is needed.

**ESMF\_NOTNEEDED** Data is not needed.

### 10.1.7 ESMF\_ReadyFlag

**DESCRIPTION:**

Specifies whether a data item is ready to read or write.

Valid values are:

**ESMF\_READYTOREAD** Data is ready to read.

**ESMF\_READYTOWRITE** Data is ready to write.

**ESMF\_NOTREADY** Data is not ready.

### 10.1.8 ESMF\_ReduceFlag

**DESCRIPTION:**

Indicates reduce operation to a Reduce ( ) method.

Valid values are:

**ESMF\_SUM** Use arithmetic sum to add all data elements.

**ESMF\_MIN** Determine the minimum of all data elements.

**ESMF\_MAX** Determine the maximum of all data elements.

### 10.1.9 ESMF\_ReqForRestartFlag

**DESCRIPTION:**

Specifies whether a data item is necessary for restart.

Valid values are:

**ESMF\_REQUIRED\_FOR\_RESTART** Data is required for restart.

**ESMF\_NOTREQUIRED\_FOR\_RESTART** Data is not required for restart.

### 10.1.10 ESMF\_ValidFlag

**DESCRIPTION:**

Specifies whether a data item contains valid data.

Valid values are:

**ESMF\_VALID** Data is ready to read.

**ESMF\_INVALID** Data is ready to write.

**ESMF\_NOTREADY** Data is not ready.

## 10.2 Parameters

### 10.2.1 ESMF\_DataKind

DESCRIPTION:

Supported ESMF data kinds.

Valid values are:

**ESMF\_I1** 1 byte integer.

**ESMF\_I2** 2 byte integer.

**ESMF\_I4** 4 byte integer.

**ESMF\_I8** 8 byte integer.

**ESMF\_R4** 4 byte real.

**ESMF\_R8** 8 byte real.

**ESMF\_C8** 8 byte character.

**ESMF\_C16** 16 byte character.

### 10.2.2 ESMF\_DataType

DESCRIPTION:

Supported ESMF data types.

Valid values are:

**ESMF\_DATA\_INTEGER** Integer type.

**ESMF\_DATA\_REAL** Real type.

**ESMF\_DATA\_LOGICAL** Logical type.

**ESMF\_DATA\_CHARACTER** Character type.

## 11 Overall Design and Implementation Notes

1. **Deep and shallow classes.** The deep and shallow classes described in Section 7.2 differ in how and where they are allocated within a multi-language implementation environment. We distinguish between the implementation language, which is the language a method is written in, and the calling language, which is the language that the user application is written in. Deep classes are allocated off the process heap by the implementation language. Shallow classes are allocated off the stack by the calling language.
2. **Base class.** All ESMF classes are built upon a Base class. The Base is used to hold system-wide capabilities, such as Attributes. Attributes are implemented in the Base class so they can be attached to any object in the system which is built on the Base object. (This is true for all deep objects in the system.) Attributes are created by making a private copy of the information provided during the Set call. Lists of values are supported, but they are not intended for large data arrays. Attribute data is copied during a Get operation.

**Part II**  
**Superstructure**

## 12 Overview of Superstructure

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

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

### Key Features

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

### 12.1 Superstructure Classes

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

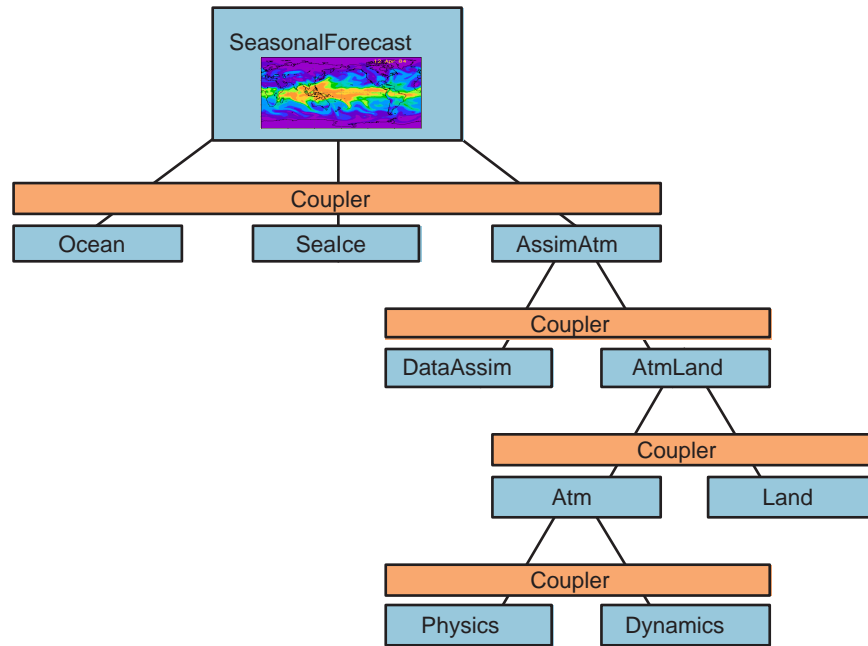
- **Component** An ESMF component has two parts, one that is supplied by the ESMF and one that is supplied by the user. The part that is supplied by the framework is an ESMF derived type that is either a Gridded Component (**GridComp**) or a Coupler Component (**CplComp**). A Gridded Component typically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

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

- **State** ESMF components exchange information with other components only through States. A State is an ESMF derived type that can contain Fields, Bundles, Arrays, and other States. A Gridded Component is associated with two States, an **Import State** and an **Export State**. Its Import State holds the data that it receives from other Gridded Components. Its Export State contains data that it can make available to other Gridded Components.
- **Application Driver** The Application Driver (**AppDriver**) is a small, generic driver program that contains the “main” routine for an ESMF application.

