Earth System Modeling Framework

ESMF Reference Manual for Fortran

Version 8.3.0 beta snapshot

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

April 19, 2022

http://www.earthsystemmodeling.org

Acknowledgements

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

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

Contents

Part I

ESMF Overview

1 What is the Earth System Modeling Framework?

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

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

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

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

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

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

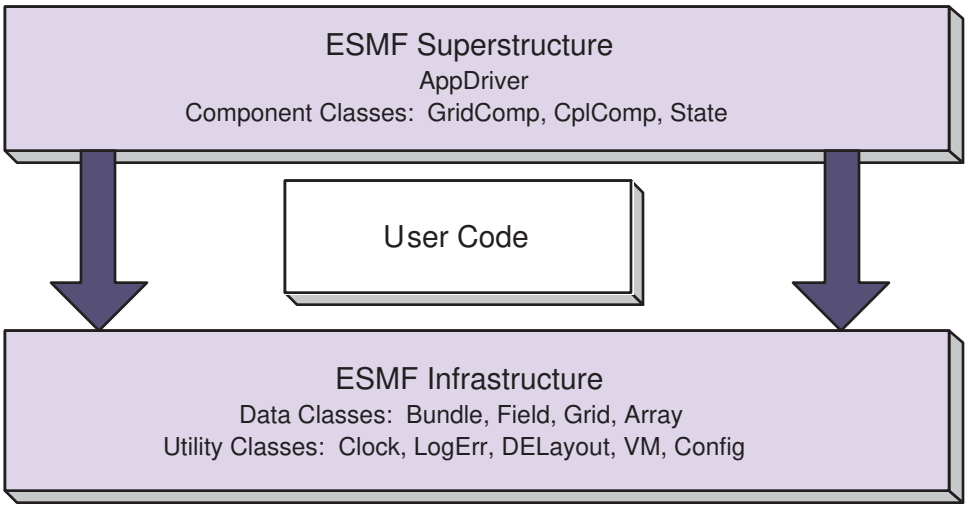
2 The ESMF Reference Manual for Fortran

ESMF has a complete set of Fortran interfaces and some C interfaces. This *ESMF Reference Manual* is a listing of ESMF interfaces for Fortran.[1]

Interfaces are grouped by class. A class is comprised of the data and methods for a specific concept like a physical field. Superstructure classes are listed first in this *Manual*, followed by infrastructure classes.

[1] Since the customer base for it is small, we have not yet prepared a comprehensive reference manual for C.

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



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

3 How to Contact User Support and Find Additional Information

The ESMF team can answer questions about the interfaces presented in this document. For user support, please contact esmf_support@ucar.edu.

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

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

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

5 Conventions

5.1 Typeface and Diagram Conventions

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

Style	Meaning	Example
italics	documents	*ESMF Reference Manual*
`courier`	code fragments	`ESMF_TRUE`
`courier()`	ESMF method name	`ESMF_FieldGet()`
boldface	first definitions	An **address space** is ...
boldface	web links and tabs	**Developers** tab on the website
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.* [**?**]

5.2 Method Name and Argument Conventions

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

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

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

A typical ESMF call looks like this:

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

where
`<ClassName>` is the class name,
`<Operation>` is the name of the action to be performed,
`classname` is a variable of the derived type associated with the class,
the `arg*` arguments are whatever other variables are required for the operation,
and `rc` is a return code.

6 The ESMF Application Programming Interface

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

The 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`.

6.1 Standard Methods and Interface Rules

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

- `ESMF_<Class>Create()` and `ESMF_<Class>Destroy()`, for creating and destroying objects of ESMF classes that require internal memory management (- called ESMF deep classes). The `ESMF_<Class>Create()` method allocates memory for the object itself and for internal variables, and initializes variables where appropriate. It is always written as a Fortran function that returns a derived type instance of the class, i.e. an object.

- `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>AddReplace()`, `ESMF_<Class>Remove()`, and `ESMF_<Class>Replace()`, for manipulating objects of ESMF container classes - such as `ESMF_State` and `ESMF_FieldBundle`. For example, the `ESMF_FieldBundleAdd()` method adds another Field to an existing FieldBundle object.

- `ESMF_<Class>Print()`, for printing the contents of an object 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()` validates the internal consistency of a Field object.

6.2 Deep and Shallow Classes

The ESMF contains two types of classes.

Deep classes require ESMF_<Class>Create() and ESMF_<Class>Destroy() calls. They involve memory allocation, take significant time to set up (due to memory management) and should not be created in a time-critical portion of code. Deep objects persist even after the method in which they were created has returned. Most classes in ESMF, including GridComp, CplComp, State, Fields, FieldBundles, Arrays, ArrayBundles, Grids, and Clocks, fall into this category.

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

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

See Section 9, Overall Design and Implementation Notes, for a brief discussion of deep and shallow classes from an implementation perspective. For an in-depth look at the design and inter-language issues related to deep and shallow classes, see the *ESMF Implementation Report*.

6.3 Special Methods

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

- ESMF_Initialize() and ESMF_Finalize() are required methods that must bracket the use of ESMF within an application. They manage the resources required to run ESMF and shut it down gracefully. ESMF does not support restarts in the same executable, i.e. ESMF_Initialize() should not be called after ESMF_Finalize().
- 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 subroutines within user code.

6.4 The ESMF Data Hierarchy

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

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

- **Array** An ESMF Array is a distributed, multi-dimensional array that can carry information such as its type, kind, rank, and associated halo widths. It contains a reference to a native Fortran array.
- **ArrayBundle** An ArrayBundle is a collection of Arrays, not necessarily distributed in the same manner. It is useful for performing collective data operations and communications.
- **Field** A Field represents a physical scalar or vector field. It contains a reference to an Array along with grid information and metadata.

- **FieldBundle** A FieldBundle is a collection of Fields discretized on the same grid. The staggering of data points may be different for different Fields within a FieldBundle. Like the ArrayBundle, it is useful for performing collective data operations and communications.
- **State** A State represents the collection of data that a Component either requires to run (an Import State) or can make available to other Components (an Export State). States may contain references to Arrays, ArrayBundles, Fields, FieldBundles, or other States.
- **Component** A Component is a piece of software with a distinct function. ESMF currently recognizes two types of Components. Components that represent a physical domain or process, such as an atmospheric model, are called Gridded Components since they are usually discretized on an underlying grid. The Components responsible for regridding and transferring data between Gridded Components are called Coupler Components. Each Component is associated with an Import and an Export State. Components can be nested so that simpler Components are contained within more complex ones.

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

6.5 ESMF Spatial Classes

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

- The **Virtual Machine**, or **VM** The ESMF VM is an abstraction of a parallel computing environment that encompasses both shared and distributed memory, single and multi-core systems. Its primary purpose is resource allocation and management. Each Component runs in its own VM, using the resources it defines. The elements of a VM are **Persistent Execution Threads**, or **PETs**, that are executing in **Virtual Address Spaces**, or **VASs**. A simple case is one in which every PET is associated with a single MPI process. In this case every PET is executing in its own private VAS. If Components are nested, the parent component allocates a subset of its PETs to its children. The children have some flexibility, subject to the constraints of the computing environment, to decide how they want to use the resources associated with the PETs they've received.
- **DELayout** A DELayout represents a data decomposition (we also refer to this as a distribution). Its basic elements are **Decomposition Elements**, or **DEs**. A DELayout associates a set of DEs with the PETs in a VM. DEs are not necessarily one-to-one with PETs. For cache blocking, or user-managed multi-threading, more DEs than PETs may be defined. Fewer DEs than PETs may also be defined if an application requires it.
- **DistGrid** A DistGrid represents the index space associated with a grid. It is a useful abstraction because often a full specification of grid coordinates is not necessary to define data communication patterns. The DistGrid contains information about the sequence and connectivity of data points, which is sufficient information for many operations. Arrays are defined on DistGrids.
- **Array** An Array defines how the index space described in the DistGrid is associated with the VAS of each PET. This association considers the type, kind and rank of the indexed data. Fields are defined on Arrays.
- **Grid** A Grid is an abstraction for a logically rectangular region in physical space. It associates a coordinate system, a set of coordinates, and a topology to a collection of grid cells. Grids in ESMF are comprised of DistGrids plus additional coordinate information.

- **Mesh** A Mesh provides an abstraction for an unstructured grid. Coordinate information is set in nodes, which represent vertices or corners. Together the nodes establish the boundaries of mesh elements or cells.
- **LocStream** A LocStream is an abstraction for a set of unstructured data points without any topological relationship to each other.
- **Field** A Field may contain more dimensions than the Grid that it is discretized on. For example, for convenience during integration, a user may want to define a single Field object that holds snapshots of data at multiple times. Fields also keep track of the stagger location of a Field data point within its associated Grid cell.

6.6 ESMF Maps

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

6.7 ESMF Specification Classes

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

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

6.8 ESMF Utility Classes

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

- **Attributes**, for storing metadata about Fields, FieldBundles, States, and other classes.
- **TimeMgr**, for calendar, time, clock and alarm functions.
- **LogErr**, for logging and error handling.
- **Config**, for creating resource files that can replace namelists as a consistent way of setting configuration parameters.

7 Integrating ESMF into Applications

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

In the bottom-up approach, the user selects desired utilities (data communications, calendar management, performance profiling, logging and error handling, etc.) from the ESMF infrastructure and either writes new code using them, introduces them into existing code, or replaces the functionality in existing code with them. This makes sense when maximizing code reuse and minimizing maintenance costs is a goal. There may be a specific need for functionality or the component writer may be starting from scratch. The calendar management utility is a popular place to start.

7.1 Using the ESMF Superstructure

The following is a typical set of steps involved in adopting the ESMF superstructure. The first two tasks, which occur before an ESMF call is ever made, have the potential to be the most difficult and time-consuming. They are the work of splitting an application into components and ensuring that each component has well-defined stages of execution. ESMF aside, this sort of code structure helps to promote application clarity and maintainability, and the effort put into it is likely to be a good investment.

1. Decide how to organize the application as discrete Gridded and Coupler Components. This might involve reorganizing code so that individual components are cleanly separated and their interactions consist of a minimal number of data exchanges.
2. Divide the code for each component into initialize, run, and finalize methods. These methods can be multi-phase, e.g., `init_1, init_2`.
3. Pack any data that will be transferred between components into ESMF Import and Export State data structures. This is done by first wrapping model data in either ESMF Arrays or Fields. Arrays are simpler to create and use than Fields, but carry less information and have a more limited range of operations. These Arrays and Fields are then added to Import and Export States. They may be packed into ArrayBundles or FieldBundles first, for more efficient communications. Metadata describing the model data can also be added. At the end of this step, the data to be transferred between components will be in a compact and largely self-describing form.
4. Pack time information into ESMF time management data structures.
5. Using code templates provided in the ESMF distribution, create ESMF Gridded and Coupler Components to represent each component in the user code.
6. Write a set services routine that sets ESMF entry points for each user component's initialize, run, and finalize methods.
7. Run the application using an ESMF Application Driver.

8 Overall Rules and Behavior

8.1 Return Code Handling

All ESMF methods pass a *return code* back to the caller via the `rc` argument. If no errors are encountered during the method execution, a value of `ESMF_SUCCESS` is returned. Otherwise one of the predefined error codes is returned to the caller. See the appendix, section **??**, for a full list of the ESMF error return codes.

Any code calling an ESMF method must check the return code. If `rc` is not equal to `ESMF_SUCCESS`, the calling code is expected to break out of its execution and pass the `rc` to the next level up. All ESMF errors are to be handled as *fatal*, i.e. the calling code must *bail-on-all-errors*.

ESMF provides a number of methods, described under section **??**, that make implementation of the bail-on-all-errors stategy more convenient. Consistent use of these methods will ensure that a full back trace is generated in the ESMF log output whenever an error condition is triggered.

Note that in ESMF requesting not present information, e.g. via a `Get()` method, will trigger an error condition. Combined with the bail-on-all-errors strategy this has the advantage of producing an error trace pointing to the earliest location in the code that attempts to access unavailable information. In cases where the calling side is able to handle the presence or absence of certain pieces of of information, the code first must query for the resepctive `isPresent` argument. If this argument comes back as `.true.` it is safe to query for the actual information.

8.2 Local and Global Views and Associated Conventions

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

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

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

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

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

8.4 Assignment, Equality, Copying and Comparing Objects

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

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

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

8.5 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 integer, real, double precision, logical, or character values. Attributes can be associated with Fields, FieldBundles, 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.

8.6 Constants

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

```
ESMF_<CONSTANT_NAME>_Flag
```

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

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

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

9 Overall Design and Implementation Notes

1. **Deep and shallow classes.** The deep and shallow classes described in Section 6.2 differ in how and where they are allocated within a multi-language implementation environment. We distinguish between the implementation language, which is the language a method is written in, and the calling language, which is the language that the user application is written in. Deep classes are allocated off the process heap by the implementation language. Shallow classes are allocated off the stack by the calling language.
2. **Base class.** All ESMF classes are built upon a Base class, which holds a small set of system-wide capabilities.

10 Overall Restrictions and Future Work

1. **32-bit integer limitations.** In general, Fortran array bounds should be limited to 2**31-1 elements or less.

This is due to the Fortran-95 limitation of returning default sized (e.g., 32 bit) integers for array bound and size inquiries, and consequent ESMF use of default sized integers for holding these values.

Part II

Command Line Tools

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

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

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

11 ESMF_PrintInfo

11.1 Description

The `ESMF_PrintInfo` command line tool that prints basic information about the ESMF installation to `stdout`.

The command line tool usage is as follows:

```
ESMF_PrintInfo  [--help]

where
  --help     prints a brief usage message


‘
```

12 ESMF_RegridWeightGen

12.1 Description

This section describes the offline regrid weight generation application provided by ESMF (for a description of ESMF regridding in general see Section 24.2). Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Regridding can be broken into two stages. The first stage is generation of an interpolation weight matrix that describes how points in the source grid contribute to points in the destination grid. The second stage is the multiplication of values on the source grid by the interpolation weight matrix to produce values on the destination grid. This is implemented as a parallel sparse matrix multiplication.

There are two options for accessing ESMF regridding functionality: integrated and offline. Integrated regridding is a process whereby interpolation weights are generated via subroutine calls during the execution of the user's code. The integrated regridding can also perform the parallel sparse matrix multiplication. In other words, ESMF integrated regridding allows a user to perform the whole process of interpolation within their code. For a further description of ESMF integrated regridding please see Section **??**. In contrast to integrated regridding, offline regridding is a process whereby interpolation weights are generated by a separate ESMF command line tool, not within the user code. The ESMF offline regridding tool also only generates the interpolation matrix, the user is responsible for reading in this matrix and doing the actual interpolation (multiplication by the sparse matrix) in their code. The rest of this section further describes ESMF offline regridding.

For a discussion of installing and accessing ESMF command line tools such as this one please see the beginning of this part of the reference manual (Section II) or for the quickest approach to just building and accessing the command line tools please refer to the "Building and using bundled ESMF Command Line Tools" Section in the ESMF User's Guide.

This application requires the NetCDF library to read the grid files and to write out the weight files in NetCDF format. To compile ESMF with the NetCDF library, please refer to the "Third Party Libraries" Section in the ESMF User's Guide for more information.

As described above, this tool reads in two grid files and outputs weights for interpolation between the two grids. The input and output files are all in NetCDF format. The grid files can be defined in five different formats: the SCRIP format 12.8.1 as is used as an input to SCRIP [**?**], the CF convension single-tile grid file 12.8.3 following the CF metadata conventions, the GRIDSPEC Mosaic file 12.8.5 following the proposed GRIDSPEC standard, the ESMF unstructured grid format 12.8.2 or the proposed CF unstructured grid data model (UGRID) 12.8.4. GRIDSPEC is a proposed CF extension for the annotation of complex Earth system grids. In the latest ESMF library, we added support for multi-tile GRIDSPEC Mosaic file with non-overlapping tiles. For UGRID, we support the 2D flexible mesh topology with mixed triangles and quadrilaterals and fully 3D unstructured mesh topology with hexahedrons and tetrahedrons.

In the latest ESMF implementation, the `ESMF_RegridWeightGen` command line toll can detect the type of the input grid files automatically. The user doesn't need to provide the source and destination grid file type arguments anymore. The following arguments `-t`, `-src_type`, `-dst_type`, `-src_meshname`, and `-dst_meshname` are no longer needed. If provided, the application will simply ingore them.

This command line tool can do regrid weight generation from a global or regional source grid to a global or regional destination grid. As is true with many global models, this application currently assumes the latitude and longitude values refer to positions on a perfect sphere, as opposed to a more complex and accurate representation of the Earth's true shape such as would be used in a GIS system. (ESMF's current user base doesn't require this level of detail in representing the Earth's shape, but it could be added in the future if necessary.)

The interpolation weights generated by this application are output to a NetCDF file (specified by the "-w" or "`--`weight" keywords). Two type of weight files are supported: the SCRIP format is the same as that generated by SCRIP, see Section 12.9 for a description of the format; and a simple weight file containing only the weights and the source and destination grid indices (In ESMF term, these are the `factorList` and `factorIndexList` generated by the ESMF weight calculation function `ESMF_FieldRegridStore()`. Note that the sequence of the weights in the file can vary with the number of processors used to run the application. This means that two weight files generated by using different numbers of processors can contain exactly the same interpolation matrix, but can appear different in a direct line by line comparison (such as would be done by ncdiff). The interpolation weights can be generated with the bilinear, patch, nearest neighbor, first-order conservative, or second-order conservative methods

described in Section 12.3.

Internally this application uses the ESMF public API to generate the interpolation weights. If a source or destination grid is a single tile logically rectangular grid, then `ESMF_GridCreate()` **??** is used to create an ESMF_Grid object. The cell center coordinates of the input grid are put into the center stagger location (`ESMF_STAGGERLOC_CENTER`). In addition, the corner coordinates are also put into the corner stagger location (`ESMF_STAGGERLOC_CORNER`) for conservative regridding. If a grid contains multiple logically rectangular tiles connected with each other by edges, such as a Cubed Sphere grid, the grid can be represented as a multi-tile ESMF_Grid object created using `ESMF_GridCreateMosaic()` **??**. Such a grid is stored in the GRIDSPEC Mosaic and tile file format. 12.8.5 The method `ESMF_MeshCreate()` **??** is used to create an ESMF_Mesh object, if the source or destination grid is an unstructured grid. When making this call, the flag `convert3D` is set to `TRUE` to convert the 2D coordinates into 3D Cartesian coordinates. Internally `ESMF_FieldRegridStore()` is used to generate the weight table and indices table representing the interpolation matrix.

12.2 Regridding Options

The offline regrid weight generation application supports most of the options available in the rest of the ESMF regrid system. The following is a description of these options as relevant to the application. For a more in-depth description see Section 24.2.