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

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



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

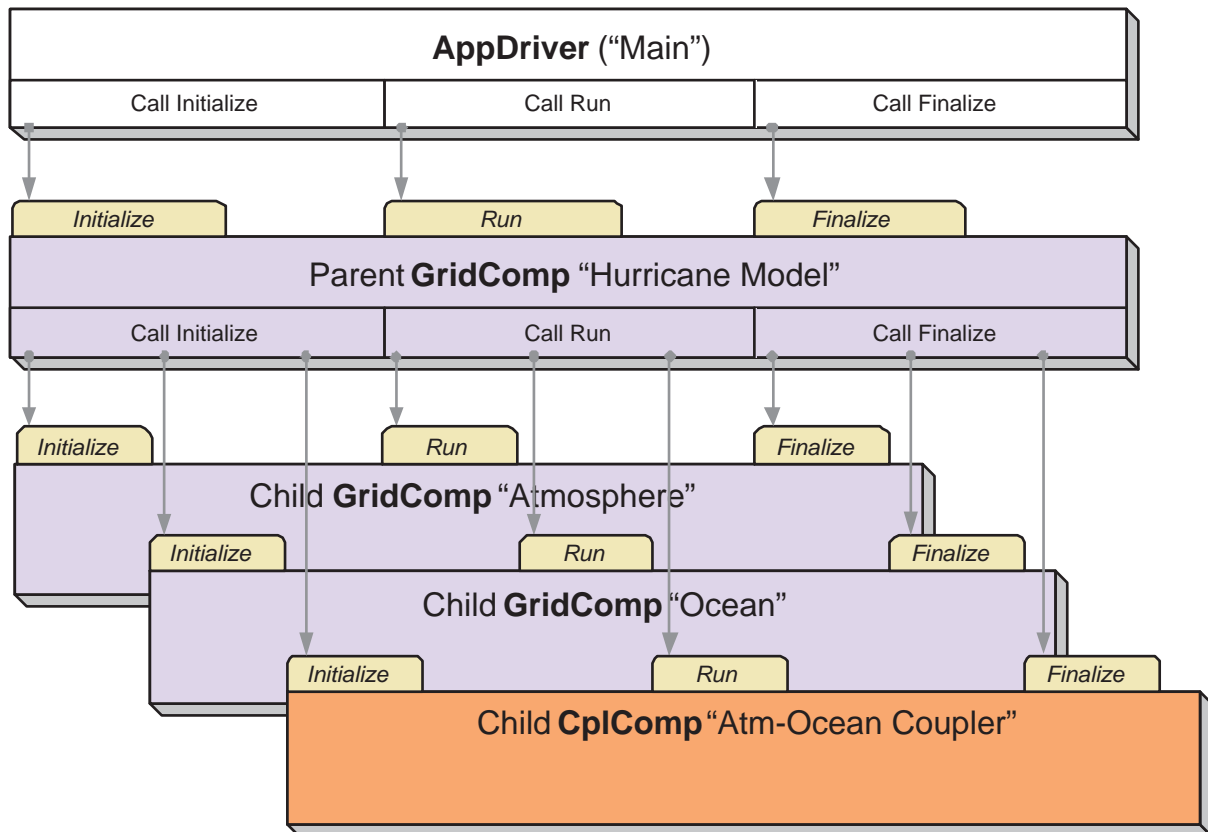
## 12.2 Distribution and Scoping of Components

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

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

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

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



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

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

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

### 12.3 Performance

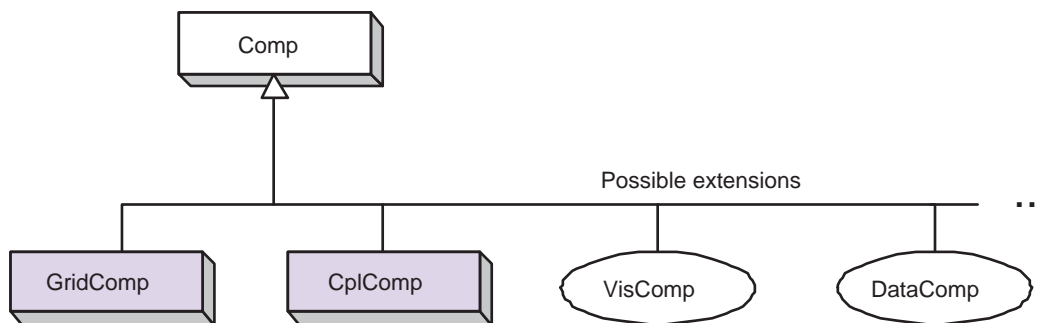
The ESMF design enables the user to configure ESMF applications so that data is transferred directly from one component to another, without requiring that it be copied or sent to a different data buffer as an interim step. This is likely

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

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

## 12.4 Object Model

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



## 13 Application Driver and Required ESMF Methods

### 13.1 Description

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

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

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

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

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

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

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

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

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

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

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

## 13.2 Use and Examples

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

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

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

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

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

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

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

```
#include "ChangeMe.F90"
```

```
program ESMF_AppDriver
```

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

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

```
implicit none
```

```
! Local variables
```

```

! Components
type(ESMF_GridComp) :: compGridded

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

! Configuration information
type(ESMF_Config) :: config

! A common grid
type(ESMF_Grid) :: grid

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

!-----
!-----
!   Register section
!-----
!-----

call ESMF_GridCompSetServices(compGridded, SetServices, rc)
if (ESMF_LogMsgFoundError(rc, "Registration failed", rc)) goto 10

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

```

```

! Clean up

call ESMF_ClockDestroy(clock, rc)

call ESMF_CalendarDestroy(gregorianCalendar, rc)

call ESMF_StateDestroy(defaultstate, rc)

call ESMF_GridCompDestroy(compGridded, rc)

call ESMF_DELayoutDestroy(defaultLayout, rc)

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

10 continue

    call ESMF_Finalize(rc)

end program ESMF_AppDriver

```

### 13.3 Restrictions and Future Work

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

### 13.4 Required ESMF Methods

---

#### 13.4.1 ESMF\_Initialize - Initialize the ESMF

INTERFACE:

```

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

```

ARGUMENTS:

```

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

```

DESCRIPTION:

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

The arguments are:

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

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

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

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

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

---

### 13.4.2 ESMF\_Finalize - Clean up and close the ESMF

INTERFACE:

```
subroutine ESMF_Finalize(rc)
```

ARGUMENTS:

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

DESCRIPTION:

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

The argument is:

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

### 13.4.3 User-Code SetServices Method

Many programs call some library routines. The library documentation must explain what the routine name is, what arguments are required and what are optional, and what the code does.

In contrast, all ESMF components must be written to *be called* by another part of the program; in effect, an ESMF component takes the place of a library. The interface is prescribed by the framework, and the component writer must provide specific subroutines which have standard argument lists and perform specific operations.

One of the required interfaces a component must provide is the set services method. This subroutine must have an externally accessible name (be a public symbol), take a component as the first argument, and an integer return code as the second. The subroutine name is not predefined, it is set by the component writer, but must be provided as part of the component documentation.

The required function of the set services subroutine is to register the rest of the required functions in the component, currently initialize, run, and finalize methods. The ESMF method `ESMF_<Grid/Cpl>CompSetEntryPoint()` should be called for each of the required subroutines.

The names of the initialize, run, and finalize user-code subroutines do not need to be public; in fact it is far better for them to be private to lower the chances of public symbol clashes between different components.

Within the set services routine, the user can also register a private data block by calling the `ESMF_<Grid|Cpl>CompSetInternalS` method.

Note that a component does not call its own set services routine; the AppDriver or parent component code which is creating a component will first call `ESMF_<Grid/Cpl>CompCreate()` to create an "empty" component, and then call the component-specific set services routine to associate ESMF-standard methods to user-code methods. After set services has been called, the framework now will be able to call the component's initialize, run, and finalize routines as required.