12.2.1 Poles

The regridding occurs in 3D to avoid problems with periodicity and with the pole singularity. This application supports four options for handling the pole region (i.e. the empty area above the top row of the source grid or below the bottom row of the source grid). Note that all of these pole options currently only work for logically rectangular grids (i.e. SCRIP format grids with grid_rank=2 or GRIDSPEC single-tile format grids). The first option is to leave the pole region empty ("-p none"), in this case if a destination point lies above or below the top row of the source grid, it will fail to map, yielding an error (unless "-i" is specified). With the next two options, the pole region is handled by constructing an artificial pole in the center of the top and bottom row of grid points and then filling in the region from this pole to the edges of the source grid with triangles. The pole is located at the average of the position of the points surrounding it, but moved outward to be at the same radius as the rest of the points in the grid. The difference between these two artificial pole options is what value is used at the pole. The default pole option ("-p all") sets the value at the pole to be the average of the values of all of the grid points surrounding the pole. For the other option ("-p N"), the user chooses a number N from 1 to the number of source grid points around the pole. For each destination point, the value at the pole is then the average of the N source points surrounding that destination point. For the last pole option ("-p teeth") no artificial pole is constructed, instead the pole region is covered by connecting points across the top and bottom row of the source Grid into triangles. As this makes the top and bottom of the source sphere flat, for a big enough difference between the size of the source and destination pole regions, this can still result in unmapped destination points. Only pole option "none" is currently supported with the conservative interpolation methods (e.g. "-m conserve") and with the nearest neighbor interpolation methods ("-m nearestdtos" and "-m neareststod").

12.2.2 Masking

Masking is supported for both the logically rectangular grids and the unstructured grids. If the grid file is in the SCRIP format, the variable "grid_imask" is used as the mask. If the value is set to 0 for a grid point, then that point is considered masked out and won't be used in the weights generated by the application. If the grid file is in the ESMF format, the variable "element Mask" is used as the mask. For a grid defined in the GRIDSPEC single-tile or multi-tile grid or in the UGRID convention, there is no mask variable defined. However, a GRIDSPEC single-tile file or a UGRID file may contain both the grid definition and the data. The grid mask is usually constructed using the

missing values defined in the data variable. The regridding application provides the argument "−−src_missingvalue" or "−−dst_missingvalue" for users to specify the variable name from where the mask can be constructed.

12.2.3 Extrapolation

The `ESMF_RegridWeightGen` application supports a number of kinds of extrapolation to fill in points not mapped by the regrid method. Please see the sections starting with section 24.2.11 for a description of these methods. When using the application an extrapolation method is specified by using the "−−extrap_method" flag. For the inverse distance weighted average method (nearestidavg), the number of source locations is specified using the "−−extrap_num_src_pnts" flag, and the distance exponent is specified using the "−−extrap_dist_exponent" flag. For the creep fill method (creep), the number of creep levels is specified using the "−−extrap_num_levels" flag.

12.2.4 Unmapped destination points

If a destination point can't be mapped, then the default behavior of the application is to stop with an error. By specifying "-i" or the equivalent "−−ignore_unmapped " the user can cause the application to ignore unmapped destination points. In this case, the output matrix won't contain entries for the unmapped destination points. Note that the unmapped point detection doesn't currently work for nearest destination to source method ("-m nearestdtos"), so when using that method it is as if "-i" is always on.

12.2.5 Line type

Another variation in the regridding supported with spherical grids is **line type**. This is controlled by the "−−line_type" or "-l" flag. This switch allows the user to select the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated, for example in bilinear interpolation the distances are used to calculate the weights and the cell edges are used to determine to which source cell a destination point should be mapped.

ESMF currently supports two line types: "cartesian" and "greatcircle". The "cartesian" option specifies that the line between two points follows a straight path through the 3D Cartesian space in which the sphere is embedded. Distances are measured along this 3D Cartesian line. Under this option cells are approximated by planes in 3D space, and their boundaries are 3D Cartesian lines between their corner points. The "greatcircle" option specifies that the line between two points follows a great circle path along the sphere surface. (A great circle is the shortest path between two points on a sphere.) Distances are measured along the great circle path. Under this option cells are on the sphere surface, and their boundaries are great circle paths between their corner points.

12.3 Regridding Methods

This regridding application can be used to generate bilinear, patch, nearest neighbor, first-order conservative, or second-order conservative interpolation weights. The following is a description of these interpolation methods as relevant to the offline weight generation application. For a more in-depth description see Section 24.2.

12.3.1 Bilinear

The default interpolation method for the weight generation application is bilinear. The algorithm used by this application to generate the bilinear weights is the standard one found in many textbooks. Each destination point is mapped

to a location in the source Mesh, the position of the destination point relative to the source points surrounding it is used to calculate the interpolation weights. A restriction on bilinear interpolation is that ESMF doesn't support self-intersecting cells (e.g. a cell twisted into a bow tie) in the source grid.

12.3.2 Patch

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

The patch interpolation process works as follows. For each source element containing a destination point we construct a patch for each corner node that makes up the element (e.g. 4 patches for quadrilateral elements, 3 for triangular elements). To construct a polynomial patch for a corner node we gather all the elements around that node. (Note that this means that the patch interpolation weights depends on the source element's nodes, and the nodes of all elements neighboring the source element.) We then use a least squares fitting algorithm to choose the set of coefficients for the polynomial that produces the best fit for the data in the elements. This polynomial will give a value at the destination point that fits the source data in the elements surrounding the corner node. We then repeat this process for each corner node of the source element generating a new polynomial for each set of elements. To calculate the value at the destination point we do a weighted average of the values of each of the corner polynomials evaluated at that point. The weight for a corner's polynomial is the bilinear weight of the destination point with regard to that corner.

The patch method has a larger stencil than the bilinear, for this reason the patch weight matrix can be correspondingly larger than the bilinear matrix (e.g. for a quadrilateral grid the patch matrix is around 4x the size of the bilinear matrix). This can be an issue when performing a regrid weight generation operation close to the memory limit on a machine.

The patch method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the patch method, the destination field will contain values less than 0.0.

This method currently doesn't support self-intersecting cells (e.g. a cell twisted into a bow tie) in the source grid.

12.3.3 Nearest neighbor

The nearest neighbor interpolation options work by associating a point in one set with the closest point in another set. If two points are equally close then the point with the smallest index is arbitrarily used (i.e. the point with that would have the smallest index in the weight matrix). There are two versions of this type of interpolation available in the regrid weight generation application. One of these is the nearest source to destination method ("-m neareststod"). In this method each destination point is mapped to the closest source point. The other of these is the nearest destination to source method ("-m nearestdtos"). In this method each source point is mapped to the closest destination point. Note that with this method the unmapped destination point detection doesn't work, so no error will be returned even if there are destination points which don't map to any source point.

12.3.4 First-order conservative

The main purpose of this method is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 12.4.) In this method the value across each source cell is treated as a constant, so it will typically have a larger

interpolation error than the bilinear or patch methods. The first-order method used here is similar to that described in the following paper [**?**].

By default (or if "−−norm_type dstarea"), the weight w_{ij} for a particular source cell i and destination cell j are calculated as $w_{ij} = f_{ij} * A_{si}/A_{dj}$. In this equation f_{ij} is the fraction of the source cell i contributing to destination cell j, and A_{si} and A_{dj} are the areas of the source and destination cells. If "−−norm_type fracarea", then the weights are further divided by the destination fraction. In other words, in that case $w_{ij} = f_{ij} * A_{si}/(A_{dj} * D_j)$ where D_j is fraction of the destination cell that intersects the unmasked source grid.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 12.5. For a grid on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

12.3.5 Second-order conservative

Like the first-order conservative method, this method's main purpose is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 12.4.) The difference between the first and second-order conservative methods is that the second-order takes the source gradient into account, so it yields a smoother destination field that typically better matches the source field. This difference between the first and second-order methods is particularly apparent when going from a coarse source grid to a finer destination grid. Another difference is that the second-order method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the second-order method, the destination field will contain values less than 0.0. The implementation of this method is based on the one described in this paper [**?**].

The weights for second-order are calculated in a similar manner to first-order 12.3.4 with additional weights that take into account the gradient across the source cell.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 12.5. For a grid on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

12.4 Conservation

Conservation means that the following equation will hold: $\sum^{all-source-cells}(V_{si} * A'_{si}) = \sum^{all-destination-cells}(V_{dj} * A'_{dj})$, where V is the variable being regridded and A is the area of a cell. The subscripts s and d refer to source and destination values, and the i and j are the source and destination grid cell indices (flattening the arrays to 1 dimension).

There are a couple of options for how the areas (A) in the proceding equation can be calculated. By default, ESMF calculates the areas. For a grid on a sphere, areas are calculated by connecting the corner coordinates of each grid cell (obtained from the grid file) with great circles. For a Cartesian grid, areas are calculated in the typcial manner for 2D polygons. If the user specifies the user area's option ("−−user_areas"), then weights will be adjusted so that the equation above will hold for the areas provided in the grid files. In either case, the areas output to the weight file are the ones for which the weights have been adjusted to conserve.

12.5 The effect of normalization options on integrals and values produced by conservative methods

It is important to note that by default (i.e. using destination area normalization) conservative regridding doesn't normalize the interpolation weights by the destination fraction. This means that for a destination grid which only partially overlaps the source grid the destination field which is output from the regrid operation should be divided by the corresponding destination fraction to yield the true interpolated values for cells which are only partially covered by the source grid. The fraction also needs to be included when computing the total source and destination integrals. To include the fraction in the conservative weights, the user can specify the fraction area normalization type. This can be done by specifying "--norm_type fracarea" on the command line.

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying "--norm_type dstarea"), the following pseudo-code shows how to adjust a destination field (`dst_field`) by the destination fraction (`dst_frac`) called `frac_b` in the weight file:

```
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_field(i)=dst_field(i)/dst_frac(i)
   end if
end for
```

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying "--norm_type dstarea"), the following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the sparse matrix multiplication of the weights in the weight file by the source field, the destination area (`dst_area`) called `area_b` in the weight file, and the destination fraction (`dst_frac`) called `frac_b` in the weight file. As in the previous paragraph, it also shows how to adjust the destination field (`dst_field`) resulting from the sparse matrix multiplication by the fraction (`dst_frac`) called `frac_b` in the weight file:

```
dst_total=0.0
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_total=dst_total+dst_field(i)*dst_area(i)
      dst_field(i)=dst_field(i)/dst_frac(i)
      ! If mass computed here after dst_field adjust, would need to be:
      ! dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
   end if
end for
```

For weights generated using fraction area normalization (set by specifying "--norm_type fracarea"), no adjustment of the destination field (`dst_field`) by the destination fraction is necessary. The following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the sparse matrix multiplication of the weights in the weight file by the source field, the destination area (`dst_area`) called `area_b` in the weight file, and the destination fraction (`dst_frac`) called `frac_b` in the weight file:

```
dst_total=0.0
for each destination element i
      dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
end for
```

For either normalization type, the following pseudo-code shows how to compute the total source integral (`src_total`) given the source field values (`src_field`), the source area (`src_area`) called `area_a` in the weight file, and the source fraction (`src_frac`) called `frac_a` in the weight file:

```
src_total=0.0
for each source element i
   src_total=src_total+src_field(i)*src_area(i)*src_frac(i)
end for
```

12.6 Usage

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

```
ESMF_RegridWeightGen
        --source|-s src_grid_filename
        --destination|-d dst_grid_filename
        --weight|-w out_weight_file
        [--method|-m bilinear|patch|nearestdtos|neareststod|conserve|conserve2nd]
        [--pole|-p none|all|teeth|1|2|..]
        [--line_type|-l cartesian|greatcircle]
        [--norm_type dstarea|fracarea]
        [--extrap_method none|neareststod|nearestidavg|nearestd|creep|creepnrstd]
        [--extrap_num_src_pnts <N>]
        [--extrap_dist_exponent <P>]
        [--extrap_num_levels <L>]
        [--ignore_unmapped|-i]
        [--ignore_degenerate]
        [-r]
        [--src_regional]
        [--dst_regional]
        [--64bit_offset]
        [--netcdf4]
        [--src_missingvalue var_name]
        [--dst_missingvalue var_name]
        [--src_coordinates lon_name,lat_name]
        [--dst_coordinates lon_name,var_name]
        [--tilefile_path filepath]
        [--src_loc center|corner]
        [--dst_loc center|corner]
        [--user_areas]
        [--weight_only]
        [--check]
        [--checkFlag]
        [--no_log]
        [--help]
        [--version]
        [-V]
```

```
where:
  --source or -s      - a required argument specifying the source grid
                        file name

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

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

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

                        bilinear     - for bilinear interpolation, also the
                                       default method if not specified.
                        patch        - for patch recovery interpolation
                        neareststod  - for nearest source to destination interpolation
                        nearestdtos  - for nearest destination to source interpolation
                        conserve     - for first-order conservative interpolation
                        conserve2nd  - for second-order conservative interpolation

  --pole or -p        - an optional argument indicating how to extrapolate
                        in the pole region.
                        The value can be one of the following:

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

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

                        <N>   - Construct an artificial pole placed in the
                                center of the top (or bottom) row of nodes,
                                but projected onto the sphere formed by the
                                rest of the grid. The value at this pole is
                                the average of the N source nodes next to
```

```
                        the pole and surrounding the destination
                        point (i.e.  the value may differ for each
                        destination point. Here N ranges from 1 to
                        the number of nodes around the pole.

--line_type
     or
     -l           - an optional argument indicating the type of path
                    lines (e.g. cell edges) follow on a spherical
                    surface. The default value depends on the regrid
                    method. For non-conservative methods the default is
                    cartesian. For conservative methods the default is
                    greatcircle.

--norm_type       - an optional argument indicating the type of normal-
                    ization to do when generating conservative weights.
                    The default value is dstarea.

--extrap_method   - an optional argument specifying which extrapolation
                    method is used to handle unmapped destination locations.
                    The value can be one of the following:

                      none         - no extrapolation method should be used.
                                     This is the default.

                      neareststod  - nearest source to destination. Each
                                     unmapped destination location is mapped
                                     to the closest source location. This
                                     extrapolation method is not supported with
                                     conservative regrid methods (e.g. conserve).

                      nearestidavg - inverse distance weighted average.
                                     The value of each unmapped destination location
                                     is the weighted average of the closest N
                                     source locations. The weight is the reciprocal
                                     of the distance of the source from the destination
                                     raised to a power P. All the weights contributing
                                     to one destination point are normalized so that
                                     they sum to 1.0. The user can choose N and P by
                                     using --extrap_num_src_pnts and
                                     --extrap_dist_exponent, but defaults are
                                     also provided. This extrapolation method is not
                                     supported with conservative regrid methods
                                     (e.g. conserve).

                      nearestd     - nearest mapped destination to
                                     unmapped destination. Each
                                     unmapped destination location is mapped
                                     to the closest mapped destination location. This
                                     extrapolation method is not supported with
                                     conservative regrid methods (e.g. conserve).
```

creep - creep fill.
Here unmapped destination points are filled by moving values from mapped locations to neighboring unmapped locations. The value filled into a new location is the average of its already filled neighbors' values. This process is repeated for the number of levels indicated by the --extrap_num_levels flag. This extrapolation method is not supported with conservative regrid methods (e.g. conserve).

creepnrstd - creep fill with nearest destination.
Here unmapped destination points are filled by first doing a creep fill, and then filling the remaining unmapped points by using the nearest destination method (both of these methods are described in the entries above). This extrapolation method is not supported with conservative regrid methods (e.g. conserve).

--extrap_num_src_pnts - an optional argument specifying how many source points should be used when the extrapolation method is nearestidavg. If not specified, the default is 8.

--extrap_dist_exponent - an optional argument specifying the exponent that the distance should be raised to when the extrapolation method is nearestidavg. If not specified, the default is 2.0.

--extrap_num_levels - an optional argument specifying how many levels should be filled for level based extrapolation methods (e.g. creep).

--ignore_unmapped
or
-i - ignore unmapped destination points. If not specified the default is to stop with an error if an unmapped point is found.

--ignore_degenerate - ignore degenerate cells in the input grids. If not specified the default is to stop with an error if an degenerate cell is found.

-r - an optional argument specifying that the source and destination grids are regional grids. If the argument is not given, the grids are assumed to be global.

--src_regional - an optional argument specifying that the source is a regional grid and the destination is a global grid.

```
--dst_regional    - an optional argument specifying that the destination
                    is a regional grid and the source is a global grid.

--64bit_offset    - an optional argument specifying that the weight file
                    will be created in the NetCDF 64-bit offset format
                    to allow variables larger than 2GB.  Note the 64-bit
                    offset format is not supported in the NetCDF version
                    earlier than 3.6.0.  An error message will be generated
                    if this flag is specified while the application is
                    linked with a NetCDF library earlier than 3.6.0.

--netcdf4         - an optional argument specifying that the output weight
                    will be created in the NetCDF4 format.  This option
                    only works with NetCDF library version 4.1 and above
                    that was compiled with the NetCDF4 file format enabled
                    (with HDF5 compression). An error message will be
                    generated if these conditions are not met.

--src_missingvalue - an optional argument that defines the variable name
                     in the source grid file if the file type is either CF Convension
                     single-tile or UGRID.  The regridder will generate a mask using
                     the missing values of the data variable.  The missing
                     value is defined using an attribute called "_FillValue"
                     or "missing_value".
 --dst_missingvalue - an optional argument that defines the variable name
                      in the destination grid file if the file type is
                      CF Convension single-tile or UGRID.  The regridder will generate a
                      the missing values of the data variable.  The missing
                      value is defined using an attribute called "_FillValue"
                      or "missing_value"

--src_coordinates - an optional argument that defines the longitude and
                    latitude variable names in the source grid file if
                    the file type is CF Convension single-tile.  The variable names are
                    separated by comma.  This argument is required in case
                    there are multiple sets of coordinate variables defined
                    in the file.  Without this argument, the offline regrid
                    application will terminate with an error message when
                    multiple coordinate variables are found in the file.

--dst_coordinates - an optional argument that defines the longitude and
                    latitude variable names in the destination grid file
                    if the file type is CF Convension single-tile.  The variable names
                    separated by comma.  This argument is required in case
                    there are multiple sets of coordinate variables defined
                    in the file.  Without this argument, the offline regrid
                    application will terminate with an error message when
                    multiple coordinate variables are found in the file.

--tilefile_path   - the alternative file path for the tile files when either the source
                    or the destination grid is a GRIDSPEC Mosaic grid.  The path can
```

```
                    be either relative or absolute.  If it is relative, it is relative
                    to the working directory.  When specified, the gridlocation variabl
                    defined in the Mosaic file will be ignored.

--src_loc         - an optional argument indicating which part of a source
                    grid cell to use for regridding. Currently, this flag is
                    only required for non-conservative regridding when the
                    source grid is an unstructured grid in ESMF or UGRID format.
                    For all other cases, only the center location is supported.
                    The value can be one of the following:

                    center - Regrid using the center location of each grid cell.

                    corner - Regrid using the corner location of each grid cell.

--dst_loc         - an optional argument indicating which part of a destination
                    grid cell to use for regridding. Currently, this flag is
                    only required for non-conservative regridding when the
                    destination grid is an unstructured grid in ESMF or UGRID format.
                    For all other cases, only the center location is supported.
                    The value can be one of the following:

                    center - Regrid using the center location of each grid cell.

                    corner - Regrid using the corner location of each grid cell.


--user_areas      - an optional argument specifying that the conservation
                    is adjusted to hold for the user areas provided in
                    the grid files. If not specified, then the
                    conservation will hold for the ESMF calculated
                    (great circle) areas.
                    Whichever areas the conservation holds for are output
                    to the weight file.

 --weight_only    - an optional argument specifying that the output weight file only
                    contains the weights and the source and destination grid's indices

 --check          - Check that the generated weights produce reasonable
                    regridded fields. This is done by calling ESMF_Regrid()
                    on an analytic source field using the weights generated
                    by this application.  The mean relative error between
                    the destination and analytic field is computed, as well
                    as the relative error between the mass of the source and
                    destination fields in the conservative case.

 --checkFlag      - Turn on more expensive extra error checking during
                    weight generation.

 --no_log         - Turn off the ESMF Log files.  By default, ESMF creates
                    multiple log files, one per PET.
```

```
--help           - Print the usage message and exit.

--version        - Print ESMF version and license information and exit.

-V               - Print ESMF version number and exit.
```

12.7 Examples

The example below shows the command to generate a set of conservative interpolation weights between a global SCRIP format source grid file (src.nc) and a global SCRIP format destination grid file (dst.nc). The weights are written into file w.nc. In this case the ESMF library and applications have been compiled using an MPI parallel communication library (e.g. setting ESMF_COMM to openmpi) to enable it to run in parallel. To demonstrate running in parallel the mpirun script is used to run the application in parallel on 4 processors.

```
mpirun -np 4 ./ESMF_RegridWeightGen -s src.nc -d dst.nc -m conserve -w w.nc
```

The next example below shows the command to do the same thing as the previous example except for three changes. The first change is this time the source grid is regional ("`--src_regional`"). The second change is that for this example bilinear interpolation ("`-m bilinear`") is being used. Because bilinear is the default, we could also omit the "`-m bilinear`". The third change is that in this example some of the destination points are expected to not be found in the source grid, but the user is ok with that and just wants those points to not appear in the weight file instead of causing an error ("`-i`").

```
mpirun -np 4 ./ESMF_RegridWeightGen -i --src_regional -s src.nc -d dst.nc \
               -m bilinear -w w.nc
```

The last example shows how to use the missing values of a data variable to generate the grid mask for a CF Convension single-tile file, how to specify the coordinate variable names using "`--src_coordinates`" and use user defined area for the conservative regridding.

```
mpirun -np 4 ./ESMF_RegridWeightGen -s src.nc -d dst.nc -m conserve \
               -w w.nc --src_missingvalue datavar \
               --src_coordinates lon,lat --user_areas
```

In the above example, "datavar" is the variable name defined in the source grid that will be used to construct the mask using its missing values. In addition, "`lon`" and "`lat`" are the variable names for the longitude and latitude values, respectively.