### 13.4.4 User-Code Initialize, Run, and Finalize Methods

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

## 14 GridComp Class

### 14.1 Description

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

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

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

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

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

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

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

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

### 14.2 GridComp Options

#### 14.2.1 ESMF\_GridCompType

##### DESCRIPTION:

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

Valid values are:

**ESMF\_ATM** Atmospheric model.

---

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

**ESMF\_LAND** Land model.

**ESMF\_OCEAN** Ocean model.

**ESMF\_SEAICE** Sea ice model.

**ESMF\_RIVER** River model.

**ESMF\_OTHER** Other type of model or system.

### 14.3 Use and Examples

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

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

! ESMF Framework module
use ESMF_Mod
implicit none
public GComp_SetServices

contains
```

---

#### 14.3.1 Specifying a User-Code SetServices Routine

Every `ESMF_GridComp` is required to provide and document a set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_GridComp` as the first argument, and an integer return code as the second.

It must call the ESMF method `ESMF_GridCompSetEntryPoint()` to register with the framework what user-code subroutines should be called to initialize, run, and finalize the component. There are additional routines which can be registered as well, for checkpoint and restart functions.

Note that the actual subroutines being registered do not have to be public to this module; only the set services routine itself must be available to be used by other code.

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

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

```

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

rc = ESMF_SUCCESS

end subroutine

```

---

### 14.3.2 Specifying a User-Code Initialize Routine

When a higher level component is ready to begin using an `ESMF_GridComp`, it will call its initialize routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.

At initialization time the component can allocate data space, open data files, set up initial conditions; anything it needs to do to prepare to run.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```

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

  print *, "Gridded Comp Init starting"

  ! This is where the model specific setup code goes.

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

  rc = ESMF_SUCCESS

end subroutine GComp_Init

```

---

### 14.3.3 Specifying a User-Code Run Routine

During the execution loop, the run routine may be called many times. Each time it should read data from the `importState`, use the `clock` to determine what the current time is in the calling component, compute new values or process the data, and produce any output and place it in the `exportState`.

When a higher level component is ready to use the `ESMF_GridComp` it will call its run routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.

It is expected that this is where the bulk of the model computation or data analysis will occur.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```

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

```

```

type(ESMF_Clock) :: clock
integer :: rc

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

! This is where the model specific computation goes.

! Fill export state here using ESMF_StateAddField(), etc

print *, "Gridded Comp Run returning"

rc = ESMF_SUCCESS

end subroutine GComp_Run

```

---

#### 14.3.4 Specifying a User-Code Finalize Routine

At the end of application execution, each `ESMF_GridComp` should deallocate data space, close open files, and flush final results. These functions should be placed in a finalize routine.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```

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

  print *, "Gridded Comp Final starting"

  ! Add whatever code here needed

  print *, "Gridded Comp Final returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Final

end module ESMF_GriddedCompEx

```

### 14.4 Restrictions and Future Work

1. **No concurrent components.** While the design of concurrently running Components is fairly complete, this release of the framework does not support them. Only sequentially executing Components can be run.
2. **No Transforms.** Components must exchange data through `ESMF_State` objects. The input data are available at the time the user Component code is called, and data to be returned to another Component are available when that code returns. `ESMF_Xform` objects provide a way for a Component to prepare data to be transformed and sent to another Component from within the execution of the user Component code. Transforms are not implemented in this version of the framework.

3. **Data isolation.** Gridded Components must only communicate with other components via data in State objects. They must not make direct references to data in other States.
4. **Namespace isolation.** If possible, Gridded Components should attempt to make all data private, so public names do not interfere with data in other components.
5. **Single execution mode.** It is not expected that a single Gridded Component be able to function in both sequential and concurrent modes, although Gridded Components of different types can be nested. For example, a concurrently called Gridded Component can contain several nested sequential Gridded Components. However, only sequentially executing Components are supported in this release of the framework.

## 14.5 Class API: Basic GridComp Methods

### 14.5.1 ESMF\_GridCompCreate - Create a new GridComp from a Config file

INTERFACE:

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

RETURN VALUE:

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

ARGUMENTS:

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

DESCRIPTION:

Create a new `ESMF_GridComp`, specifying optional configuration file and other information.

The return value is the new `ESMF_GridComp`.

The arguments are:

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

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

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

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

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

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

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

---

## 14.5.2 ESMF\_GridCompCreate - Create a new GridComp with VM enabled

### INTERFACE:

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

### RETURN VALUE:

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

### ARGUMENTS:

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

### DESCRIPTION:

Create a new `ESMF_GridComp`, setting the resources explicitly.

The return value is the new `ESMF_GridComp`.

The arguments are:

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

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

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

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

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

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

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

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

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

---

### 14.5.3 ESMF\_GridCompDestroy - Release resources for a GridComp

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

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

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

---

### 14.5.4 ESMF\_GridCompFinalize - Call the GridComp's finalize routine

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

**gridcomp** The `ESMF_GridComp` to call finalize routine for.

**[importState]** `ESMF_State` containing import data.

**[exportState]** ESMF\_State containing export data.

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

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

**[blockingflag]** Use ESMF\_BLOCKING (default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 14.5.5 ESMF\_GridCompGet - Query a GridComp for information

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

The arguments are:

**[gridcomp]** ESMF\_GridComp object to query.

**[name]** Return the name of the ESMF\_GridComp.

**[gridcomptype]** Return the model type of this ESMF\_GridComp.

**[grid]** Return the ESMF\_Grid associated with this ESMF\_GridComp.

**[config]** Return the ESMF\_Config object for this ESMF\_GridComp.

**[configFile]** Return the configuration filename for this ESMF\_GridComp.

**[clock]** Return the private clock for this ESMF\_GridComp.

**[vm]** Return the ESMF\_VM for this ESMF\_GridComp.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.5.6 ESMF\_GridCompInitialize - Call the GridComp's initialize routine

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

**gridcomp** ESMF\_GridComp to call initialize routine for.

**[importState]** ESMF\_State containing import data for coupling.

**[exportState]** ESMF\_State containing export data for coupling.

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

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

**[blockingflag]** Valid values are ESMF\_BLOCKING (the default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.5.7 ESMF\_GridCompPrint - Print the contents of a GridComp

#### INTERFACE:

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

#### ARGUMENTS:

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

## DESCRIPTION:

Prints information about an ESMF\_GridComp to stdout.  
The arguments are:

**gridcomp** ESMF\_GridComp to print.

**[options]** Print options are not yet supported.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.5.8 ESMF\_GridCompReadRestart - Call the GridComp's restore routine

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

**gridcomp** ESMF\_GridComp object to call readrestart routine for.

**[iospec]** ESMF\_IOSpec object which describes I/O options.

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

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

**[blockingflag]** Valid values are ESMF\_BLOCKING (the default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.5.9 ESMF\_GridCompRun - Call the GridComp's run routine

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Call the associated user run code for an ESMF\_GridComp.  
The arguments are:

**gridcomp** ESMF\_GridComp to call run routine for.

**[importState]** ESMF\_State containing import data.

**[exportState]** ESMF\_State containing export data.

**[clock]** External clock for passing in time information.

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

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

**[blockingflag]** Use ESMF\_BLOCKING (default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.5.10 ESMF\_GridCompSet - Set or reset information about the GridComp

#### INTERFACE:

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

#### ARGUMENTS:

```

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

```

#### DESCRIPTION:

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

The arguments are:

**gridcomp** ESMF\_GridComp to change.

**[name]** Set the name of the ESMF\_GridComp.

**[gridcomptype]** Set the model type for this ESMF\_GridComp.

**[grid]** Set the ESMF\_Grid associated with the ESMF\_GridComp.

**[config]** Set the configuration information for the ESMF\_GridComp from this already created ESMF\_Config object. If specified, takes priority over `configFile`.

**[configFile]** Set the configuration filename for this ESMF\_GridComp. An ESMF\_Config object will be created for this file and attached to the ESMF\_GridComp. Superceded by `config` if both are specified.

**[clock]** Set the private clock for this ESMF\_GridComp.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

#### 14.5.11 ESMF\_GridCompValidate - Check validity of a GridComp

##### INTERFACE:

```

subroutine ESMF_GridCompValidate(gridcomp, options, rc)

```

##### ARGUMENTS:

```

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

```

##### DESCRIPTION:

Currently all this method does is to check that the `gridcomp` exists.

The arguments are:

**gridcomp** ESMF\_GridComp to validate.

**[options]** Validation options are not yet supported.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

### 14.5.12 ESMF\_GridCompWriteRestart - Call the GridComp's checkpoint routine

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Call the associated user checkpoint code for an ESMF\_GridComp.

The arguments are:

**gridcomp** ESMF\_GridComp to call writerestart routine for.

**[iospec]** ESMF\_IOSpec object which describes I/O options.

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

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

**[blockingflag]** Valid values are ESMF\_BLOCKING (the default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.5.13 ESMF\_GridCompWait - Wait for a GridComp to return

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

When executing asynchronously, wait for an ESMF\_GridComp to return.

The arguments are:

**gridcomp** ESMF\_GridComp to wait for.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

## 14.6 Class API: SetServices and Related Methods

### 14.6.1 ESMF\_GridCompGetInternalState - Get private data block pointer

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

**gridcomp** An `ESMF_GridComp` object.

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

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

---

### 14.6.2 ESMF\_GridCompSetEntryPoint - Set name of GridComp subroutines

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Intended to be called by an `ESMF_GridComp` during the registration process. An `ESMF_GridComp` calls `ESMF_GridCompSetEntryPoint` for each of the predefined `init`, `run`, and `finalize` routines, to associate the internal subroutine to be called for each function. If multiple phases for `init`, `run`, or `finalize` are needed, this can be called with phase numbers.

After this subroutine returns, the framework now knows how to call the initialize, run, and finalize routines for this child `ESMF_GridComp`.

The arguments are:

**gridcomp** An ESMF\_GridComp object.

**subroutineType** One of a set of predefined subroutine types - e.g. ESMF\_SETINIT, ESMF\_SETRUN, ESMF\_SETFINAL.

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

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.6.3 ESMF\_GridCompSetInternalState - Set private data block pointer

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

The arguments are:

**gridcomp** An ESMF\_GridComp object.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 14.6.4 ESMF\_GridCompSetServices - Register GridComp interface routines

INTERFACE:

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

ARGUMENTS:

```

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

```

## DESCRIPTION:

Call a gridded `ESMF_GridComp`'s `setservices` registration routine. The parent component must first create an `ESMF_GridComp`, then call this routine. The arguments are the object returned from the create call, plus the public, well-known subroutine name that is the registration routine for this `ESMF_GridComp`. This name must be documented by the `ESMF_GridComp` provider.

After this subroutine returns, the framework now knows how to call the initialize, run, and finalize routines for the `ESMF_GridComp`.

The arguments are:

**gridcomp** An `ESMF_GridComp` object.

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

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

## 15 CplComp Class

### 15.1 Description

In a large, multi-component application such as a weather forecasting or climate prediction system running within ESMF, physical domains and major system functions are represented as Gridded Components (see Section ??). A Coupler Component, or `ESMF_CplComp`, arranges and executes the data transformations between the Gridded Components. Ideally, Coupler Components should contain all the information about inter-component communication for an application. This enables the Gridded Components in the application to be used in multiple contexts; that is, used in different coupled configurations without changes to their source code. For example, the same atmosphere might in one case be coupled to an ocean in a hurricane prediction model, and in another coupled to a data assimilation system for numerical weather prediction.

Like Gridded Components, Coupler Components have two parts, one that is provided by the user and another that is part of the framework. The user-written portion of the software is the coupling code necessary for a particular exchange between Gridded Components. The term “user-written” is somewhat misleading here, since within a Coupler Component the user can leverage ESMF infrastructure software for regridding, redistribution, lower-level communications, calendar management, and other functions. However, ESMF is unlikely to offer all the software necessary to customize a data transfer between Gridded Components. ESMF does not currently offer tools for unit transformations or time averaging operations, so users must manage those operations themselves.

The user-written Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.

The second part of a Coupler Component is the `ESMF_CplComp` derived type within ESMF. The user must create one of these types to represent a specific coupling function, such as the regular transfer of data between a data assimilation system and an atmospheric model.<sup>4</sup>

The user-written part of a Coupler Component is associated with an `ESMF_CplComp` derived type through a routine called `SetServices`. This is a routine that the user must write, and declare public. Inside the `SetServices` routine the user must call `ESMF_SetEntryPoint` methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code. For example, a user routine called “`couplerInit`” might be associated with the standard initialize routine in a Coupler Component.

Coupler Components can be written to transform data between a pair of Gridded Components, or a single Coupler Component can couple more than two Gridded Components.

---

<sup>4</sup>It is not necessary to create a Coupler Component for each individual data *transfer*.

## 15.2 Use and Examples

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

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

! ESMF Framework module
use ESMF_Mod
implicit none
public CPL_SetServices

contains
```

---

### 15.2.1 Specifying a User-Code SetServices Routine

Every ESMF\_CplComp is required to provide and document a set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an ESMF\_CplComp as the first argument, and an integer return code as the second.

It must call the ESMF method ESMF\_CplCompSetEntryPoint() to register with the framework what user-code subroutines should be called to initialize, run, and finalize the component. There are additional routines which can be registered as well, for checkpoint and restart functions.

Note that the actual subroutines being registered do not have to be public to this module; only the set services routine itself must be available to be used by other code.

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

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

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

rc = ESMF_SUCCESS
end subroutine
```

---

### 15.2.2 Specifying a User-Code Initialize Routine

When a higher level component is ready to begin using an `ESMF_CplComp`, it will call its initialize routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.