12.8 Grid File Formats

This section describes the grid file formats supported by ESMF. These are typically used either to describe grids to ESMF_RegridWeightGen or to create grids within ESMF. The following table summarizes the features supported by

each of the grid file formats.

Feature	SCRIP	ESMF Unstruct.	CF TILE	UGRID	GRIDSPEC Mosaic
Create an unstructured Mesh	YES	YES	NO	YES	NO
Create a logically-rectangular Grid	YES	NO	YES	NO	YES
Create a multi-tile Grid	NO	NO	NO	NO	YES
2D	YES	YES	YES	YES	YES
3D	NO	YES	NO	YES	NO
Spherical coordinates	YES	YES	YES	YES	YES
Cartesian coordinates	NO	YES	NO	NO	NO
Non-conserv regrid on corners	NO	YES	NO	YES	YES

The rest of this section contains a detailed descriptions of each grid file format along with a simple example of the format.

12.8.1 SCRIP Grid File Format

A SCRIP format grid file is a NetCDF file for describing grids. This format is the same as is used by the SCRIP [**?**] package, and so grid files which work with that package should also work here. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_SCRIP` can be used to indicate a file in this format.

SCRIP format files are capable of storing either 2D logically rectangular grids or 2D unstructured grids. The basic format for both of these grids is the same and they are distinguished by the value of the `grid_rank` variable. Logically rectangular grids have `grid_rank` set to 2, whereas unstructured grids have this variable set to 1.

The following is a sample header of a logically rectangular grid file:

```
netcdf remap_grid_T42 {
dimensions:
      grid_size = 8192 ;
      grid_corners = 4 ;
      grid_rank = 2 ;

variables:
      int grid_dims(grid_rank) ;
      double grid_center_lat(grid_size) ;
         grid_center_lat:units = "radians";
      double grid_center_lon(grid_size) ;
         grid_center_lon:units = "radians" ;
      int grid_imask(grid_size) ;
         grid_imask:units = "unitless" ;
      double grid_corner_lat(grid_size, grid_corners) ;
         grid_corner_lat:units = "radians" ;
      double grid_corner_lon(grid_size, grid_corners) ;
         grid_corner_lon:units ="radians" ;

// global attributes:
         :title = "T42 Gaussian Grid" ;
}
```

The grid_size dimension is the total number of cells in the grid; grid_rank refers to the number of dimensions. In this case grid_rank is 2 for a 2D logically rectangular grid. The integer array grid_dims gives the number of grid cells along each dimension. The number of corners (vertices) in each grid cell is given by grid_corners. The grid corner coordinates need to be listed in an order such that the corners are in counterclockwise order. Also, note that if your grid has a variable number of corners on grid cells, then you should set grid_corners to be the highest value and use redundant points on cells with fewer corners.

The integer array grid_imask is used to mask out grid cells which should not participate in the regridding. The array values should be zero for any points that do not participate in the regridding and one for all other points. Coordinate arrays provide the latitudes and longitudes of cell centers and cell corners. The unit of the coordinates can be either "radians" or "degrees".

Here is a sample header from a SCRIP unstructured grid file:

```
netcdf ne4np4-pentagons {
dimensions:
      grid_size = 866 ;
      grid_corners = 5 ;
      grid_rank = 1 ;
variables:
      int grid_dims(grid_rank) ;
      double grid_center_lat(grid_size) ;
         grid_center_lat:units = "degrees" ;
      double grid_center_lon(grid_size) ;
         grid_center_lon:units = "degrees" ;
      double grid_corner_lon(grid_size, grid_corners) ;
         grid_corner_lon:units = "degrees";
         grid_corner_lon:_FillValue = -9999. ;
      double grid_corner_lat(grid_size, grid_corners) ;
         grid_corner_lat:units = "degrees" ;
         grid_corner_lat:_FillValue = -9999. ;
      int grid_imask(grid_size) ;
         grid_imask:_FillValue = -9999. ;
      double grid_area(grid_size) ;
         grid_area:units = "radians^2" ;
         grid_area:long_name = "area weights" ;
}
```

The variables are the same as described above, however, here grid_rank = 1. In this format there is no notion of which cells are next to which, so to construct the unstructured mesh the connection between cells is defined by searching for cells with the same corner coordinates. (e.g. the same grid_corner_lat and grid_corner_lon values).

Both the SCRIP grid file format and the SCRIP weight file format work with the SCRIP 1.4 tools.

12.8.2 ESMF Unstructured Grid File Format

ESMF supports a custom unstructured grid file format for describing meshes. This format is more compatible than the SCRIP format with the methods used to create an ESMF Mesh object, so less conversion needs to be done to create a Mesh. The ESMF format is thus more efficient than SCRIP when used with

ESMF codes (e.g. the ESMF_RegridWeightGen application). When using the ESMF API, the file format flag ESMF_FILEFORMAT_ESMFMESH can be used to indicate a file in this format.

The following is a sample header in the ESMF format followed by a description:

```
netcdf mesh-esmf {
dimensions:
     nodeCount = 9 ;
     elementCount = 5 ;
     maxNodePElement = 4 ;
     coordDim = 2 ;
variables:
     double nodeCoords(nodeCount, coordDim);
            nodeCoords:units = "degrees" ;
     int elementConn(elementCount, maxNodePElement) ;
            elementConn:long_name = "Node Indices that define the element /
                                     connectivity";
            elementConn:_FillValue = -1 ;
            elementConn:start_index = 1 ;
     byte numElementConn(elementCount) ;
            numElementConn:long_name = "Number of nodes per element" ;
     double centerCoords(elementCount, coordDim) ;
            centerCoords:units = "degrees" ;
     double elementArea(elementCount) ;
            elementArea:units = "radians^2" ;
            elementArea:long_name = "area weights" ;
     int elementMask(elementCount) ;
            elementMask:_FillValue = -9999. ;
// global attributes:
            :gridType="unstructured";
            :version = "0.9" ;
```

In the ESMF format the NetCDF dimensions have the following meanings. The nodeCount dimension is the number of nodes in the mesh. The elementCount dimension is the number of elements in the mesh. The maxNodePElement dimension is the maximum number of nodes in any element in the mesh. For example, in a mesh containing just triangles, then maxNodePElement would be 3. However, if the mesh contained one quadrilateral then maxNodePElement would need to be 4. The coordDim dimension is the number of dimensions of the points making up the mesh (i.e. the spatial dimension of the mesh). For example, a 2D planar mesh would have coordDim equal to 2.

In the ESMF format the NetCDF variables have the following meanings. The nodeCoords variable contains the coordinates for each node. nodeCoords is a two-dimensional array of dimension (nodeCount,coordDim). For a 2D Grid, coordDim is 2 and the grid can be either spherical or Cartesian. If the units attribute is either degrees or radians, it is spherical. nodeCoords(:,1) contains the longitude coordinates and nodeCoords(:,2) contains the latitude coordinates. If the value of the units attribute is km, kilometers or meters, the grid is in 2D Cartesian coordinates. nodeCoords(:,1) contains the x coordinates and nodeCoords(:,2) contains the y coordinates. The same order applies to centerCoords. For a 3D Grid, coordDim is 3 and the grid is assumed to be Cartesian. nodeCoords(:,1) contains the x coordinates, nodeCoords(:,2) contains the y coordinates, and nodeCoords(:,3) contains the z coordinates. The same order applies to centerCoords. A 2D grid in the Cartesian coordinate can only be regridded into another 2D grid in the Cartesian coordinate.

The elementConn variable describes how the nodes are connected together to form each element. For each element,

this variable contains a list of indices into the nodeCoords variable pointing to the nodes which make up that element. By default, the index is 1-based. It can be changed to 0-based by adding an attribute start_index of value 0 to the elementConn variable. The order of the indices describing the element is important. The proper order for elements available in an ESMF mesh can be found in Section **??**. The file format does support 2D polygons with more corners than those in that section, but internally these are broken into triangles. For these polygons, the corners should be listed such that they are in counterclockwise order around the element. elementConn can be either a 2D array or a 1D array. If it is a 2D array, the second dimension of the elementConn variable has to be the size of the largest number of nodes in any element (i.e. maxNodePElement), the actual number of nodes in an element is given by the numElementConn variable. For a given dimension (i.e. coordDim) the number of nodes in the element indicates the element shape. For example in 2D, if numElementConn is 4 then the element is a quadrilateral. In 3D, if numElementConn is 8 then the element is a hexahedron.

If the grid contains some elements with large number of edges, using a 2D array for elementConn could take a lot of space. In that case, elementConn can be represented as a 1D array that stores the edges of all the elements continuously. When elementConn is a 1D array, the dimension maxNodePElement is no longer needed, instead, a new dimension variable connectionCount is required to define the size of elementConn. The value of connectionCount is the sum of all the values in numElementConn.

The following is an example grid file using 1D array for elementConn:

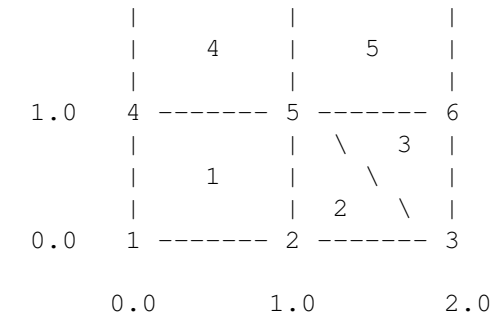
```
netcdf catchments_esmf1 {
dimensions:
        nodeCount = 1824345 ;
        elementCount = 68127 ;
        connectionCount = 18567179 ;
        coordDim = 2 ;
variables:
        double nodeCoords(nodeCount, coordDim) ;
                nodeCoords:units = ``degrees'' ;
        double centerCoords(elementCount, coordDim) ;
                centerCoords:units = ``degrees'' ;
        int elementConn(connectionCount) ;
                elementConn:polygon_break_value = -8 ;
                elementConn:start_index = 0. ;
        int numElementConn(elementCount) ;
}
```

In some cases, one mesh element may contain multiple polygons and these polygons are separated by a special value defined in the attribute polygon_break_value.

The rest of the variables in the format are optional. The centerCoords variable gives the coordinates of the center of the corresponding element. This variable is used by ESMF for non-conservative interpolation on the data field residing at the center of the elements. The elementArea variable gives the area (or volume in 3D) of the corresponding element. This area is used by ESMF during conservative interpolation. If not specified, ESMF calculates the area (or volume) based on the coordinates of the nodes making up the element. The final variable is the elementMask variable. This variable allows the user to specify a mask value for the corresponding element. If the value is 1, then the element is unmasked and if the value is 0 the element is masked. If not specified, ESMF assumes that no elements are masked.

The following is a picture of a small example mesh and a sample ESMF format header using non-optional variables describing that mesh:

```
  2.0   7 ------- 8 ------- 9
```




```
netcdf mesh-esmf {
dimensions:
        nodeCount = 9 ;
        elementCount = 5 ;
        maxNodePElement = 4 ;
        coordDim = 2 ;
variables:
        double  nodeCoords(nodeCount, coordDim);
                nodeCoords:units = "degrees" ;
        int elementConn(elementCount, maxNodePElement) ;
                elementConn:long_name = "Node Indices that define the element /
                                         connectivity";
                elementConn:_FillValue = -1 ;
        byte numElementConn(elementCount) ;
                numElementConn:long_name = "Number of nodes per element" ;
// global attributes:
                :gridType="unstructured";
                :version = "0.9" ;
data:
    nodeCoords=
        0.0, 0.0,
        1.0, 0.0,
        2.0, 0.0,
        0.0, 1.0,
        1.0, 1.0,
        2.0, 1.0,
        0.0, 2.0,
        1.0, 2.0,
        2.0, 2.0 ;

    elementConn=
        1, 2, 5,  4,
        2, 3, 5, -1,
        3, 6, 5, -1,
        4, 5, 8,  7,
        5, 6, 9,  8 ;
```

```
    numElementConn= 4, 3, 3, 4, 4 ;
}
```

12.8.3 CF Convention Single Tile File Format

ESMF_RegridWeightGen supports single tile logically rectangular lat/lon grid files that follow the NETCDF CF convention based on CF Metadata Conventions V1.6. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_CFGRID` (or its equivalent `ESMF_FILEFORMAT_GRIDSPEC`) can be used to indicate a file in this format.

An example grid file is shown below. The cell center coordinate variables are determined by the value of its attribute `units`. The longitude variable has the attribute value set to either `degrees_east`, `degree_east`, `degrees_E`, `degree_E`, `degreesE` or `degreeE`. The latitude variable has the attribute value set to `degrees_north`, `degree_north`, `degrees_N`, `degree_N`, `degreesN` or `degreeN`. The latitude and the longitude variables are one-dimensional arrays if the grid is a regular lat/lon grid, two-dimensional arrays if the grid is curvilinear. The bound coordinate variables define the bound or the corner coordinates of a cell. The bound variable name is specified in the `bounds` attribute of the latitude and longitude variables. In the following example, the latitude bound variable is `lat_bnds` and the longitude bound variable is `lon_bnds`. The bound variables are 2D arrays for a regular lat/lon grid and a 3D array for a curvilinear grid. The first dimension of the bound array is 2 for a regular lat/lon grid and 4 for a curvilinear grid. The bound coordinates for a curvilinear grid are defined in counterclockwise order. Since the grid is a regular lat/lon grid, the coordinate variables are 1D and the bound variables are 2D with the first dimension equal to 2. The bound coordinates will be read in and stored in a ESMF Grid object as the corner stagger coordinates when doing a conservative regrid. In case there are multiple sets of coordinate variables defined in a grid file, the offline regrid application will return an error for duplicate latitude or longitude variables unless "`--src_coordinates`" or "`--src_coordinates`" options are used to specify the coordinate variable names to be used in the regrid.

```
netcdf single_tile_grid {
dimensions:
time = 1 ;
bound = 2 ;
lat = 181 ;
lon = 360 ;
variables:
double lat(lat) ;
lat:bounds = "lat_bnds" ;
lat:units = "degrees_north" ;
lat:long_name = "latitude" ;
lat:standard_name = "latitude" ;
double lat_bnds(lat, bound) ;
double lon(lon) ;
lon:bounds = "lon_bnds" ;
lon:long_name = "longitude" ;
lon:standard_name = "longitude" ;
lon:units = "degrees_east" ;
double lon_bnds(lon, bound) ;
float so(time, lat, lon) ;
so:standard_name = "sea_water_salinity" ;
so:units = "psu" ;
so:missing_value = 1.e+20f ;
```

}

2D Cartesian coordinates can be supplied in additional to the required longitude/latitude coordinates. They can be used in ESMF to create a grid and used in ESMF_RegridWeightGen. The Cartesian coordinate variables have to include an "axis" attribute with value "X" or "Y". The "units" attribute can be either "m" or "meters" for meters or "km" or "kilometers" for kilometers. When a grid with 2D Cartesian coordinates are used in ESMF_RegridWeightGen, the optional arguments "--src_coordinates" or "--src_coordinates" have to be used to specify the coordinate variable names. A grid with 2D Cartesian coordinates can only be regridded with another grid in 2D Cartesian coordinates. Internally in ESMF, the Cartesian coordinates are all converted into kilometers. Here is an example of the 2D Cartesian coordinates:

```
double xc(xc) ;
        xc:long_name = "x-coordinate in Cartesian system" ;
        xc:standard_name = "projection_x_coordinate" ;
        xc:axis = "X" ;
        xc:units = "m" ;
double yc(yc) ;
        yc:long_name = "y-coordinate in Cartesian system" ;
        yc:standard_name = "projection_y_coordinate" ;
        yc:axis = "Y" ;
        yc:units = "m" ;
```

Since a CF convension tile file does not have a way to specify the grid mask, the mask is usually derived by the missing values stored in a data variable. ESMF_RegridWeightGen provides an option for users to derive the grid mask from a data variable's missing values. The value of the missing value is defined by the variable attribute missing_value or _FillValue. If the value of the data point is equal to the missing value, the grid mask for that grid point is set to 0, otherwise, it is set to 1. In the following grid, the variable so can be used to derive the grid mask. A data variable could be a 2D, 3D or 4D. For example, it may have additional depth and time dimensions. It is assumed that the first and the second dimensions of the data variable should be the longitude and the latitude dimension. ESMF_RegridWeightGen will use the first 2D data values to derive the grid mask.

12.8.4 CF Convention UGRID File Format

ESMF_RegridWeightGen supports NetCDF files that follow the UGRID conventions for unstructured grids.

The UGRID file format is a proposed extension to the CF metadata conventions for the unstructured grid data model. The latest proposal can be found at https://github.com/ugrid-conventions/ugrid-conventions. The proposal is still evolving, the Mesh creation API and ESMF_RegridWeightGen in the current ESMF release is based on UGRID Version 0.9.0 published on October 29, 2013. When using the ESMF API, the file format flag ESMF_FILEFORMAT_UGRID can be used to indicate a file in this format.

In the UGRID proposal, a 1D, 2D, or 3D mesh topology can be defined for an unstructured grid. Currently, ESMF supports two types of meshes: (1) the 2D flexible mesh topology where each cell (a.k.a. "face" as defined in the UGRID document) in the mesh is either a triangle or a quadrilateral, and (2) the fully 3D unstructured mesh topology where each cell (a.k.a. "volume" as defined in the UGRID document) in the mesh is either a tetrahedron or a hexahedron. Pyramids and wedges are not currently supported in ESMF, but they can be defined as degenerate hexahedrons. ESMF_RegridWeightGen also supports UGRID 1D network mesh topology in a limited way: A 1D mesh in UGRID can be used as the source grid for nearest neighbor regridding, and as the destination grid for non-conservative regridding.