At initialization time the component can allocate data space, open data files, set up initial conditions; anything it needs to do to prepare to run.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

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

  print *, "Coupler Init starting"

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

  rc = ESMF_SUCCESS

  print *, "Coupler Init returning"

end subroutine CPL_Init
```

---

### 15.2.3 Specifying a User-Code Run Routine

During the execution loop, the run routine may be called many times. Each time it should read data from the `importState`, use the `clock` to determine what the current time is in the calling component, compute new values or process the data, and produce any output and place it in the `exportState`.

When a higher level component is ready to use the `ESMF_CplComp` it will call its run routine. The component writer must supply a subroutine with the exact calling sequence below; no arguments can be optional, and the types and order must match.

It is expected that this is where the bulk of the model computation or data analysis will occur.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

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

  type(ESMF_State) :: nestedstate

  print *, "Coupler Run starting"

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

```

    rc = ESMF_SUCCESS

    print *, "Coupler Run returning"

end subroutine CPL_Run

```

---

### 15.2.4 Specifying a User-Code Finalize Routine

At the end of application execution, each `ESMF_CplComp` should deallocate data space, close open files, and flush final results. These functions should be placed in a finalize routine.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```

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

  type(ESMF_State) :: nestedstate

  print *, "Coupler Final starting"

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

  rc = ESMF_SUCCESS

  print *, "Coupler Final returning"

end subroutine CPL_Final

end module ESMF_CouplerEx

```

### 15.3 Restrictions and Future Work

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

## 15.4 Class API: Basic CplComp Methods

### 15.4.1 ESMF\_CplCompCreate - Create a new CplComp from a Config file

#### INTERFACE:

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

#### RETURN VALUE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Create a new `ESMF_CplComp`, specifying optional configuration file information.

The return value is the new `ESMF_CplComp`.

The arguments are:

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

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

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

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

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

---

### 15.4.2 ESMF\_CplCompCreate - Create a new CplComp with VM enabled

#### INTERFACE:

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

#### RETURN VALUE:

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

#### ARGUMENTS:

```

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

```

#### DESCRIPTION:

Create a new ESMF\_CplComp, setting the resources explicitly.

The return value is the new ESMF\_CplComp.

The arguments are:

**vm** ESMF\_VM object for the parent component out of which this ESMF\_CplCompCreate() call is issued. This will become the parent ESMF\_VM of the newly created ESMF\_CplComp.

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

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

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

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

**[petList]** List of PETS in the given ESMF\_VM that the parent component is giving to the created child component. If petList is not specified all of the parents PETS will be given to the child component.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

### 15.4.3 ESMF\_CplCompDestroy - Release resources for a CplComp

#### INTERFACE:

```

subroutine ESMF_CplCompDestroy(cplcomp, rc)

```

#### ARGUMENTS:

```

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

```

#### DESCRIPTION:

Releases all resources associated with this ESMF\_CplComp.

The arguments are:

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

#### 15.4.4 ESMF\_CplCompFinalize - Call the CplComp's finalize routine

##### INTERFACE:

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

##### ARGUMENTS:

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

##### DESCRIPTION:

Call the associated user-supplied finalization routine for an ESMF\_CplComp.  
The arguments are:

**cplcomp** The ESMF\_CplComp to call finalize routine for.

**[importState]** ESMF\_State containing import data for coupling.

**[exportState]** ESMF\_State containing export data for coupling.

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

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

**[blockingflag]** Use ESMF\_BLOCKING (default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 15.4.5 ESMF\_CplCompGet - Query a CplComp for information

##### INTERFACE:

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

##### ARGUMENTS:

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

## DESCRIPTION:

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

The arguments are:

**cplcomp** ESMF\_CplComp to query.

**[name]** Return the name of the ESMF\_CplComp.

**[config]** Return the ESMF\_Config object for this ESMF\_CplComp.

**[configFile]** Return the configuration filename for this ESMF\_CplComp.

**[clock]** Return the private clock for this ESMF\_CplComp.

**[vm]** Return the ESMF\_VM for this ESMF\_CplComp.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 15.4.6 ESMF\_CplCompInitialize - Call the CplComp's initialize routine

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Call the associated user initialization code for an ESMF\_CplComp.

The arguments are:

**cplcomp** ESMF\_CplComp to call initialize routine for.

**[importState]** ESMF\_State containing import data for coupling.

**[exportState]** ESMF\_State containing export data for coupling.

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

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

**[blockingflag]** Valid values are ESMF\_BLOCKING (the default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 15.4.7 ESMF\_CplCompPrint - Print the contents of a CplComp

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Prints information about an ESMF\_CplComp to stdout.

The arguments are:

**cplcomp** ESMF\_CplComp to print.

**[options]** Print options are not yet supported.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 15.4.8 ESMF\_CplCompReadRestart – Call the CplComp’s restore routine

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Call the associated user restore code for an ESMF\_CplComp.

The arguments are:

**cplcomp** ESMF\_CplComp to call readrestart routine for.

**[iospec]** ESMF\_IOSpec object which describes I/O options.

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

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

**[blockingflag]** Valid values are ESMF\_BLOCKING (the default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 15.4.9 ESMF\_CplCompRun - Call the CplComp's run routine

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Call the associated user run code for an ESMF\_CplComp.

The arguments are:

**cplcomp** ESMF\_CplComp to call run routine for.

**[importState]** ESMF\_State containing import data for coupling.

**[exportState]** ESMF\_State containing export data for coupling.

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

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

**[blockingflag]** Use ESMF\_BLOCKING (default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 15.4.10 ESMF\_CplCompSet - Set or reset information about the CplComp

##### INTERFACE:

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

##### ARGUMENTS:

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

##### DESCRIPTION:

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

The arguments are:

**cplcomp** ESMF\_CplComp to change.

**[name]** Set the name of the ESMF\_CplComp.

**[config]** Set the configuration information for the ESMF\_CplComp from this already created ESMF\_Config object. If specified, takes priority over `configFile`.

**[configFile]** Set the configuration filename for this ESMF\_CplComp. An ESMF\_Config object will be created for this file and attached to the ESMF\_CplComp. Superceded by `config` if both are specified.

**[clock]** Set the private clock for this ESMF\_CplComp.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 15.4.11 ESMF\_CplCompValidate – Ensure the CplComp is internally consistent

##### INTERFACE:

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

##### ARGUMENTS:

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

##### DESCRIPTION:

Currently all this method does is to check that the `cplcomp` exists.

The arguments are:

**cplcomp** ESMF\_CplComp to validate.

**[options]** Validation options are not yet supported.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 15.4.12 ESMF\_CplCompWriteRestart – Call the CplComp’s checkpoint routine

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Call the associated user checkpoint code for an ESMF\_CplComp.