The main addition of the UGRID extension is a dummy variable that defines the mesh topology. This additional variable has a required attribute cf_role with value "mesh_topology". In addition, it has two more required attributes: topology_dimension and node_coordinates. If it is a 1D mesh, topology_dimension is set to 1. If it is a 2D mesh (i.e., topology_dimension equals to 2), an additional attribute face_node_connectivity is required. If it is a 3D mesh (i.e., topology_dimension equals to 3), two additional attributes volume_node_connectivity and volume_shape_type are required. The value of attribute node_coordinates is a list of the names of the node longitude and latitude variables, plus the elevation variable if it is a 3D mesh. The value of attribute face_node_connectivity or volume_node_connectivity is the variable name that defines the corner node indices for each mesh cell. The additional attribute volume_shape_type for the 3D mesh points to a flag variable that specifies the shape type of each cell in the mesh.

Below is a sample 2D mesh called FVCOM_grid2d. The dummy mesh topology variable is fvcom_mesh. As described above, its cf_role attribute has to be mesh_topology and the topology_dimension attribute has to be 2 for a 2D mesh. It defines the node coordinate variable names to be lon and lat. It also specifies the face/node connectivity variable name as nv.

The variable nv is a two-dimensional array that defines the node indices of each face. The first dimension defines the maximal number of nodes for each face. In this example, it is a triangle mesh so the number of nodes per face is 3. Since each face may have a different number of corner nodes, some of the cells may have fewer nodes than the specified dimension. In that case, it is filled with the missing values defined by the attribute _FillValue. If _FillValue is not defined, the default value is -1. The nodes are in counterclockwise order. An optional attribute start_index defines whether the node index is 1-based or 0-based. If start_index is not defined, the default node index is 0-based.

The coordinate variables follows the CF metadata convention for coordinates. They are 1D array with attribute standard_name being either latitude or longitude. The units of the coordinates can be either degrees or radians.

The UGRID files may also contain data variables. The data may be located at the nodes or at the faces. Two additional attributes are introduced in the UGRID extension for the data variables: location and mesh. The location attribute defines where the data is located, it can be either face or node. The mesh attribute defines which mesh topology this variable belongs to since multiple mesh topologies may be defined in one file. The coordinates attribute defined in the CF conventions can also be used to associate the variables to their locations. ESMF checks both location and coordinates attributes to determine where the data variable is defined upon. If both attributes are present, the location attribute takes the precedence. ESMF_RegridWeightGen uses the data variable on the face to derive the element masks for the mesh cell and variable on the node to derive the node masks for the mesh.

When creating a ESMF Mesh from a UGRID file, the user has to provide the mesh topology variable name to ESMF_MeshCreate().

```
netcdf FVCOM_grid2d {
dimensions:
node = 417642 ;
nele = 826866 ;
three = 3 ;
        time  = 1 ;

variables:
// Mesh topology
int fvcom_mesh;
fvcom_mesh:cf_role = "mesh_topology" ;
fvcom_mesh:topology_dimension = 2. ;
```

```
fvcom_mesh:node_coordinates = "lon lat" ;
fvcom_mesh:face_node_connectivity = "nv" ;
int nv(nele, three) ;
nv:standard_name = "face_node_connectivity" ;
nv:start_index = 1. ;

// Mesh node coordinates
float lon(node) ;
                lon:standard_name = "longitude" ;
        lon:units = "degrees_east" ;
float lat(node) ;
                lat:standard_name = "latitude" ;
lat:units = "degrees_north" ;

// Data variable
float ua(time, nele) ;
ua:standard_name = "barotropic_eastward_sea_water_velocity" ;
ua:missing_value = -999. ;
ua:location = "face" ;
ua:mesh = "fvcom_mesh" ;
float va(time, nele) ;
va:standard_name = "barotropic_northward_sea_water_velocity" ;
va:missing_value = -999. ;
va:location = "face" ;
va:mesh = "fvcom_mesh" ;
}
```

Following is a sample 3D UGRID file containing hexahedron cells. The dummy mesh topology variable is `fvcom_mesh`. Its `cf_role` attribute has to be `mesh_topology` and `topology_dimension` attribute has to be 3 for a 3D mesh. There are two additional required attributes: `volume_node_connectivity` specifies a variable name that defines the corner indices of the mesh cells and `volume_shape_type` specifies a variable name that defines the type of the mesh cells.

The node coordinates are defined by variables `nodelon`, `nodelat` and `height`. Currently, the units attribute for the height variable is either `kilometers`, `km` or `meters`. The variable `vertids` is a two-dimensional array that defines the corner node indices of each mesh cell. The first dimension defines the maximal number of nodes for each cell. There is only one type of cells in the sample grid, i.e. hexahedrons, so the maximal number of nodes is 8. The node order is defined in **??**. The index can be either 1-based or 0-based and the default is 0-based. Setting an optional attribute `start_index` to 1 changed it to 1-based index scheme. The variable `meshtype` is a one-dimensional integer array that defines the shape type of each cell. Currently, ESMF only supports tetrahedron and hexahedron shapes. There are three attributes in `meshtype`: `flag_range`, `flag_values`, and `flag_meanings` representing the range of the flag values, all the possible flag values, and the meaning of each flag value, respectively. `flag_range` and `flag_values` are either a scalar or an array of integers. `flag_meanings` is a text string containing a list of shape types separated by space. In this example, there is only one shape type, thus, the values of `meshtype` are all 1.

```
netcdf wam_ugrid100_110 {
dimensions:
nnodes = 78432 ;
ncells = 66030 ;
eight = 8 ;
variables:
```

```
int mesh ;
mesh:cf_role = "mesh_topology" ;
mesh:topology_dimension = 3. ;
mesh:node_coordinates = "nodelon nodelat height" ;
mesh:volume_node_connectivity = "vertids" ;
mesh:volume_shape_type = "meshtype" ;
double nodelon(nnodes) ;
nodelon:standard_name = "longitude" ;
nodelon:units = "degrees_east" ;
double nodelat(nnodes) ;
nodelat:standard_name = "latitude" ;
nodelat:units = "degrees_north" ;
double height(nnodes) ;
height:standard_name = "elevation" ;
height:units = "kilometers" ;
int vertids(ncells, eight) ;
vertids:cf_role = "volume_node_connectivity" ;
vertids:start_index = 1. ;
int meshtype(ncells) ;
meshtype:cf_role = "volume_shape_type" ;
meshtype:flag_range = 1. ;
meshtype:flag_values = 1. ;
meshtype:flag_meanings = "hexahedron" ;
}
```

12.8.5 GRIDSPEC Mosaic File Format

GRIDSPEC is a draft proposal to extend the Climate and Forecast (CF) metadata conventions for the representation of gridded data for Earth System Models. The original GRIDSPEC standard was proposed by V. Balaji and Z. Liang of GFDL (see ref). GRIDSPEC extends the current CF convention to support grid mosaics, i.e., a grid consisting of multiple logically rectangular grid tiles. It also provides a mechanism for storing a grid dataset in multiple files. Therefore, it introduces different types of files, such as a mosaic file that defines the multiple tiles and their connectivity, and a tile file for a single tile grid definition on a so-called "Supergrid" format. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_MOSAIC` can be used to indicate a file in this format.

Following is an example of a mosaic file that defines a 6 tile Cubed Sphere grid:

```
netcdf C48_mosaic {
dimensions:
ntiles = 6 ;
ncontact = 12 ;
string = 255 ;
variables:
char mosaic(string) ;
mosaic:standard_name = "grid_mosaic_spec" ;
mosaic:children = "gridtiles" ;
mosaic:contact_regions = "contacts" ;
mosaic:grid_descriptor = "" ;
char gridlocation(string) ;
char gridfiles(ntiles, string) ;
```

```
char gridtiles(ntiles, string) ;
char contacts(ncontact, string) ;
contacts:standard_name = "grid_contact_spec" ;
contacts:contact_type = "boundary" ;
contacts:alignment = "true" ;
contacts:contact_index = "contact_index" ;
contacts:orientation = "orient" ;
char contact_index(ncontact, string) ;
contact_index:standard_name = "starting_ending_point_index_of_contact" ;

data:

mosaic = "C48_mosaic" ;

gridlocation = "./data/" ;

gridfiles =
  "horizontal_grid.tile1.nc",
  "horizontal_grid.tile2.nc",
  "horizontal_grid.tile3.nc",
  "horizontal_grid.tile4.nc",
  "horizontal_grid.tile5.nc",
  "horizontal_grid.tile6.nc" ;

gridtiles =
  "tile1",
  "tile2",
  "tile3",
  "tile4",
  "tile5",
  "tile6" ;

contacts =
  "C48_mosaic:tile1::C48_mosaic:tile2",
  "C48_mosaic:tile1::C48_mosaic:tile3",
  "C48_mosaic:tile1::C48_mosaic:tile5",
  "C48_mosaic:tile1::C48_mosaic:tile6",
  "C48_mosaic:tile2::C48_mosaic:tile3",
  "C48_mosaic:tile2::C48_mosaic:tile4",
  "C48_mosaic:tile2::C48_mosaic:tile6",
  "C48_mosaic:tile3::C48_mosaic:tile4",
  "C48_mosaic:tile3::C48_mosaic:tile5",
  "C48_mosaic:tile4::C48_mosaic:tile5",
  "C48_mosaic:tile4::C48_mosaic:tile6",
  "C48_mosaic:tile5::C48_mosaic:tile6" ;

 contact_index =
  "96:96,1:96::1:1,1:96",
  "1:96,96:96::1:1,96:1",
  "1:1,1:96::96:1,96:96",
  "1:96,1:1::1:96,96:96",
```

```
  "1:96,96:96::1:96,1:1",
  "96:96,1:96::96:1,1:1",
  "1:96,1:1::96:96,96:1",
  "96:96,1:96::1:1,1:96",
  "1:96,96:96::1:1,96:1",
  "1:96,96:96::1:96,1:1",
  "96:96,1:96::96:1,1:1",
  "96:96,1:96::1:1,1:96" ;
}
```

A GRIDSPEC Mosaic file is identified by a dummy variable with its `standard_name` attribute set to `grid_mosaic_spec`. The `children` attribute of this dummy variable provides the variable name that contains the tile names and the `contact_region` attribute points to the variable name that defines a list of tile pairs that are connected to each other. For a Cubed Sphere grid, there are six tiles and 12 connections. The `contacts` variable, the variable that defines the contact_region has three required attributes: `standard_name`, `contact_type`, and `contact_index`. `startand_name` has to be set to `grid_contact_spec`. `contact_type` can be either `boundary` or `overlap`. Currently, ESMF only supports non-overlapping tiles connected by `boundary`. `contact_index` defines the variable name that contains the information defining how the two adjacent tiles are connected to each other. In the above example, the `contact_index` variable contains 12 entries. Each entry contains the index of four points that defines the two edges that contact to each other from the two neighboring tiles. Assuming the four points are A, B, C, and D. A and B defines the edge of tile 1 and C and D defines the edge of tile 2. A is the same point as C and B is the same as D. (Ai, Aj) is the index for point A. The entry looks like this:

```
  Ai:Bi,Aj:Bj::Ci:Di,Cj:Dj
```

There are two fixed-name variables required in the mosaic file: variable `gridfiles` defines the associated tile file names and variable `gridlocation` defines the directory path of the tile files. The `gridlocation` can be overwritten with an command line argument `-tilefile_path` in ESMF_RegridWeightGen application.

It is possible to define a single-tile Mosaic file. If there is only one tile in the Mosaic, the `contact_region` attribute in the `grid_mosaic_spec` varilable will be ignored.

Each tile in the Mosaic is a logically rectangular lat/lon grid and is defined in a separate file. The tile file used in the GRIDSPEC Mosaic file defines the coordinates of a so-called `supergrid`. A supergrid contains all the stagger locations in one grid. It contains the corner, edge and center coordinates all in one 2D array. In this example, there are 48 elements in each side of a tile, therefore, the size of the supergrid is 48*2+1=97, i.e. 97x97.

Here is the header of one of the tile files:

```
netcdf horizontal_grid.tile1 {
dimensions:
string = 255 ;
nx = 96 ;
ny = 96 ;
nxp = 97 ;
nyp = 97 ;
variables:
char tile(string) ;
tile:standard_name = "grid_tile_spec" ;
tile:geometry = "spherical" ;
tile:north_pole = "0.0 90.0" ;
tile:projection = "cube_gnomonic" ;
```

```
tile:discretization = "logically_rectangular" ;
tile:conformal = "FALSE" ;
double x(nyp, nxp) ;
x:standard_name = "geographic_longitude" ;
x:units = "degree_east" ;
double y(nyp, nxp) ;
y:standard_name = "geographic_latitude" ;
y:units = "degree_north" ;
double dx(nyp, nx) ;
dx:standard_name = "grid_edge_x_distance" ;
dx:units = "meters" ;
double dy(ny, nxp) ;
dy:standard_name = "grid_edge_y_distance" ;
dy:units = "meters" ;
double area(ny, nx) ;
area:standard_name = "grid_cell_area" ;
area:units = "m2" ;
double angle_dx(nyp, nxp) ;
angle_dx:standard_name = "grid_vertex_x_angle_WRT_geographic_east" ;
angle_dx:units = "degrees_east" ;
double angle_dy(nyp, nxp) ;
angle_dy:standard_name = "grid_vertex_y_angle_WRT_geographic_north" ;
angle_dy:units = "degrees_north" ;
char arcx(string) ;
arcx:standard_name = "grid_edge_x_arc_type" ;
arcx:north_pole = "0.0 90.0" ;

// global attributes:
:grid_version = "0.2" ;
:history = "/home/z1l/bin/tools_20091028/make_hgrid --grid_type gnomonic_ed --nlon 96" ;
}
```

The tile file not only defines the coordinates at all staggers, it also has a complete specification of distances, angles, and areas. In ESMF, we only use the `geographic_longitude` and `geographic_latitude` variables and its subsets on the center and corner staggers. ESMF currently supports the Mosaic containing tiles of the same size. A tile can be square or rectangular. For a cubed sphere grid, each tile is a square, i.e. the x and y dimensions are the same.

12.9 Regrid Weight File Format

A regrid weight file is a NetCDF format file containing the information necessary to perform a regridding between two grids. It also optionally contains information about the grids used to compute the regridding. This information is provided to allow applications (e.g. `ESMF_RegridWeightGenCheck`) to independently compute the accuracy of the regridding weights. In some cases, `ESMF_RegridWeightGen` doesn't output the full grid information (e.g. when it's costly to compute, or when the current grid format doesn't support the type of grids used to generate the weights). In that case, the weight file can still be used for regridding, but applications which depend on the grid information may not work.

The following is the header of a sample regridding weight file that describes a bilinear regridding from a logically rectangular 2D grid to a triangular unstructured grid:

```
netcdf t42mpas-bilinear {
dimensions:
        n_a = 8192 ;
        n_b = 20480 ;
        n_s = 42456 ;
        nv_a = 4 ;
        nv_b = 3 ;
        num_wgts = 1 ;
        src_grid_rank = 2 ;
        dst_grid_rank = 1 ;
variables:
        int src_grid_dims(src_grid_rank) ;
        int dst_grid_dims(dst_grid_rank) ;
        double yc_a(n_a) ;
               yc_a:units = "degrees" ;
        double yc_b(n_b) ;
               yc_b:units = "radians" ;
        double xc_a(n_a) ;
               xc_a:units = "degrees" ;
        double xc_b(n_b) ;
               xc_b:units = "radians" ;
        double yv_a(n_a, nv_a) ;
               yv_a:units = "degrees" ;
        double xv_a(n_a, nv_a) ;
               xv_a:units = "degrees" ;
        double yv_b(n_b, nv_b) ;
               yv_b:units = "radians" ;
        double xv_b(n_b, nv_b) ;
               xv_b:units = "radians" ;
        int mask_a(n_a) ;
               mask_a:units = "unitless" ;
        int mask_b(n_b) ;
               mask_b:units = "unitless" ;
        double area_a(n_a) ;
               area_a:units = "square radians" ;
        double area_b(n_b) ;
               area_b:units = "square radians" ;
        double frac_a(n_a) ;
               frac_a:units = "unitless" ;
        double frac_b(n_b) ;
               frac_b:units = "unitless" ;
        int col(n_s) ;
        int row(n_s) ;
        double S(n_s) ;

// global attributes:
        :title = "ESMF Offline Regridding Weight Generator" ;
        :normalization = "destarea" ;
        :map_method = "Bilinear remapping" ;
        :ESMF_regrid_method = "Bilinear" ;
        :conventions = "NCAR-CSM" ;
```

```
        :domain_a = "T42_grid.nc" ;
        :domain_b = "grid-dual.nc" ;
        :grid_file_src = "T42_grid.nc" ;
        :grid_file_dst = "grid-dual.nc" ;
        :ESMF_version = "ESMF_8_2_0_beta_snapshot_05-3-g2193fa3f8a" ;
}
```

The weight file contains four types of information: a description of the source grid, a description of the destination grid, the output of the regrid weight calculation, and global attributes describing the weight file.

12.9.1 Source Grid Description

The variables describing the source grid in the weight file end with the suffix "_a". To be consistent with the original use of this weight file format the grid information is written to the file such that the location being regridded is always the cell center. This means that the grid structure described here may not be identical to that in the source grid file. The full set of these variables may not always be present in the weight file. The following is an explanation of each variable:

n_a The number of source cells.

nv_a The maximum number of corners (i.e. vertices) around a source cell. If a cell has less than the maximum number of corners, then the remaining corner coordinates are repeats of the last valid corner's coordinates.

xc_a The longitude coordinates of the centers of each source cell.

yc_a The latitude coordinates of the centers of each source cell.

xv_a The longitude coordinates of the corners of each source cell.

yv_a The latitude coordinates of the corners of each source cell.

mask_a The mask for each source cell. A value of 0, indicates that the cell is masked.

area_a The area of each source cell. This quantity is either from the source grid file or calculated by `ESMF_RegridWeightGen`. When a non-conservative regridding method (e.g. bilinear) is used, the area is set to 0.0.

src_grid_rank The number of dimensions of the source grid. Currently this can only be 1 or 2. Where 1 indicates an unstructured grid and 2 indicates a 2D logically rectangular grid.

src_grid_dims The number of cells along each dimension of the source grid. For unstructured grids this is equal to the number of cells in the grid.

12.9.2 Destination Grid Description

The variables describing the destination grid in the weight file end with the suffix "_b". To be consistent with the original use of this weight file format the grid information is written to the file such that the location being regridded is always the cell center. This means that the grid structure described here may not be identical to that in the destination grid file. The full set of these variables may not always be present in the weight file. The following is an explanation of each variable:

n_b The number of destination cells.

nv_b The maximum number of corners (i.e. vertices) around a destination cell. If a cell has less than the maximum number of corners, then the remaining corner coordinates are repeats of the last valid corner's coordinates.

xc_b The longitude coordinates of the centers of each destination cell.

yc_b The latitude coordinates of the centers of each destination cell.

xv_b The longitude coordinates of the corners of each destination cell.

yv_b The latitude coordinates of the corners of each destination cell.

mask_b The mask for each destination cell. A value of 0, indicates that the cell is masked.

area_b The area of each destination cell. This quantity is either from the destination grid file or calculated by `ESMF_RegridWeightGen`. When a non-conservative regridding method (e.g. bilinear) is used, the area is set to 0.0.

dst_grid_rank The number of dimensions of the destination grid. Currently this can only be 1 or 2. Where 1 indicates an unstructured grid and 2 indicates a 2D logically rectangular grid.

dst_grid_dims The number of cells along each dimension of the destination grid. For unstructured grids this is equal to the number of cells in the grid.

12.9.3 Regrid Calculation Output

The following is an explanation of the variables containing the output of the regridding calculation:

n_s The number of entries in the regridding matrix.

col The position in the source grid for each entry in the regridding matrix.

row The position in the destination grid for each entry in the weight matrix.

S The weight for each entry in the regridding matrix.

frac_a When a conservative regridding method is used, this contains the fraction of each source cell that participated in the regridding. When a non-conservative regridding method is used, this array is set to 0.0.

frac_b When a conservative regridding method is used, this contains the fraction of each destination cell that participated in the regridding. When a non-conservative regridding method is used, this array is set to 1.0 where the point participated in the regridding (i.e. was within the unmasked source grid), and 0.0 otherwise.

The following code shows how to apply the weights in the weight file to interpolate a source field (`src_field`) defined over the source grid to a destination field (`dst_field`) defined over the destination grid. The variables `n_s`, `n_b`, `row`, `col`, and `S` are from the weight file.

```
! Initialize destination field to 0.0
do i=1, n_b
  dst_field(i)=0.0
enddo

! Apply weights
do i=1, n_s
  dst_field(row(i))=dst_field(row(i))+S(i)*src_field(col(i))
enddo
```

If the first-order conservative interpolation method is specified ("-m conserve") then the destination field may need to be adjusted by the destination fraction (`frac_b`). This should be done if the normalization type is "dstarea" and if the destination grid extends outside the unmasked source grid. If it isn't known if the destination extends outside the source, then it doesn't hurt to apply the destination fraction. (If it doesn't extend outside, then the fraction will be 1.0 everywhere anyway.) The following code shows how to adjust an already interpolated destination field (`dst_field`) by the destination fraction. The variables `n_b`, and `frac_b` are from the weight file:

```
! Adjust destination field by fraction
do i=1, n_b
  if (frac_b(i) .ne. 0.0) then
     dst_field(i)=dst_field(i)/frac_b(i)
  endif
enddo
```

12.9.4 Weight File Description Attributes

The following is an explanation of the global attributes describing the weight file:

title Always set to "ESMF Offline Regridding Weight Generator" when generated by `ESMF_RegridWeightGen`.

normalization The normalization type used to compute conservative regridding weights. The options for this are described in section 12.3.4 which contains a description of the conservative regridding method.

map_method An indication of the mapping method which is constrained by the original use of this format. In some cases the method specified here will differ from the actual regridding method used, for example weights generated with the "patch" method will have this attribute set to "Bilinear remapping".

ESMF_regrid_method The ESMF regridding method used to generate the weight file.

conventions The set of conventions that the weight file follows. Currently only "NCAR-CSM" is supported.

domain_a The source grid file name.

domain_b The destination grid file name.

grid_file_src The source grid file name.

grid_file_dst The destination grid file name.

ESMF_version The version of ESMF used to generate the weight file.

12.9.5 Weight Only Weight File

In the current ESMF distribution, a new simplified weight file option `-weight_only` is added to `ESMF_RegridWeightGen`. The simple weight file contains only a subset of the Regrid Calculation Output defined in 12.9.3, i.e. the weights `S`, the source grid indices `col` and destination grid indices `row`. The dimension of these three variables is `n_s`.

12.10 ESMF_RegridWeightGenCheck

The ESMF_RegridWeightGen application is used in the ESMF_RegridWeightGenCheck external demo to generate interpolation weights. These weights are then tested by using them for a regridding operation and then comparing them against an analytic function on the destination grid. This external demo is also used to regression test ESMF regridding, and it is run nightly on over 150 combinations of structured and unstructured, regional and global grids, and regridding methods.

13 ESMF_Regrid

13.1 Description

This section describes the file-based regridding command line tool provided by ESMF (for a description of ESMF regridding in general see Section 24.2). Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Regridding can be broken into two stages. The first stage is generation of an interpolation weight matrix that describes how points in the source grid contribute to points in the destination grid. The second stage is the multiplication of values on the source grid by the interpolation weight matrix to produce values on the destination grid. This is implemented as a parallel sparse matrix multiplication.

The `ESMF_RegridWeightGen` command line tool described in Section 12 performs the first stage of the regridding process - generate the interpolation weight matrix. This tool not only calculates the interpolation weights, it also applies the weights to a list of variables stored in the source grid file and produces the interpolated values on the destination grid. The interpolated output variable is written out to the destination grid file. This tool supports three CF compliant file formats: the CF Single Tile grid file format(12.8.3) for a logically rectangular grid, the UGRID file format(12.8.4) for unstructured grid and the GRIDSPEC Mosaic file format(12.8.5) for cubed-sphere grid. For the GRIDSPEC Mosaic file format, the data are stored in seperate data files, one file per tile. The SCRIP format(12.8.1) and the ESMF unstructured grid format(12.8.2) are not supported because there is no way to define a variable field using these two formats. Currently, the tool only works with 2D grids, the support for the 3D grid will be made available in the future release. The variable array can be up to four dimensions. The variable type is currently limited to single or double precision real numbers. The support for other data types, such as integer or short will be added in the future release.

The user interface of this tool is greatly simplified from `ESMF_RegridWeightGen`. User only needs to provide two input file names, the source and the destination variable names and the regrid method. The tool will figure out the type of the grid file automatically based on the attributes of the variable. If the variable has a `coordinates` attribute, the grid file is a GRIDSPEC file and the value of the `coordinates` defines the longitude and latitude variable's names. For example, following is a simple GRIDSPEC file with a variable named `PSL` and coordinate variables named `lon` and `lat`.

```
netcdf simple_gridspec {
dimensions:
      lat = 192 ;
      lon = 288 ;
variables:
      float PSL(lat, lon) ;
         PSL:time = 50. ;
         PSL:units = "Pa" ;
         PSL:long_name = "Sea level pressure" ;
         PSL:cell_method = "time: mean" ;
         PSL:coordinates = "lon lat" ;
      double lat(lat) ;
         lat:long_name = "latitude" ;
         lat:units = "degrees_north" ;
      double lon(lon) ;
         lon:long_name = "longitude" ;
```

```
         lon:units = "degrees_east" ;
}
```

If the variable has a `mesh` attribute and a `location` attribute, the grid file is in UGRID format(12.8.4). The value of `mesh` attribute is the name of a dummy variable that defines the mesh topology. If the application performs a conservative regridding, the value of the `location` attribute has to be `face`, otherwise, it has to be `node`. This is because ESMF only supports non-conservative regridding on the data stored at the nodes of a ESMF_Mesh object, and conservative regridding on the data stored at the cells of a ESMF_Mesh object.

Here is an example 2D UGRID file:

```
netcdf simple_ugrid {
dimensions:
      node = 4176 ;
      nele = 8268 ;
      three = 3 ;
      time  = 2 ;
variables:
      float lon(node) ;
         lon:units = "degrees_east" ;
      float lat(node) ;
         lat:units = "degrees_north" ;
      float lonc(nele) ;
         lonc:units = "degrees_east" ;
      float latc(nele) ;
         latc:units = "degrees_north" ;
      int nv(nele, three) ;
         nv:standard_name = "face_node_connectivity" ;
         nv:start_index = 1. ;
      float zeta(time, node) ;
         zeta:standard_name = "sea_surface_height_above_geoid" ;
         zeta:_FillValue = -999. ;
         zeta:location = "node" ;
         zeta:mesh = "fvcom_mesh" ;
      float ua(time, nele) ;
         ua:standard_name = "barotropic_eastward_sea_water_velocity" ;
         ua:_FillValue = -999. ;
         ua:location = "face" ;
         ua:mesh = "fvcom_mesh" ;
      float va(time, nele) ;
         va:standard_name = "barotropic_northward_sea_water_velocity" ;
         va:_FillValue = -999. ;
         va:location = "face" ;
         va:mesh = "fvcom_mesh" ;
      int fvcom_mesh(node) ;
         fvcom_mesh:cf_role = "mesh_topology" ;
         fvcom_mesh:dimension = 2. ;
         fvcom_mesh:locations = "face node" ;
         fvcom_mesh:node_coordinates = "lon lat" ;
         fvcom_mesh:face_coordinates = "lonc latc" ;
         fvcom_mesh:face_node_connectivity = "nv" ;
```

```
}
```

There are three variables defined in the above UGRID file - `zeta` on the node of the mesh, `ua` and `va` on the face of the mesh. All three variables have one extra time dimension.

The GRIDSPEC MOSAIC file(12.8.5) can be identified by a dummy variable with `standard_name` attribute set to `grid_mosaic_spec`. The data for a GRIDSPEC Mosaic file are stored in seperate files, one tile per file. The name of the data file is not specified in the mosaic file. Therefore, additional optional argument `-srcdatafile` or `-dstdatafile` is required to provide the prefix of the datafile. The datafile is also a CF compliant NetCDF file. The complete name of the datafile is constructed by appending the tilename (defined in the Mosaic file in a variable specified by the `children` attribute of the dummy variable). For instance, if the prefix of the datafile is `mosaicdata`, then the datafile names are `mosaicdata.tile1.nc`, `mosaicdata.tile2.nc`, etc... using the mosaic file example in 12.8.5. The path of the datafile is defined by `gridlocation` variable, similar to the tile files. To overwrite it, an optional argument `tilefile_path` can be specified.

Following is an example GRIDSPEC MOSAIC datafile:

```
netcdf mosaictest.tile1 {
dimensions:
     grid_yt = 48 ;
     grid_xt = 48 ;
     time = UNLIMITED ; // (12 currently)
variables:
     float area_land(grid_yt, grid_xt) ;
        area_land:long_name = "area in the grid cell" ;
        area_land:units = "m2" ;
     float evap_land(time, grid_yt, grid_xt) ;
        evap_land:long_name = "vapor flux up from land" ;
        evap_land:units = "kg/(m2 s)" ;
        evap_land:coordinates = "geolon_t geolat_t" ;
     double geolat_t(grid_yt, grid_xt) ;
        geolat_t:long_name = "latitude of grid cell centers" ;
        geolat_t:units = "degrees_N" ;
     double geolon_t(grid_yt, grid_xt) ;
        geolon_t:long_name = "longitude of grid cell centers" ;
        geolon_t:units = "degrees_E" ;
     double time(time) ;
        time:long_name = "time" ;
        time:units = "days since 1900-01-01 00:00:00" ;
}
```

This is a database for the C48 Cubed Sphere grid defined in 12.8.5. Note currently we assume that the data are located at the center stagger of the grid. The coordinate variables `geolon_t` and `geolat_t` should be identical to the center coordinates defined in the corresponding tile files. They are not used to create the multi-tile grid. For this application, they are only used to construct the analytic field to check the correctness of the regridding results if `-check` argument is given.

If the variable specified for the destination file does not already exist in the file, the file type is determined as follows: First search for a variable that has a `cf_role` attribute of value `mesh_topology`. If successful, the file is a UGRID file. The destination variable will be created on the nodes if the regrid method is non-conservative and an optional argument `dst_loc` is set to `corner`. Otherwise, the destination variable will be created on the face. If the destination file is not a UGRID file, check if there is a variable with its `units` attribute set to `degrees_east` and

another variable with it's `units` attribute set to `degrees_west`. If such a pair is found, the file is a GRIDSPEC file and the above two variables will be used as the coordinate variables for the variable to be created. If more than one pair of coordinate variables are found in the file, the application will fail with an error message.

If the destination variable exists in the destination grid file, it has to have the same number of dimensions and the same type as the source variable. Except for the latitude and longitude dimensions, the size of the destination variable's extra dimensions (e.g., time and vertical layers) has to match with the source variable. If the destination varialbe does not exist in the destination grid file, a new variable will be created with the same type and matching dimensions as the source variable. All the attributes of the source variable will be copied to the destination variable except those related to the grid definition (i.e. `coordinates` attribute if the destination file is in GRIDSPEC or MOSAIC format or `mesh` and `location` attributes if the destination file is in UGRID format.

Additional rules beyond the CF convention are adopted to determine whether there is a time dimension defined in the source and destination files. In this application, only a dimension with a name `time` is considered as a time dimension. If the source variable has a `time` dimension and the destination variable is not already defined, the application first checks if there is a `time` dimension defined in the destination file. If so, the values of the `time` dimension in both files have to be identical. If the time dimension values don't match, the application terminates with an error message. The application does not check the existence of a `time` variable or if the `units` attribute of the `time` variable match in two input files. If the destination file does not have a `time` dimension, it will be created. UNLIMITED time dimension is allowed in the source file, but the `time` dimension created in the destination file is not UNLIMITED.

This application requires the NetCDF library to read the grid files and write out the interpolated variables. To compile ESMF with the NetCDF library, please refer to the "Third Party Libraries" Section in the ESMF User's Guide for more information.

Internally this application uses the ESMF public API to perform regridding. If a source or destination grid is logically rectangular, then `ESMF_GridCreate()`(**??**) is used to create an ESMF_Grid object from the file. The coordinate variables are stored at the center stagger location (`ESMF_STAGGERLOC_CENTER`). If the application performs a conservative regridding, the `addCornerStager` argument is set to `TRUE` and the bound variables in the grid file will be read in and stored at the corner stagger location (`ESMF_STAGGERLOC_CORNER`). If the variable has an `_FillValue` attribute defined, a mask will be generated using the missing values of the variable. The data variable is defined as a ESMF_Field object at the center stagger location (`ESMF_STAGGERLOC_CENTER`) of the grid.

If the source grid is an unstructured grid and the the regrid method is nearest neighbor, or if the destination grid is unstructured and the regrid method is non-conservative, `ESMF_LocStreamCreate()`(**??** is used to create an ESMF_LocStream object. Otherwise, `ESMF_MeshCreate()`(**??**) is used to create an ESMF_Mesh object for the unstructured input grids. Currently, only the 2D unstructured grid is supported. If the application performs a conservative regridding, the variable has to be defined on the face of the mesh cells, i.e., its `location` attribute has to be set to `face`. Otherwise, the variable has to be defined on the node and its (`location` attribute is set to `node`).

If a source or a destination grid is a Cubed Sphere grid defined in GRIDSPEC MOSAIC file format, `ESMF_GridCreateMosaic()`(**??**) will be used to create a multi-tile ESMF_Grid object from the file. The coordinates at the center and the corner stagger in the tile files will be stored in the grid. The data has to be located at the center stagger of the grid.

Similar to the `ESMF_RegridWeightGen` command line tool (Section 12), this application supports bilinear, patch, nearest neighbor, first-order and second-order conservative interpolation. The descriptions of different interpolation methods can be found at Section 24.2 and Section 12. It also supports different pole methods for non-conservative interpolation and allows user to choose to ignore the errors when some of the destination points cannot be mapped by any source points.

If the optional argument `-check` is given, the interpolated fields will be checked agaist a synthetic field defined as follows:

13.2 Usage

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

```
ESMF_Regrid
        --source|-s src_grid_filename
        --destination|-d dst_grid_filename
--src_var var_name[,var_name,..]
--dst_var var_name[,var_name,..]
        [--srcdatafile]
        [--dstdatafile]
        [--tilefile_path filepath]
        [--dst_loc center|corner]
        [--method|-m bilinear|patch|nearestdtos|neareststod|conserve|conserve2nd]
        [--pole|-p none|all|teeth|1|2|..]
        [--ignore_unmapped|-i]
        [--ignore_degenerate]
        [-r]
        [--src_regional]
        [--dst_regional]
        [--check]
        [--no_log]
[--help]
        [--version]
        [-V]
where
  --source or -s      - a required argument specifying the source grid
                        file name

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

  --src_var           - a required argument specifying the variable names
                        in the src grid file to be interpolated from.  If more
                        than one, separated them with comma.

  --dst_var           - a required argument specifying the variable names
                        to be interpolated to.  If more than one, separated
                        them with comma. The variable may or may not
                        exist in the destination grid file.

  --srcdatafile       - If the source grid is a GRIDSPEC MOSAIC grid, the data
                        is stored in separate files, one per tile. srcdatafile
                        is the prefix of the source data file.  The filename
                        is srcdatafile.tilename.nc, where tilename is the tile
                        name defined in the MOSAIC file.
```

```
--srcdatafile       - If the destination grid is a GRIDSPEC MOSAIC grid, the data
                      is stored in separate files, one per tile. dstdatafile
                      is the prefix of the destination data file.  The filename
                      is dstdatafile.tilename.nc, where tilename is the tile
                      name defined in the MOSAIC file.

--tilefile_path    - the alternative file path for the tile files and the
                      data files when either the source or the destination grid
                      is a GRIDSPEC MOSAIC grid.  The path can be either relative
                      or absolute.  If it is relative, it is relative to the
                      working directory.  When specified, the gridlocation variable
                      defined in the Mosaic file will be ignored.

  --dst_loc         - an optional argument that specifies whether the destination
                      variable is located at the center or the corner of the grid
                      if the destination variable does not exist in the destination
                      grid file. This flag is only required for non-conservative
                      regridding when the destination grid is in UGRID format.
                      For all other cases, only the center location is supported
                      that is also the default value if this argument is not specified.

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

                      bilinear   - for bilinear interpolation, also the
                                   default method if not specified.
                      patch      - for patch recovery interpolation
                      nearstdtos - for nearest destination to source interpolation
                      nearststod - for nearest source to destination interpolation
                      conserve   - for first-order conservative interpolation

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

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

                      teeth - No new pole point is constructed, instead
                              the holes at the poles are filled by
                              constructing triangles across the top and
                              bottom row of the source Grid. This can be
                              useful because no averaging occurs, however,
                              because the top and bottom of the sphere are
```

```
                        now flat, for a big enough mismatch between
                        the size of the destination and source pole
                        regions, some destination points may still
                        not be able to be mapped to the source Grid.

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

--ignore_unmapped
       or
       -i          - ignore unmapped destination points. If not specified
                     the default is to stop with an error if an unmapped
                     point is found.

--ignore_degenerate - ignore degenerate cells in the input grids. If not specified
                     the default is to stop with an error if an degenerate
                     cell is found.

-r                  - an optional argument specifying that the source and
                     destination grids are regional grids.  If the argument
                     is not given, the grids are assumed to be global.

--src_regional      - an optional argument specifying that the source is
                     a regional grid and the destination is a global grid.

--dst_regional      - an optional argument specifying that the destination
                     is a regional grid and the source is a global grid.

--check             - Check the correctness of the interpolated destination
                     variables against an analytic field. The source variable
                     has to be synthetically constructed using the same analytic
                     method in order to perform meaningful comparison.
                     The analytic field is calculated based on the coordinate
                     of the data point.  The formular is as follows:
                     data(i,j,k,l)=2.0+cos(lat(i,j))**2*cos(2.0*lon(i,j))+(k-1)+2*(l-1)
                     The data field can be up to four dimensional with the
                     first two dimension been longitude and latitude.
                     The mean relative error between the destination and
                     analytic field is computed.

 --no_log           - Turn off the ESMF error log.

 --help             - Print the usage message and exit.
```

```
--version        - Print ESMF version and license information and exit.

-V               - Print ESMF version number and exit.
```

13.3 Examples

The example below regrids the node variable `zeta` defined in the sample UGRID file(13.1) to the destination grid defined in the sample GRIDSPEC file(13.1) using bilinear regridding method and write the interpolated data into a variable named `zeta`.

```
mpirun -np 4 ESMF_Regrid -s simple_ugrid.nc -d simple_gridspec.nc \
             --src_var zeta --dst_var zeta
```

In this case, the destination variable does not exist in `simple_ugrid.nc` and the `time` dimension is not defined in the destination file. The resulting output file has a new time dimension and a new variable `zeta`. The attributes from the source variable `zeta` are copied to the destination variable except for `mesh` and `location`. A new attribute `coordinates` is created for the destination variable to specify the names of the coordinate variables. The header of the output file looks like:

```
netcdf simple_gridspec {
dimensions:
      lat = 192 ;
      lon = 288 ;
      time = 2  ;
variables:
      float PSL(lat, lon) ;
         PSL:time = 50. ;
         PSL:units = "Pa" ;
         PSL:long_name = "Sea level pressure" ;
         PSL:cell_method = "time: mean" ;
         PSL:coordinates = "lon lat" ;
      double lat(lat) ;
         lat:long_name = "latitude" ;
         lat:units = "degrees_north" ;
      double lon(lon) ;
         lon:long_name = "longitude" ;
         lon:units = "degrees_east" ;
      float zeta(time, lat, lon) ;
         zeta:standard_name = "sea_surface_height_above_geoid" ;
         zeta:_FillValue = -999. ;
         zeta:coordinates = "lon lat" ;
}
```

The next example shows the command to do the same thing as the previous example but for a different variable `ua`. Since `ua` is defined on the face, we can only do a conservative regridding.

```
mpirun -np 4 ESMF_Regrid -s simple_ugrid.nc -d simple_gridspec.nc \
             --src_var ua --dst_var ua -m conserve
```

14 ESMF_Scrip2Unstruct

14.1 Description

The ESMF_Scrip2Unstruct application is a parallel program that converts a SCRIP format grid file 12.8.1 into an unstructured grid file in the ESMF unstructured file format 12.8.2 or in the UGRID file format 12.8.4. This application program can be used together with ESMF_RegridWeightGen 12 application for the unstructured SCRIP format grid files. An unstructured SCRIP grid file will be converted into the ESMF unstructured file format internally in ESMF_RegridWeightGen. The conversion subroutine used in ESMF_RegridWeightGen is sequential and could be slow if the grid file is very big. It will be more efficient to run the ESMF_Scrip2Unstruct first and then regrid the output ESMF or UGRID file using ESMF_RegridWeightGen. Note that a logically rectangular grid file in the SCRIP format (i.e. the dimension grid_rank is equal to 2) can also be converted into an unstructured grid file with this application.