The arguments are:

**cplcomp** ESMF\_CplComp to call writerestart routine for.

**[iospec]** ESMF\_IOSpec object which describes I/O options.

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

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

**[blockingflag]** Valid values are ESMF\_BLOCKING (the default) or ESMF\_NONBLOCKING.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 15.4.13 ESMF\_CplCompWait - Wait for a CplComp to return

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

When executing asynchronously, wait for an ESMF\_CplComp to return.

The arguments are:

**cplcomp** ESMF\_CplComp to wait for.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

## 15.5 Class API: SetServices and Related Methods

### 15.5.1 ESMF\_CplCompGetInternalState - Get private data block pointer

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

The arguments are:

**cplcomp** An `ESMF_CplComp` object.

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

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

---

### 15.5.2 ESMF\_CplCompSetEntryPoint - Set name of CplComp subroutines

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Intended to be called by an `ESMF_CplComp` during the registration process. An `ESMF_CplComp` calls `ESMF_CplCompSetEntryPoint` for each of the predefined `init`, `run`, and `finalize` routines, to associate the internal subroutine to be called for each function. If multiple phases for `init`, `run`, or `finalize` are needed, this can be called with phase numbers.

After this subroutine returns, the framework now knows how to call the initialize, run, and finalize routines for this child `ESMF_CplComp`.

The arguments are:

**cplcomp** An ESMF\_CplComp object.

**subroutineType** One of a set of predefined subroutine types - e.g. ESMF\_SETINIT, ESMF\_SETRUN, ESMF\_SETFINAL.

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

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 15.5.3 ESMF\_CplCompSetInternalState - Set private data block pointer

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

**cplcomp** An ESMF\_CplComp object.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 15.5.4 ESMF\_CplCompSetServices - Register CplComp interface routines

INTERFACE:

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

ARGUMENTS:

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

## DESCRIPTION:

Call an `ESMF_CplComp`'s `setservices` registration routine. The parent component must first create an `ESMF_CplComp`, then call this routine. The arguments are the object returned from the create call, plus the public, well-known, subroutine name that is the registration routine for this `ESMF_CplComp`. This name must be documented by the `ESMF_CplComp` provider.

After this subroutine returns the framework now knows how to call the initialize, run, and finalize routines for the `ESMF_CplComp`.

The arguments are:

**cplcomp** An `ESMF_CplComp` object.

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

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

## 16 State Class

### 16.1 Description

A State contains the data and metadata to be transferred between ESMF components. It is an important class, because it defines a standard for how data is represented in Earth science components. The State construct is a rational compromise between a fully prescribed interface - one that would dictate what specific fields should be transferred between components - and an interface in which data structures are completely ad hoc.

There are two types of States, import and export. An import State contains data that is necessary for a Gridded Component or Coupler Component to execute, and an export State contains the data that a Gridded Component or Coupler Component can make available.

States can contain Bundles, Fields, Arrays, and other States. They cannot contain Fortran arrays.

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

### 16.2 State Options

#### 16.2.1 ESMF\_StateItemType

##### DESCRIPTION:

Specifies the type of object being added to or retrieved from an `ESMF_State`.

Valid values are:

**ESMF\_STATEITEM\_BUNDLE** Refers to an `ESMF_Bundle` within an `ESMF_State`.

**ESMF\_STATEITEM\_FIELD** Refers to an `ESMF_Field` within an `ESMF_State`.

**ESMF\_STATEITEM\_ARRAY** Refers to an `ESMF_Array` within an `ESMF_State`.

**ESMF\_STATEITEM\_STATE** Refers to an `ESMF_State` within an `ESMF_State`.

**ESMF\_STATEITEM\_NAME** Refers to a data name used as a placeholder within an `ESMF_State`.

**ESMF\_STATEITEM\_UNKNOWN** Object type within an `ESMF_State` is unknown.

#### 16.2.2 ESMF\_StateType

##### DESCRIPTION:

Specifies whether an `ESMF_State` contains data to be imported into a component or exported from a component.

Valid values are:

**ESMF\_STATE\_IMPORT** Contains data to be imported into a component.

**ESMF\_STATE\_EXPORT** Contains data to be exported out of a component.

**ESMF\_STATE\_INVALID** Does not contain valid data.

### 16.3 Use and Examples

In ESMF applications States are created at the same time as the components that they will be associated with. A Gridded Component generally has one associated import State and one export State. In most cases the States associated with a Gridded Component will be created by the Gridded Component's parent component with no data buffers yet attached. Both the incomplete States and the pointer to the newly created Gridded Component are passed by the parent component into the Gridded Component's initialize method. This is where the States get prepared for use and the import State is first filled with data.

States can be created without the Fields, Arrays, Bundles, and other States they will eventually contain in a number of ways. They can be created with names as placeholders where these data items will eventually be. When the States are passed into the Gridded Component's initialize method, Field, Bundle, and Array create calls can be made in that method to replace the name placeholders with real data objects.

States can also be filled with data items that do not yet have data allocated. Fields, Bundles, and Arrays each have methods that support their creation without actual data allocation - the grid and metadata are set up but no Fortran array of data values is allocated. In this approach, when a State is passed into its associated Gridded component's initialize method, the incomplete Arrays, Fields, and Bundles within the State can allocate or reference data inside the initialize method.

States are passed through the interfaces of the Gridded and Coupler Components' run methods in order to carry data between the components. While we expect a Gridded Component's import State to be filled with data during initialization, its export State will typically be filled over the course of its run method. At the end of a Gridded Component's run method, the filled export State is passed out through the argument list into a Coupler Component's run method. We recommend the convention that it enters the Coupler Component as the Coupler Component's import State. Here it is transformed into a form that another Gridded Component requires, and passed out of the Coupler Component as its export State. It can then be passed into the run method of a recipient Gridded Component as that component's import State.

While the above sounds complicated, the rule is simple: a State going into a component is an import State, and a State leaving a component is an export State.

Data items within a State can be marked needed or not needed, depending on whether they are required for a particular application configuration. If the item is marked not needed, the user can make the Gridded Component's initialize method clever enough to not allocate the data for that item at all and not compute it within the Gridded Component code. For example, some diagnostics may not be desired for all runs.

Other flags will eventually be available for data items within a State, such as data ready for reading or writing, data valid or invalid, and data required for restart or not. These are not yet fully implemented, so only the default value for each value can be set at this time.

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

! ESMF Framework module
use ESMF_Mod
implicit none

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

```

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

```

### 16.3.1 Empty State Create

Creation of an empty ESMF\_State, which will be added to later.

```

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

```

### 16.3.2 Adding Items to a State

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

```

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

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

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

call ESMF_StateDestroy(state2, rc)

call ESMF_BundleDestroy(bundle1, rc)

```

### 16.3.3 Adding Placeholders to a State

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

```

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

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

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

```

### 16.3.4 Marking an Item Needed

How to set the NEEDED state of an item.