The application usage is as follows:

```
ESMF_Scrip2Unstruct  inputfile outputfile dualflag [fileformat]

where
  inputfile       - a SCRIP format grid file

  outputfile      - the output file name

  dualflag        - 0 for straight conversion and 1 for dual
    mesh.  A dual mesh is a mesh constructed
                    by putting the corner coordinates in the
                    center of the elements and using the
    center coordinates to form the mesh
    corner vertices.

  fileformat      - an optional argument for the output file
    format.  It could be either ESMF or UGRID.
                    If not specified, the output file is in
    the ESMF format.
```

Part III

Superstructure

15 Overview of Superstructure

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

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

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

15.1 Superstructure Classes

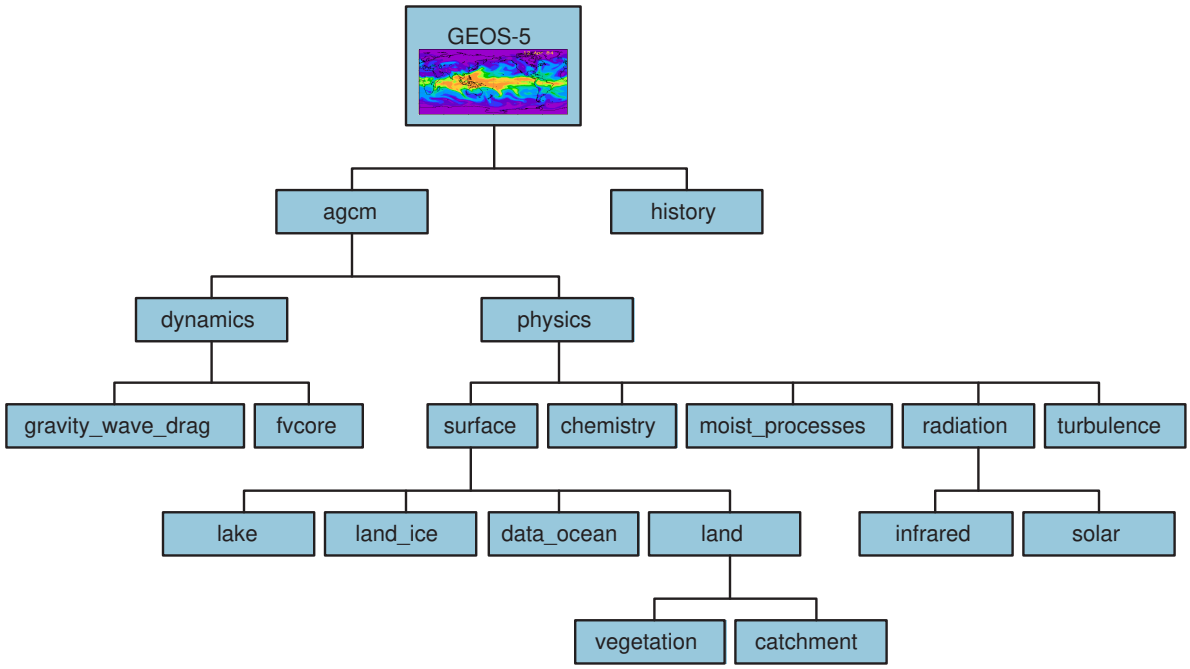
There are a small number of classes in the ESMF superstructure:

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

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

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

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



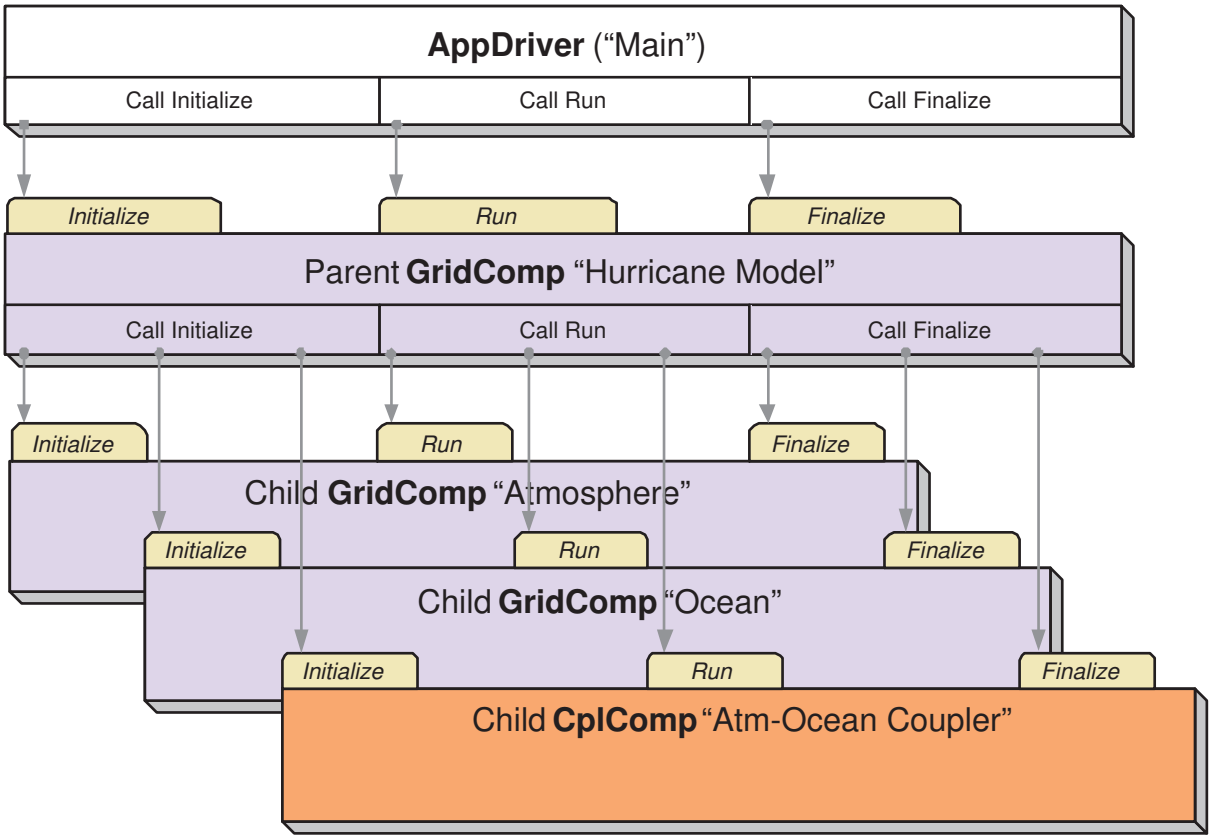
An ESMF coupled application typically involves a parent Gridded Component, two or more child Gridded Components and one or more Coupler Components.

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

15.2 Hierarchical Creation of Components

Components are allocated computational resources in the form of **Persistent Execution Threads**, or **PET**s. A list of a Component's PETs is contained in a structure called a **Virtual Machine**, or **VM**. The VM also contains information about the topology and characteristics of the underlying computer. Components are created hierarchically, with parent Components creating child Components and allocating some or all of their PETs to each one. By default ESMF creates a new VM for each child Component, which allows Components to tailor their VM resources to match their needs. In some cases, a child may want to share its parent's VM - ESMF supports this, too.

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



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

15.3 Sequential and Concurrent Execution of Components

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

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

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

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

15.4 Intra-Component Communication

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

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

15.5 Data Distribution and Scoping in Components

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

15.6 Performance

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

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

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

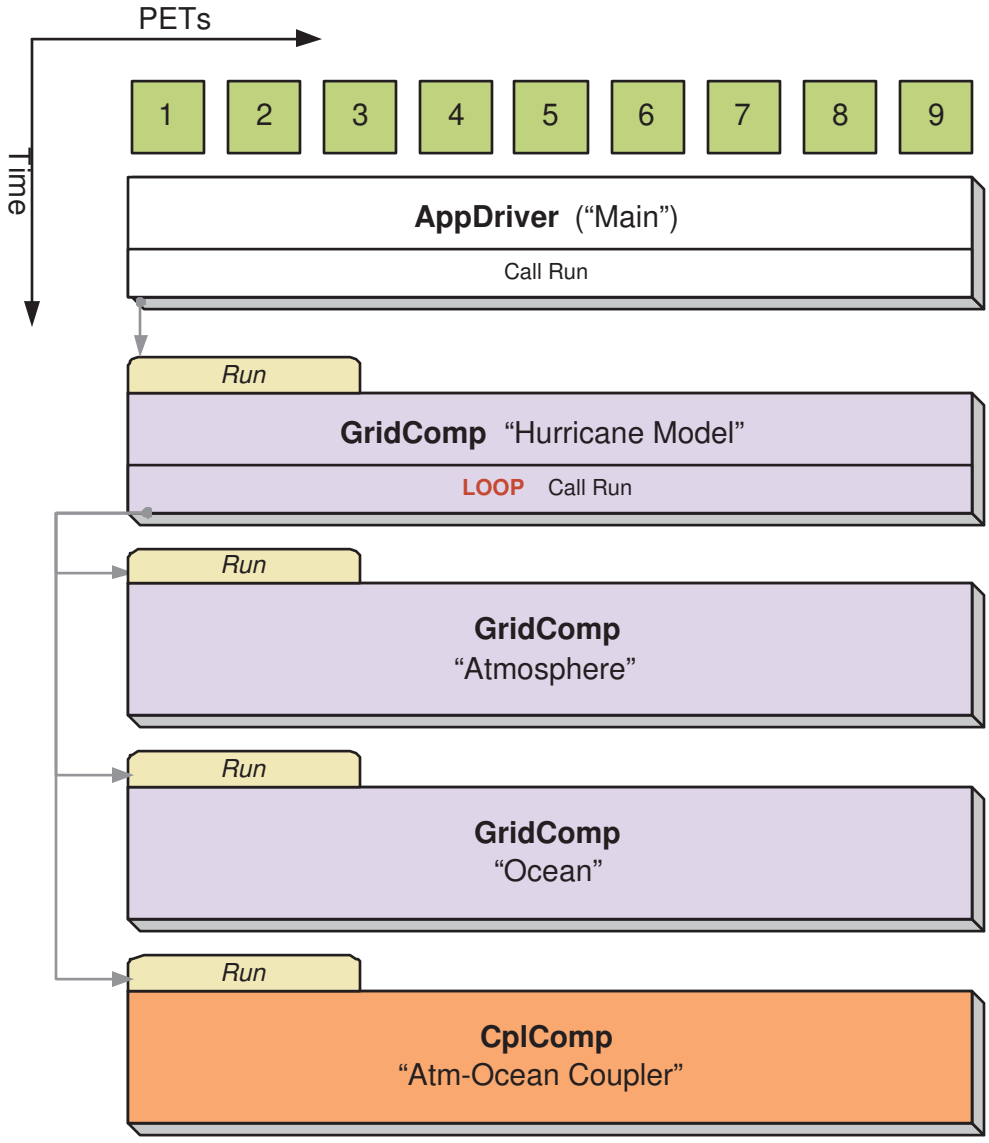


Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The application driver,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

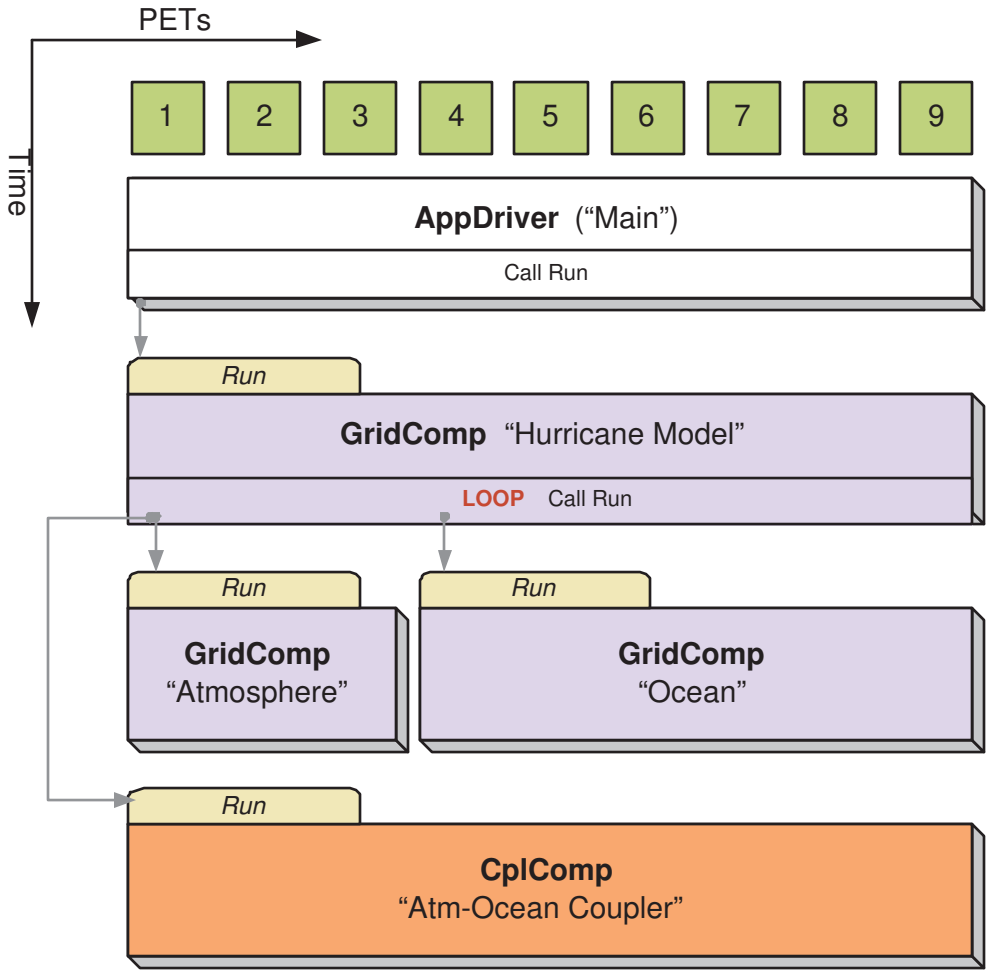
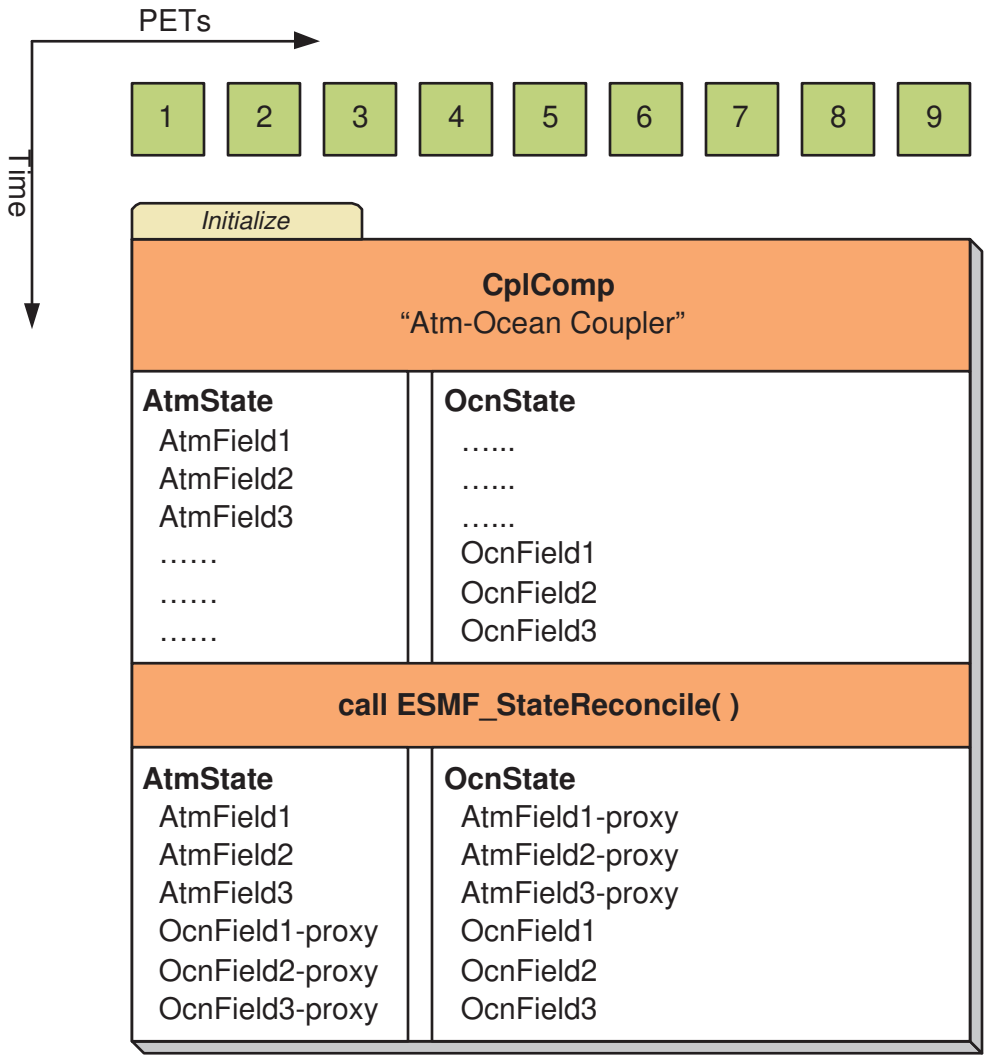
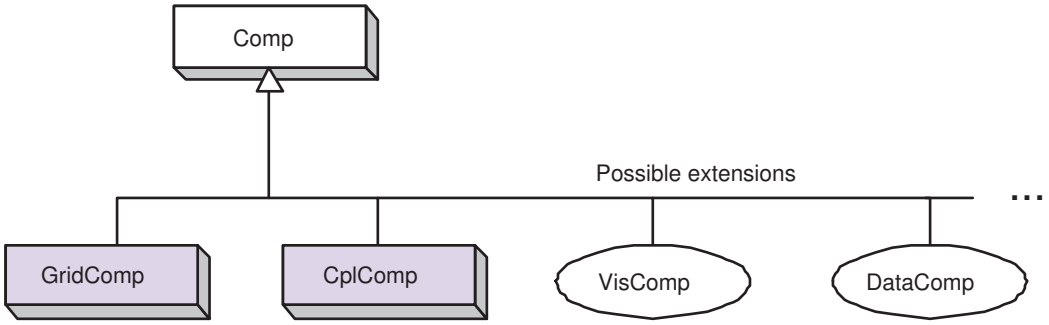


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



15.7 Object Model

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



16 Application Driver and Required ESMF Methods

16.1 Description

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

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

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

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

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

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

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

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

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

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

The alternative is to start a different executable program on different PETs. This is a **MPMD** model, Multiple Program Multiple Data. There are complications with many job control systems on multiprocessor machines in getting the different executables started, and getting inter-process communications established. ESMF currently has some support for MPMD: different Components can run as separate executables, but the Coupler that transfers data between the Components must still run on the union of their PETs. This means that the Coupler Component must be linked into all of the executables.