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

### 16.3.5 Creating a Needed Item

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

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

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

    call ESMF_StateAddBundle(state3, bundle2, rc)

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

## 16.4 Restrictions and Future Work

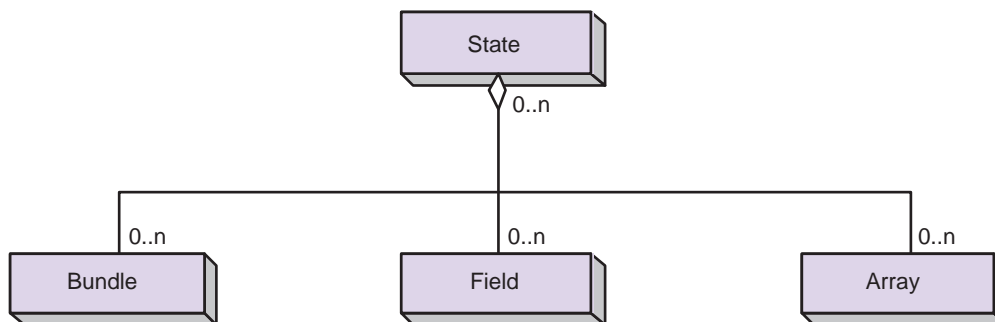
1. **Flags not fully implemented.** The flags for indicating various qualities associated with data items in a State - validity, whether or not the item is required for restart, read/write status - are not fully implemented. Although their defaults can be set, the associated methods for setting and getting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

## 16.5 Design and Implementation Notes

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

## 16.6 Object Model

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



## 16.7 Class API: Basic State Methods

### 16.7.1 ESMF\_StateAddArray - Add an Array to a State

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

**state** An ESMF\_State object.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.2 ESMF\_StateAddArray - Add a list of Arrays to a State

INTERFACE:

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

*ARGUMENTS:*

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

**DESCRIPTION:**

Add multiple ESMF\_Arrays to an ESMF\_State.

The arguments are:

**state** An ESMF\_State object.

**arrayCount** The number of ESMF\_Arrays to be added.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.3 ESMF\_StateAddAttribute - Add an integer attribute

**INTERFACE:**

```
! Private name; call using ESMF_StateAddAttribute()
subroutine ESMF_StateAddIntAttr(state, name, value, rc)
```

*ARGUMENTS:*

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

**DESCRIPTION:**

Attaches an integer attribute to the state. The attribute has a name and a value.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to add.

**value** The integer value of the attribute to add.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 16.7.4 ESMF\_StateAddAttribute - Add an integer list attribute

##### INTERFACE:

```
! Private name; call using ESMF_StateAddAttribute()  
subroutine ESMF_StateAddIntListAttr(state, name, count, valueList, rc)
```

##### ARGUMENTS:

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

##### DESCRIPTION:

Attaches an integer list attribute to the state. The attribute has a name and a valueList. The number of integer items in the valueList is given by count.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to add.

**count** The number of integers in the valueList.

**valueList** The integer values of the attribute to add.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 16.7.5 ESMF\_StateAddAttribute - Add a real attribute

##### INTERFACE:

```
! Private name; call using ESMF_StateAddAttribute()  
subroutine ESMF_StateAddRealAttr(state, name, value, rc)
```

##### ARGUMENTS:

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

##### DESCRIPTION:

Attaches a real attribute to the state. The attribute has a name and a value.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to add.

**value** The real value of the attribute to add.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.6 ESMF\_StateAddAttribute - Add a real list attribute

#### INTERFACE:

```
! Private name; call using ESMF_StateAddAttribute()  
subroutine ESMF_StateAddRealListAttr(state, name, count, valueList, rc)
```

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to add.

**count** The number of reals in the `valueList`.

**value** The real values of the attribute to add.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.7 ESMF\_StateAddAttribute - Add a logical attribute

#### INTERFACE:

```
! Private name; call using ESMF_StateAddAttribute()  
subroutine ESMF_StateAddLogicalAttr(state, name, value, rc)
```

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to add.

**value** The logical true/false value of the attribute to add.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.8 ESMF\_StateAddAttribute - Add a logical list attribute

#### INTERFACE:

```
! Private name; call using ESMF_StateAddAttribute()  
subroutine ESMF_StateAddLogicalListAttr(state, name, count, valueList, rc)
```

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

**state** An `ESMF_State` object.

**name** The name of the attribute to add.

**count** The number of logicals in the `valueList`.

**valueList** The logical true/false values of the attribute.

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

---

### 16.7.9 ESMF\_StateAddAttribute - Add a character attribute

#### INTERFACE:

```
! Private name; call using ESMF_StateAddAttribute()  
subroutine ESMF_StateAddCharAttr(state, name, value, rc)
```

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

**state** An `ESMF_State` object.

**name** The name of the attribute to add.

**value** The character value of the attribute to add.

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

---

### 16.7.10 ESMF\_StateAddBundle - Add a Bundle to a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

**state** The ESMF\_State object.

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

[rc] Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.11 ESMF\_StateAddBundle - Add a list of Bundles to a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Add multiple ESMF\_Bundles to an ESMF\_State. The arguments are:

**state** An ESMF\_State object.

**bundleCount** The number of ESMF\_Bundles to be added.

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

[rc] Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.12 ESMF\_StateAddField - Add a Field to a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

**state** An ESMF\_State object.

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

[rc] Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.13 ESMF\_StateAddField - Add a list of Fields to a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Add multiple ESMF\_Fields to an ESMF\_State. The arguments are:

**state** An ESMF\_State object.

**fieldCount** The number of ESMF\_Fields to be added.

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

[rc] Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 16.7.14 ESMF\_StateAddNameOnly - Add a name to a State as a placeholder

##### INTERFACE:

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

##### ARGUMENTS:

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

##### DESCRIPTION:

Add the character string name to an existing state. It can subsequently be replaced by an actual object with the same name. The name must be unique within the state. It is available to be marked needed by the consumer of the export ESMF\_State. Then the data provider can replace the name with the actual ESMF\_Bundle, ESMF\_Field, or ESMF\_Array which carries the needed data.

The arguments are:

**state** An ESMF\_State object.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

#### 16.7.15 ESMF\_StateAddNameOnly - Add a list of names to a State

##### INTERFACE:

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

##### ARGUMENTS:

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

##### DESCRIPTION:

Add a list of names to an existing state. They can subsequently be replaced by actual objects with the same name. Each name in the nameList must be unique within the state. It is available to be marked needed by the consumer of the export ESMF\_State. Then the data provider can replace the name with the actual ESMF\_Bundle, ESMF\_Field, or ESMF\_Array which carries the needed data. Unneeded data need not be generated.