16.2 Constants

16.2.1 ESMF_END

DESCRIPTION:

The `ESMF_End_Flag` determines how an ESMF application is shut down.

The type of this flag is:

```
type(ESMF_End_Flag)
```

The valid values are:

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

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

ESMF_END_KEEPMPI Same as `ESMF_END_NORMAL` but `MPI_Finalize()` will *not* be called. It is the user code's responsibility to shut down MPI cleanly if necessary.

16.3 Use and Examples

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

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

```
<seq|concur>_<pairwise|hub>_<f|c>driver_<spmd|mpmd>
```

The example that is currently implemented is `seq_pairwise_fdriver_spmd`, which has sequential component execution, a pairwise coupler, a main program in Fortran, and all processors launching the same executable. It is also copied automatically into a top-level `quick_start` directory at compilation time.

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

Examples of concurrent component execution can be found in the system tests that are bundled with the ESMF distribution.

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

    The ChangeMe.F90 file that's included below contains a number of
    definitions that are used by the AppDriver, such as the name of the
    application's main configuration file and the name of the application's
    SetServices routine.  This file is in the same directory as the
    AppDriver.F90 file.
  ------------------------------------------------------------------------------

 #include "ChangeMe.F90"

     program ESMF_AppDriver
 #define ESMF_METHOD "program ESMF_AppDriver"

 #include "ESMF.h"

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

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

     implicit none

  ------------------------------------------------------------------------------
```

```
  Define local variables
------------------------------------------------------------------------

   ! Components and States
   type(ESMF_GridComp) :: compGridded
   type(ESMF_State) :: defaultstate

   ! Configuration information
   type(ESMF_Config) :: config

   ! A common Grid
   type(ESMF_Grid) :: grid

   ! A Clock, a Calendar, and timesteps
   type(ESMF_Clock) :: clock
   type(ESMF_TimeInterval) :: timeStep
   type(ESMF_Time) :: startTime
   type(ESMF_Time) :: stopTime

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

   ! Return codes for error checks
   integer :: rc, localrc

------------------------------------------------------------------------
  Initialize ESMF.  Note that an output Log is created by default.
------------------------------------------------------------------------

   call ESMF_Initialize(defaultCalKind=ESMF_CALKIND_GREGORIAN, rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

   call ESMF_LogWrite("ESMF AppDriver start", ESMF_LOGMSG_INFO)

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

   config = ESMF_ConfigCreate(rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

   call ESMF_ConfigLoadFile(config, USER_CONFIG_FILE, rc = localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Get configuration information.
```

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

   call ESMF_ConfigGetAttribute(config, i_max, label='I Counts:', &
     default=10, rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)
   call ESMF_ConfigGetAttribute(config, j_max, label='J Counts:', &
     default=40, rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create the top Gridded Component.
------------------------------------------------------------------------

   compGridded = ESMF_GridCompCreate(name="ESMF Gridded Component", &
       rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

   call ESMF_LogWrite("Component Create finished", ESMF_LOGMSG_INFO)

------------------------------------------------------------------------
  Register the set services method for the top Gridded Component.
------------------------------------------------------------------------

   call ESMF_GridCompSetServices(compGridded, userRoutine=SetServices, rc=rc)
   if (ESMF_LogFoundError(rc, msg="Registration failed", rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create and initialize a Clock.
------------------------------------------------------------------------

     call ESMF_TimeIntervalSet(timeStep, s=2, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     call ESMF_TimeSet(startTime, yy=2004, mm=9, dd=25, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     call ESMF_TimeSet(stopTime, yy=2004, mm=9, dd=26, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
```

```
          call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     clock = ESMF_ClockCreate(timeStep, startTime, stopTime=stopTime, &
               name="Application Clock", rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create and initialize a Grid.

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

     grid = ESMF_GridCreateNoPeriDim(maxIndex=(/i_max, j_max/), &
                             name="source grid", rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     ! Attach the grid to the Component
     call ESMF_GridCompSet(compGridded, grid=grid, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

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

     defaultstate = ESMF_StateCreate(name="Default State", rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

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

     call ESMF_GridCompInitialize(compGridded, importState=defaultstate, &
       exportState=defaultstate, clock=clock, rc=localrc)
     if (ESMF_LogFoundError(rc, msg="Initialize failed", rcToReturn=rc)) &
         call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)
```

```
      call ESMF_GridCompRun(compGridded, importState=defaultstate, &
        exportState=defaultstate, clock=clock, rc=localrc)
      if (ESMF_LogFoundError(rc, msg="Run failed", rcToReturn=rc)) &
          call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

      call ESMF_GridCompFinalize(compGridded, importState=defaultstate, &
        exportState=defaultstate, clock=clock, rc=localrc)
      if (ESMF_LogFoundError(rc, msg="Finalize failed", rcToReturn=rc)) &
          call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)


------------------------------------------------------------------------
  Destroy objects.
------------------------------------------------------------------------

      call ESMF_ClockDestroy(clock, rc=localrc)
      if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
        ESMF_CONTEXT, rcToReturn=rc)) &
        call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

      call ESMF_StateDestroy(defaultstate, rc=localrc)
      if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
        ESMF_CONTEXT, rcToReturn=rc)) &
        call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

      call ESMF_GridCompDestroy(compGridded, rc=localrc)
      if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
        ESMF_CONTEXT, rcToReturn=rc)) &
        call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Finalize and clean up.
------------------------------------------------------------------------

    call ESMF_Finalize()

    end program ESMF_AppDriver
```

16.4 Required ESMF Methods

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

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

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

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

16.4.1 ESMF_Initialize - Initialize ESMF

INTERFACE:

```
subroutine ESMF_Initialize(configFilename, &
  defaultCalKind, defaultDefaultLogFilename, defaultLogFilename, &
  defaultLogAppendFlag, logAppendFlag, defaultLogKindFlag, logKindFlag, &
  mpiCommunicator,  ioUnitLBound, ioUnitUBound, &
  defaultGlobalResourceControl, globalResourceControl, config, vm, rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    character(len=*),        intent(in),  optional :: configFilename
    type(ESMF_CalKind_Flag), intent(in),  optional :: defaultCalKind
    character(len=*),        intent(in),  optional :: defaultDefaultLogFilename
    character(len=*),        intent(in),  optional :: defaultLogFilename
    logical,                 intent(in),  optional :: defaultLogAppendFlag
    logical,                 intent(in),  optional :: logAppendFlag
    type(ESMF_LogKind_Flag), intent(in),  optional :: defaultLogKindFlag
    type(ESMF_LogKind_Flag), intent(in),  optional :: logKindFlag
    integer,                 intent(in),  optional :: mpiCommunicator
    integer,                 intent(in),  optional :: ioUnitLBound
    integer,                 intent(in),  optional :: ioUnitUBound
    logical,                 intent(in),  optional :: defaultGlobalResourceControl
    logical,                 intent(in),  optional :: globalResourceControl
    type(ESMF_Config),       intent(out), optional :: config
    type(ESMF_VM),           intent(out), optional :: vm
    integer,                 intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  7.0.0 Added argument logAppendFlag to allow specifying that the existing log files will be overwritten.

8.2.0 Added argument `globalResourceControl` to support ESMF-aware threading and resource control on the global VM level.
Added argument `config` to return default handle to the defaultConfig.
Renamed argument `defaultConfigFilename` to `configFilename`, in order to clarify that provided settings in the Config file are *not* defaults, but final overrides.
Introduce `default` prefixed arguments: `defaultDefaultLogFilename`, `defaultLogAppendFlag`, `defaultLogKindFlag`, `defaultGlobalResourceControl`. These arguments allow specification of defaults for the associated settings. This default can be overridden via the associated argument, without the extra `default` prefix, either specified in the call, or within the specified Config file.

DESCRIPTION:

This method must be called once on each PET before any other ESMF methods are used. The method contains a barrier before returning, ensuring that all processes made it successfully through initialization.

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

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

`ESMF_Initialize()` supports running ESMF inside a user MPI program. Details of this feature are discussed under the VM example **??**. It is not necessary that all MPI ranks are handed to ESMF. Section **??** shows how an MPI communicator can be used to execute ESMF on a subset of MPI ranks. `ESMF_Initialize()` supports running multiple concurrent instances of ESMF under the same user MPI program. This feature is discussed under **??**.

In order to use any of the advanced resource management functions that ESMF provides via the `ESMF_*CompSetVM*()` methods, the MPI environment must be thread-safe. `ESMF_Initialize()` handles this automatically if it is in charge of initializing MPI. However, if the user code initializes MPI before calling into `ESMF_Initialize()`, it must do so via `MPI_Init_thread()`, specifying `MPI_THREAD_SERIALIZED` or above for the required level of thread support.

In cases where `ESMF_*CompSetVM*()` methods are used to move processing elements (PEs), i.e. CPU cores, between persistent execution threads (PETs), ESMF uses POSIX signals between PETs. In order to do so safely, the proper signal handlers must be installed *before* MPI is initialized. `ESMF_Initialize()` handles this automatically if it is in charge of initializing MPI. If, however, MPI is explicitly initialized by user code, then to ensure correct signal handling it is necessary to call `ESMF_InitializePreMPI()` from the user code prior to the MPI initialization.

By default, `ESMF_Initialize()` will open multiple error log files, one per processor. This is very useful for debugging purpose. However, when running the application on a large number of processors, opening a large number of log files and writing log messages from all the processors could become a performance bottleneck. Therefore, it is recommended to turn the Error Log feature off in these situations by setting `logKindFlag` to ESMF_LOGKIND_NONE.

When integrating ESMF with applications where Fortran unit number conflicts exist, the optional `ioUnitLBound` and `ioUnitUBound` arguments may be used to specify an alternate unit number range. See section **??** for more information on how ESMF uses Fortran unit numbers.

Before exiting the application the user must call `ESMF_Finalize()` to release resources and clean up ESMF gracefully. See the `ESMF_Finalize()` documentation about details relating to the MPI environment.

The arguments are:

[configFilename] Name of the configuration file for the entire application. If this argument is specified, the configuration file must exist, and its content is read during `ESMF_Initialize()`. If any of the following labels are found in the specified configuration file, their values are used to set the associated `ESMF_Initialize()` argument, overriding any defaults. If the same argument is also specified in the `ESMF_Initialize()` call directly, an error is returned, and ESMF is not initialized. The supported config labels are:

- `defaultLogFilename:`
- `logAppendFlag:`
- `logKindFlag:`
- `globalResourceControl:`

[defaultCalKind] Sets the default calendar to be used by ESMF Time Manager. See section **??** for a list of valid options. If not specified, defaults to `ESMF_CALKIND_NOCALENDAR`.

[defaultDefaultLogFilename] Default value for argument `defaultLogFilename`, the name of the default log file for warning and error messages. If not specified, the default is `ESMF_LogFile`.

[defaultLogFilename] Name of the default log file for warning and error messages. If not specified, defaults according to `defaultDefaultLogFilename`.

[defaultLogAppendFlag] Default value for argument `logAppendFlag`, indicating the overwrite behavior in case the default log file already exists. If not specified, the default is `.true.`.

[logAppendFlag] If the default log file already exists, a value of `.false.` will set the file position to the beginning of the file. A value of `.true.` sets the position to the end of the file. If not specified, defaults according to `defaultLogAppendFlag`.

[defaultLogKindFlag] Default value for argument `logKindFlag`, setting the LogKind of the default ESMF log. If not specified, the default is `ESMF_LOGKIND_MULTI`.

[logKindFlag] Sets the LogKind of the default ESMF log. See section **??** for a list of valid options. If not specified, defaults according to `defaultLogKindFlag`.

[mpiCommunicator] MPI communicator defining the group of processes on which the ESMF application is running. See section **??** and **??** for details. If not specified, defaults to `MPI_COMM_WORLD`.

[ioUnitLBound] Lower bound for Fortran unit numbers used within the ESMF library. Fortran units are primarily used for log files. Legal unit numbers are positive integers. A value higher than 10 is recommended in order to avoid the compiler-specific reservations which are typically found on the first few units. If not specified, defaults to `ESMF_LOG_FORT_UNIT_NUMBER`, which is distributed with a value of 50.

[ioUnitUBound] Upper bound for Fortran unit numbers used within the ESMF library. Must be set to a value at least 5 units higher than `ioUnitLBound`. If not specified, defaults to `ESMF_LOG_UPPER`, which is distributed with a value of 99.

[defaultGlobalResourceControl] Default value for argument `globalResourceControl`, indicating whether PETs of the global VM are pinned to PEs and the OpenMP threading level is reset. If not specified, the default is `.false.`.

[globalResourceControl] For `.true.`, each global PET is pinned to the corresponding PE (i.e. CPU core) in order. Further, if OpenMP support is enabled for the ESMF installation (during build time), the `OMP_NUM_THREADS` is set to `1` on every PET, regardless of the setting in the launching environment. The `.true.` setting is recommended for applications that utilize the ESMF-aware threading and resource control features. For `.false.`, global PETs are *not* pinned by ESMF, and `OMP_NUM_THREADS` is *not* modified. If not specified, defaults according to `defaultGlobalResourceControl`.

[config] Returns the default `ESMF_Config` if the `configFilename` argument was provided. Otherwise the presence of this argument triggers an error.

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

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

16.4.2 ESMF_InitializePreMPI - Initialize parts of ESMF that must happen before MPI is initialized

INTERFACE:

```
subroutine ESMF_InitializePreMPI(rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                intent(out), optional :: rc
```

DESCRIPTION:

This method is *only* needed for cases where MPI is initialized explicitly by user code. In most typical cases ESMF_Initialize() is called before MPI is initialized, and takes care of all the internal initialization, including MPI.

There are circumstances where it is necessary or convenient to initialize MPI before calling into ESMF_Initialize(). This option is supported by ESMF, and for most cases no special action is required on the user side. However, for cases where ESMF_*CompSetVM*() methods are used to move processing elements (PEs), i.e. CPU cores, between persistent execution threads (PETs), ESMF uses POSIX signals between PETs. In order to do so safely, the proper signal handlers must be installed before MPI is initialized. This is accomplished by calling ESMF_InitializePreMPI() from the user code prior to the MPI initialization.

Note also that in order to use any of the advanced resource management functions that ESMF provides via the ESMF_*CompSetVM*() methods, the MPI environment must be thread-safe. ESMF_Initialize() handles this automatically if it is in charge of initializing MPI. However, if the user code initializes MPI before calling into ESMF_Initialize(), it must do so via MPI_Init_thread(), specifying MPI_THREAD_SERIALIZED or above for the required level of thread support.

The arguments are:

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

16.4.3 ESMF_IsInitialized - Query Initialized status of ESMF

INTERFACE:

```
function ESMF_IsInitialized(rc)
```

RETURN VALUE:

```
      logical :: ESMF_IsInitialized
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                intent(out), optional :: rc
```

DESCRIPTION:

Returns `.true.` if the framework has been initialized. This means that `ESMF_Initialize()` has been called. Otherwise returns `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

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

16.4.4 ESMF_IsFinalized - Query Finalized status of ESMF

INTERFACE:

```
    function ESMF_IsFinalized(rc)
```

RETURN VALUE:

```
      logical :: ESMF_IsFinalized
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                intent(out), optional :: rc
```

DESCRIPTION:

Returns `.true.` if the framework has been finalized. This means that `ESMF_Finalize()` has been called. Otherwise returns `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

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

16.4.5 ESMF_Finalize - Clean up and shut down ESMF

INTERFACE:

```
subroutine ESMF_Finalize(endflag, rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_End_Flag), intent(in), optional  :: endflag
   integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

This must be called once on each PET before the application exits to allow ESMF to flush buffers, close open connections, and release internal resources cleanly. The optional argument `endflag` may be used to indicate the mode of termination. Note that this call must be issued only once per PET with `endflag=ESMF_END_NORMAL`, and that this call may not be followed by `ESMF_Initialize()`. This last restriction means that it is not possible to restart ESMF within the same execution.

The arguments are:

[endflag] Specify mode of termination. The default is `ESMF_END_NORMAL` which waits for all PETs of the global VM to reach `ESMF_Finalize()` before termination. See section 16.2.1 for a complete list and description of valid flags.

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

16.4.6 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. For technical reasons *none* of the arguments in user-provided subroutines must be declared as *optional*.

The only *required* public interface of a Component is its SetServices method. This subroutine must have an externally accessible name (be a public symbol), take a component as the first argument, and an integer return code as the second. Both arguments are required and must *not* be declared as `optional`. If an intent is specified in the interface it must be `intent(inout)` for the first and `intent(out)` for the second argument. The subroutine name is not predefined, it is set by the component writer, but must be provided as part of the component documentation.

The required function that the SetServices subroutine must provide is to specify the user-code entry points for the standard ESMF Component methods. To this end the user-written SetServices routine calls the

ESMF_<Grid/Cpl>CompSetEntryPoint() method to set each Component entry point.

See sections 17.2.1 and 18.2.1 for examples of how to write a user-code SetServices routine.