The arguments are:

**state** An ESMF\_State object.

**nameCount** The count of names in the nameList.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.16 ESMF\_StateAddState - Add a State to a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

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

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

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

---

### 16.7.17 ESMF\_StateAddState - Add a list of States to a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

The arguments are:

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

**nestedStateCount** The number of `ESMF_States` to be added.

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

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

---

## 16.7.18 ESMF\_StateCreate - Create a new State

### INTERFACE:

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

### RETURN VALUE:

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

### ARGUMENTS:

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

### DESCRIPTION:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

### 16.7.19 ESMF\_StateDestroy - Release resources for a State

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

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

The arguments are:

**state** Destroy contents of this `ESMF_State`.

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

---

### 16.7.20 ESMF\_StateGet - Get information about a State

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Returns the requested information about this `ESMF_State`.

The arguments are:

**state** An ESMF\_State object to be queried.

**[name]** Name of this ESMF\_State.

**[statetype]** Import or Export ESMF\_State. Possible values are listed in Section 16.2.2.

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

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

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.21 ESMF\_StateGetArray - Retrieve a data Array from a State

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Returns an ESMF\_Array from an ESMF\_State by name. If the ESMF\_State contains the object directly, only *arrayName* is required. If the *state* contains multiple nested ESMF\_States and the object is one level down, this routine can return the object in a single call by specifying the proper *nestedStateName*. ESMF\_States can be nested to any depth, but this routine only searches in immediate descendents. It is an error to specify a *nestedStateName* if the *state* contains no nested ESMF\_States.

The arguments are:

**state** State to query for an ESMF\_Array named *arrayName*.

**arrayName** Name of ESMF\_Array to be returned.

**array** Returned reference to the ESMF\_Array.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.22 ESMF\_StateGetAttribute - Retrieve an integer attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns an integer attribute from the *state*.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**value** The integer value of the named attribute.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.23 ESMF\_StateGetAttribute - Retrieve an integer list attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns an integer list attribute from the *state*.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**count** The number of values in the attribute.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.24 ESMF\_StateGetAttribute - Retrieve a real attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns a real attribute from the `state`.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**value** The real value of the named attribute.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.25 ESMF\_StateGetAttribute - Retrieve a real list attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns a list of real attributes from the `state`.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**count** The number of values in the attribute.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.26 ESMF\_StateGetAttribute - Retrieve a logical attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns a logical attribute from the state.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**value** The logical value of the named attribute.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.27 ESMF\_StateGetAttribute - Retrieve a logical list attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns a logical list attribute from the state.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**count** The number of values in the attribute.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.28 ESMF\_StateGetAttribute - Retrieve a character attribute

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns a character attribute from the state.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to retrieve.

**value** The character value of the named attribute.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.29 ESMF\_StateGetAttributeCount - Query the number of attributes

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns the number of attributes associated with the given state in the argument count.

The arguments are:

**state** An ESMF\_State object.

**count** The number of attributes associated with this object.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.30 ESMF\_StateGetAttributeInfo - Query State attributes by name

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns information associated with the named attribute, including datatype and count.

The arguments are:

**state** An ESMF\_State object.

**name** The name of the attribute to query.

**datatype** The datatype of the attribute.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.31 ESMF\_StateGetAttributeInfo - Query State attributes by index number

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns information about an attribute by specifying an ordinal index number. Attributes with unknown names can be queried using this routine.

The arguments are:

**state** An ESMF\_State object.

**attributeIndex** The index number of the attribute to query.

**name** Returns the name of the attribute.

**datatype** Returns the datatype of the attribute.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.32 ESMF\_StateGetBundle - Retrieve a Bundle from a State

INTERFACE:

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

ARGUMENTS:

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

DESCRIPTION:

Returns an ESMF\_Bundle from an ESMF\_State by name. If the ESMF\_State contains the object directly, only bundleName is required. If the state contains multiple nested ESMF\_States and the object is one level down, this routine can return the object in a single call by specifying the proper nestedStateName. ESMF\_States can be nested to any depth, but this routine only searches in immediate descendents. It is an error to specify a nestedStateName if the state contains no nested ESMF\_States.

The arguments are:

**state** State to query for a ESMF\_Bundle named bundleName.

**bundleName** Name of ESMF\_Bundle to be returned.

**bundle** Returned reference to the ESMF\_Bundle.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.33 ESMF\_StateGetField - Retrieve a Field from a State

INTERFACE:

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

ARGUMENTS:

```

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

```

#### DESCRIPTION:

Returns an ESMF\_Field from an ESMF\_State by name. If the ESMF\_State contains the object directly, only fieldName is required. If the state contains multiple nested ESMF\_States and the object is one level down, this routine can return the object in a single call by specifying the proper nestedStateName. ESMF\_States can be nested to any depth, but this routine only searches in immediate descendents. It is an error to specify a nestedStateName if the state contains no nested ESMF\_States.

The arguments are:

**state** State to query for an ESMF\_Field named fieldName.

**fieldName** Name of ESMF\_Field to be returned.

**field** Returned reference to the ESMF\_Field.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

### 16.7.34 ESMF\_StateGetNeeded - Query whether a data item is needed

#### INTERFACE:

```

subroutine ESMF_StateGetNeeded(state, itemName, neededflag, rc)

```

#### ARGUMENTS:

```

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

```

#### DESCRIPTION:

Returns the status of the neededflag for the data item named by itemName in the ESMF\_State.

The arguments are:

**state** The ESMF\_State to query.

**itemName** Name of the data item to query.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

### 16.7.35 ESMF\_StateGetState - Retrieve a State nested in a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns a nested ESMF\_State from another ESMF\_State by name. This does not allow the caller to retrieve an ESMF\_State from two levels down. It returns immediate child objects only.

The arguments are:

**state** The ESMF\_State to query for a nested ESMF\_State named stateName.

**nestedStateName** Name of nested ESMF\_State to return.

**nestedState** Returned ESMF\_State.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.36 ESMF\_StateIsNeeded – Return logical true if data item needed

#### INTERFACE:

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

#### RETURN VALUE:

```
logical :: ESMF_StateIsNeeded
```

#### ARGUMENTS:

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

#### DESCRIPTION:

Returns true if the status of the needed flag for the data item named by itemName in the ESMF\_State is ESMF\_STATEITEM\_NEEDED. Returns false for no item found with the specified name or item marked not needed.

Also sets error code if dataname not found.

The arguments are:

**state** ESMF\_State to query.

**itemName** Name of the data item to query.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.37 ESMF\_StatePrint - Print the internal data for a State

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

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

**state** The ESMF\_State to print.

**[options]** Print options are not yet supported.

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---

### 16.7.38 ESMF\_StateSetNeeded - Set if a data item is needed

#### INTERFACE:

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

#### ARGUMENTS:

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

#### DESCRIPTION:

Sets the status of the needed flag for the data item named by itemName in the ESMF\_State.  
The arguments are:

**state** The ESMF\_State to set.

**itemName** Name of the data item to set.

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

**[rc]** Return code; equals ESMF\_SUCCESS if there are no errors.

---