Note that a component does not call its own SetServices routine; the AppDriver or parent component code, which is creating a component, will first call ESMF_<Grid/Cpl>CompCreate() to create a component object, and then must call into ESMF_<Grid/Cpl>CompSetServices(), supplying the user-code SetServices routine as an argument. The framework then calls into the user-code SetServices, after the Component's VM has been started up.

It is good practice to package the user-code implementing a component into a Fortran module, with the user-code SetService routine being the only public module method. ESMF supports three mechanisms for accessing the user-code SetServices routine from the calling AppDriver or parent component.

- **Fortran USE association**: The AppDriver or parent component utilizes the standard Fortran USE statement on the component module to make all public entities available. The user-code SetServices routine can then be passed directly into the ESMF_<Grid/Cpl>CompSetServices() interface documented in 17.4.19 and 18.4.19, respectively.

  Pros: Standard Fortran module use: name mangling and interface checking is handled by the Fortran compiler.

  Cons: Fortran 90/95 has no mechanism to implement a "smart" dependency scheme through USE association. Any change in a lower level component module (even just adding or changing a comment!) will trigger a complete recompilation of all of the higher level components throughout the component hierarchy. This situation is particularly annoying for ESMF componentized code, where the prescribed ESMF component interfaces, in principle, remove all interdependencies between components that would require recompilation.

  Fortran *submodules*, introduced as an extension to Fortran 2003, and now part for the Fortran 2008 standard, are designed to avoid this "false" dependency issue. A code change to an ESMF component that keeps the actual implementation within a submodule, will not trigger a recompilation of the components further up in the component hierarchy. Unfortunately, as of mid-2015, only two compiler vendors support submodules.

- **External routine**: The AppDriver or parent component provides an explicit interface block for an external routine that implements (or calls) the user-code SetServices routine. This routine can then be passed directly into the ESMF_<Grid/Cpl>CompSetServices() interface documented in 17.4.19 and 18.4.19, respectively. (In practice this can be implemented by the component as an external subroutine that simply calls into the user-code SetServices module routine.)

  Pros: Avoids Fortran USE dependencies: a change to lower level component code will not trigger a complete recompilation of all of the higher level components throughout the component hierarchy. Name mangling is handled by the Fortran compiler.

  Cons: The user-code SetServices interface is not checked by the compiler. The user must ensure uniqueness of the external routine name across the entire application.

- **Name lookup**: The AppDriver or parent component specifies the user-code SetServices routine by name. The actual lookup and code association does not occur until runtime. The name string is passed into the ESMF_<Grid/Cpl>CompSetServices() interface documented in 17.4.20 and 18.4.20, respectively.

  Pros: Avoids Fortran USE dependencies: a change to lower level component code will not trigger a complete recompilation of all of the higher level components throughout the component hierarchy. The component code does not have to be accessible until runtime and may be located in a shared object, thus avoiding relinking of the application.

  Cons: The user-code SetServices interface is not checked by the compiler. The user must explicitly deal with all of the Fortran name mangling issues: 1) Accessing a module routine requires precise knowledge of the name mangling rules of the specific compiler. Alternatively, the user-code SetServices routine may be implemented as an external routine, avoiding the module name mangling. 2) Even then, Fortran compilers typically

append one or two underscores on a symbol name. This must be considered when passing the name into the ESMF_<Grid/Cpl>CompSetServices() method.

16.4.7 User-code Initialize, Run, and Finalize methods

The required standard ESMF Component methods, for which user-code entry points must be set, are Initialize, Run, and Finalize. Currently optional, a Component may also set entry points for the WriteRestart and ReadRestart methods.

Sections 17.2.1 and 18.2.1 provide examples of how the entry points for Initialize, Run, and Finalize are set during the user-code SetServices routine, using the ESMF_<Grid/Cpl>CompSetEntryPoint() library call.

All standard user-code methods must abide *exactly* to the prescribed interfaces. *None* of the arguments must be declared as *optional*.

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.

See sections 17.2.2, 17.2.3, 17.2.4, and 18.2.2, 18.2.3, 18.2.4 for examples of how to write entry points for the standard ESMF Component methods.

16.4.8 User-code SetVM method

When the AppDriver or parent component code calls ESMF_<Grid/Cpl>CompCreate() it has the option to specify a petList argument. All of the parent PETs contained in this list become resources of the child component. By default, without the petList argument, all of the parent PETs are provided to the child component.

Typically each component has its own virtual machine (VM) object. However, using the optional contextflag argument during ESMF_<Grid/Cpl>CompCreate() a child component can inherit its parent component's VM. Unless a child component inherits the parent VM, it has the option to set certain aspects of how its VM utilizes the provided resources. The resources provided via the parent PETs are the associated processing elements (PEs) and virtual address spaces (VASs).

The optional user-written SetVM routine is called from the parent for the child through the ESMF_<Grid/Cpl>CompSetVM() method. This is the only place where the child component can set aspects of its own VM before it is started up. The child component's VM must be running before the SetServices routine can be called, and thus the parent must call the optional ESMF_<Grid/Cpl>CompSetVM() method *before* ESMF_<Grid/Cpl>CompSetServices().

Inside the user-code called by the SetVM routine, the component has the option to specify how the PETs share the provided parent PEs. Further, PETs on the same single system image (SSI) can be set to run multi-threaded within a reduced number of virtual address spaces (VAS), allowing a component to leverage shared memory concepts.

Sections 17.2.5 and 18.2.5 provide examples for simple user-written SetVM routines.

One common use of the SetVM approach is to implement hybrid parallelism based on MPI+OpenMP. Under ESMF, each component can use its own hybrid parallelism implementation. Different components, even if running on the same PE resources, do not have to agree on the number of MPI processes (i.e. PETs), or the number of OpenMP threads launched under each PET. Hybrid and non-hybrid components can be mixed within the same application. Coupling between components of any flavor is supported under ESMF.

In order to obtain best performance when using SetVM based resource control for hybrid parallelism, it is *strongly recommended* to set OMP_WAIT_POLICY=PASSIVE in the environment. This is one of the standard OpenMP environment variables. The PASSIVE setting ensures that OpenMP threads relinquish the PEs as soon as they have

completed their work. Without that setting ESMF resource control threads can be delayed, and context switching between components becomes more expensive.

16.4.9 Use of `internal procedures` as user-provided procedures

Internal procedures are nested within a surrounding procedure, and only local to the surrounding procedure. They are specified by using the CONTAINS statement.

Prior to Fortran-2008 an internal procedure could not be used as a user-provided callback procedure. In Fortran-2008 this restriction was lifted. It is important to note that if ESMF is passed an internal procedure, that the surrounding procedure be active whenever ESMF calls it. This helps ensure that local variables at the surrounding procedures scope are properly initialized.

When internal procedures contained within a main program unit are used for callbacks, there is no problem. This is because the main program unit is always active. However when internal procedures are used within other program units, initialization could become a problem. The following outlines the issue:

```
module my_procs_mod
  use ESMF
  implicit none

contains

  subroutine my_procs (...)
    integer :: my_setting
    :
    call ESMF_GridCompSetEntryPoint(gridcomp, methodflag=ESMF_METHOD_INITIALIZE, &
        userRoutine=my_grid_proc_init, rc=localrc)
    :
    my_setting = 42

  contains

    subroutine my_grid_proc_init (gridcomp, importState, exportState, clock, rc)
      :
    ! my_setting is possibly uninitialized when my_grid_proc_init is used as a call-back
      something = my_setting
      :
    end subroutine my_grid_proc_init
  end subroutine my_procs
end module my_procs_mod
```

The Fortran standard does not specify whether variable *my_setting* is statically or automatically allocated, unless it is explicitly given the SAVE attribute. Thus there is no guarantee that its value will persist after *my_procs* has finished. The SAVE attribute is usually given to a variable via specifying a SAVE attribute in its delaration. However it can also be inferred by initializing the variable in its declaration:

```
    :
    integer, save : my_setting
    :
```

or,

```
:
integer :: my_setting = 42
:
```

Because of the potential initialization issues, it is recommended that internal procedures only be used as ESMF callbacks when the surrounding procedure is also active.

17 GridComp Class

17.1 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, such as an atmospheric model or an ocean model. Gridded Components may also represent individual processes, such as radiation or chemistry. It's up to the application writer to decide how deeply to "componentize."

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

A well designed Gridded Component does not store information internally about how it couples to other Gridded Components. That allows it to be used in different contexts without changes to source code. The idea here is to avoid situations in which slightly different versions of the same model source are maintained for use in different contexts - standalone vs. coupled versions, for example. Data is passed in and out of Gridded Components using an ESMF State, this is described in Section 21.1.

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

ESMF provides the Gridded Component derived type, `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 `ESMF_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.

17.2 Use and Examples

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

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

17.2.1 Implement a user-code `SetServices` routine

Every `ESMF_GridComp` is required to provide and document a public set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_GridComp` as the first argument, and an integer return code as the second. Both arguments are required and must *not* be declared as `optional`. If an intent is specified in the interface it must be `intent(inout)` for the first and `intent(out)` for the second argument.

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

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

```
! Example Gridded Component
module ESMF_GriddedCompEx

! ESMF Framework module
use ESMF
implicit none
public GComp_SetServices
public GComp_SetVM

contains

subroutine GComp_SetServices(comp, rc)
  type(ESMF_GridComp)   :: comp   ! must not be optional
  integer, intent(out)  :: rc     ! must not be optional

  ! Set the entry points for standard ESMF Component methods
  call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_INITIALIZE, &
                            userRoutine=GComp_Init, rc=rc)
  call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_RUN, &
                            userRoutine=GComp_Run, rc=rc)
  call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_FINALIZE, &
                            userRoutine=GComp_Final, rc=rc)

  rc = ESMF_SUCCESS

end subroutine
```

17.2.2 Implement 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 interface shown below. Arguments must not be declared as 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                  ! must not be optional
  type(ESMF_State)      :: importState           ! must not be optional
  type(ESMF_State)      :: exportState           ! must not be optional
  type(ESMF_Clock)      :: clock                 ! must not be optional
  integer, intent(out)  :: rc                    ! must not be optional

  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_StateAdd(exportState, field, rc)
  !call ESMF_StateAdd(exportState, bundle, rc)
  print *, "Gridded Comp Init returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Init
```

17.2.3 Implement 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 interface shown below. Arguments must not be declared as 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                  ! must not be optional
  type(ESMF_State)      :: importState           ! must not be optional
  type(ESMF_State)      :: exportState           ! must not be optional
  type(ESMF_Clock)      :: clock                 ! must not be optional
  integer, intent(out)  :: rc                    ! must not be optional
```

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

  ! This is where the model specific computation goes.

  ! Fill export state here using ESMF_StateAdd(), etc

  print *, "Gridded Comp Run returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Run
```

17.2.4 Implement 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 component writer must supply a subroutine with the exact interface shown below. Arguments must not be declared as optional, and the types and order must match.

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                  ! must not be optional
  type(ESMF_State)      :: importState           ! must not be optional
  type(ESMF_State)      :: exportState           ! must not be optional
  type(ESMF_Clock)      :: clock                 ! must not be optional
  integer, intent(out)  :: rc                    ! must not be optional

  print *, "Gridded Comp Final starting"

  ! Add whatever code here needed

  print *, "Gridded Comp Final returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Final
```

17.2.5 Implement a user-code `SetVM` routine

Every `ESMF_GridComp` can optionally provide and document a public set vm routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_GridComp` as the first argument, and an

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

The set vm routine is the only place where the child component can use the `ESMF_GridCompSetVMMaxPEs()`, or `ESMF_GridCompSetVMMaxThreads()`, or `ESMF_GridCompSetVMMinThreads()` call to modify aspects of its own VM.

A component's VM is started up right before its set services routine is entered. `ESMF_GridCompSetVM()` is executing in the parent VM, and must be called *before* `ESMF_GridCompSetServices()`.

```
subroutine GComp_SetVM(comp, rc)
  type(ESMF_GridComp)   :: comp   ! must not be optional
  integer, intent(out)  :: rc     ! must not be optional

  type(ESMF_VM) :: vm
  logical :: pthreadsEnabled

  ! Test for Pthread support, all SetVM calls require it
  call ESMF_VMGetGlobal(vm, rc=rc)
  call ESMF_VMGet(vm, pthreadsEnabledFlag=pthreadsEnabled, rc=rc)

  if (pthreadsEnabled) then
    ! run PETs single-threaded
    call ESMF_GridCompSetVMMinThreads(comp, rc=rc)
  endif

  rc = ESMF_SUCCESS

end subroutine

end module ESMF_GriddedCompEx
```

17.2.6 Set and Get the Internal State

ESMF provides the concept of an Internal State that is associated with a Component. Through the Internal State API a user can attach a private data block to a Component, and later retrieve a pointer to this memory allocation. Setting and getting of Internal State information are supported from anywhere in the Component's SetServices, Initialize, Run, or Finalize code.

The code below demonstrates the basic Internal State API of `ESMF_<Grid|Cpl>SetInternalState()` and `ESMF_<Grid|Cpl>GetInternalState()`. Notice that an extra level of indirection to the user data is necessary!

```
! ESMF Framework module
use ESMF
use ESMF_TestMod
implicit none

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

! Internal State Variables
type testData
```

```
  sequence
    integer :: testValue
    real    :: testScaling
  end type

  type dataWrapper
  sequence
    type(testData), pointer :: p
  end type

  type(dataWrapper) :: wrap1, wrap2
  type(testData), target :: data
  type(testData), pointer :: datap  ! extra level of indirection


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

  call ESMF_Initialize(defaultlogfilename="InternalStateEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

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

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

!------------------------------------------------------------------------
! This could be called, for example, during a Component's initialize phase.

    ! Initialize private data block
  data%testValue = 4567
  data%testScaling = 0.5

  ! Set Internal State
  wrap1%p => data
  call ESMF_GridCompSetInternalState(comp, wrap1, rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

!------------------------------------------------------------------------
! This could be called, for example, during a Component's run phase.

  ! Get Internal State
  call ESMF_GridCompGetInternalState(comp, wrap2, rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

  ! Access private data block and verify data
  datap => wrap2%p
  if ((datap%testValue .ne. 4567) .or. (datap%testScaling .ne. 0.5)) then
    print *, "did not get same values back"
    finalrc = ESMF_FAILURE
  else
    print *, "got same values back from GetInternalState as original"
  endif
```

When working with ESMF Internal States it is important to consider the applying scoping rules. The user must ensure that the private data block that is being referenced persists for the entire access period. This is not an issue in the previous example, where the private data block was defined on the scope of the main program. However, the Internal State construct is often useful inside of Component modules to hold Component specific data between calls. One option to ensure persisting private data blocks is to use the Fortran SAVE attribute either on local or module variables. A second option, illustrated in the following example, is to use Fortran pointers and user controlled memory management via allocate() and deallocate() calls.

One situation where the Internal State is useful is in the creation of ensembles of the same Component. In this case it can be tricky to distinguish which data, held in saved module variables, belongs to which ensemble member - especially if the ensemble members are executing on the same set of PETs. The Internal State solves this problem by providing a handle to instance specific data allocations.

```
module user_mod

  use ESMF

  implicit none

  ! module variables
  private

  ! Internal State Variables
  type testData
  sequence
    integer       :: testValue        ! scalar data
    real          :: testScaling      ! scalar data
    real, pointer :: testArray(:)     ! array data
  end type

  type dataWrapper
  sequence
    type(testData), pointer :: p
  end type



  contains !------------------------------------------------------------



  subroutine mygcomp_init(gcomp, istate, estate, clock, rc)
    type(ESMF_GridComp):: gcomp
    type(ESMF_State):: istate, estate
    type(ESMF_Clock):: clock
    integer, intent(out):: rc

    ! Local variables
    type(dataWrapper) :: wrap
    type(testData), pointer :: data
    integer :: i
```

```
  rc = ESMF_SUCCESS

  ! Allocate private data block
  allocate(data)

  ! Initialize private data block
  data%testValue = 4567         ! initialize scalar data
  data%testScaling = 0.5        ! initialize scalar data
  allocate(data%testArray(10))  ! allocate array data

  do i=1, 10
    data%testArray(i) = real(i) ! initialize array data
  enddo

  ! In a real ensemble application the initial data would be set to
  ! something unique for this ensemble member. This could be
  ! accomplished for example by reading a member specific config file
  ! that was specified by the driver code. Alternatively, Attributes,
  ! set by the driver, could be used to label the Component instances
  ! as specific ensemble members.

  ! Set Internal State
  wrap%p => data
  call ESMF_GridCompSetInternalState(gcomp, wrap, rc)

end subroutine !-------------------------------------------------------

subroutine mygcomp_run(gcomp, istate, estate, clock, rc)
  type(ESMF_GridComp):: gcomp
  type(ESMF_State):: istate, estate
  type(ESMF_Clock):: clock
  integer, intent(out):: rc

  ! Local variables
  type(dataWrapper) :: wrap
  type(testData), pointer :: data
  logical :: match = .true.
  integer :: i

  rc = ESMF_SUCCESS

  ! Get Internal State
  call ESMF_GridCompGetInternalState(gcomp, wrap, rc)
  if (rc/=ESMF_SUCCESS) return

  ! Access private data block and verify data
  data => wrap%p
  if (data%testValue .ne. 4567) match = .false.   ! test scalar data
  if (data%testScaling .ne. 0.5) match = .false.  ! test scalar data
  do i=1, 10
    if (data%testArray(i) .ne. real(i)) match = .false. ! test array data
  enddo

  if (match) then
```

```
      print *, "got same values back from GetInternalState as original"
    else
      print *, "did not get same values back"
      rc = ESMF_FAILURE
    endif

  end subroutine !-------------------------------------------------------

  subroutine mygcomp_final(gcomp, istate, estate, clock, rc)
    type(ESMF_GridComp):: gcomp
    type(ESMF_State):: istate, estate
    type(ESMF_Clock):: clock
    integer, intent(out):: rc

    ! Local variables
    type(dataWrapper) :: wrap
    type(testData), pointer :: data

    rc = ESMF_SUCCESS

    ! Get Internal State
    call ESMF_GridCompGetInternalState(gcomp, wrap, rc)
    if (rc/=ESMF_SUCCESS) return

    ! Deallocate private data block
    data => wrap%p
    deallocate(data%testArray)  ! deallocate array data
    deallocate(data)

  end subroutine !-------------------------------------------------------------


end module
```

17.3 Restrictions and Future Work

1. **No optional arguments.** User-written routines called by SetServices, and registered for Initialize, Run and Finalize, *must not* declare any of the arguments as optional.

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

3. **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.

17.4 Class API

17.4.1 ESMF_GridCompAssignment(=) - GridComp assignment

INTERFACE:

```
interface assignment(=)
gridcomp1 = gridcomp2
```

ARGUMENTS:

```
type(ESMF_GridComp) :: gridcomp1
type(ESMF_GridComp) :: gridcomp2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Assign gridcomp1 as an alias to the same ESMF GridComp object in memory as gridcomp2. If gridcomp2 is invalid, then gridcomp1 will be equally invalid after the assignment.

The arguments are:

gridcomp1 The `ESMF_GridComp` object on the left hand side of the assignment.

gridcomp2 The `ESMF_GridComp` object on the right hand side of the assignment.

17.4.2 ESMF_GridCompOperator(==) - GridComp equality operator

INTERFACE:

```
interface operator(==)
  if (gridcomp1 == gridcomp2) then ... endif
            OR
  result = (gridcomp1 == gridcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in) :: gridcomp1
type(ESMF_GridComp), intent(in) :: gridcomp2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Test whether gridcomp1 and gridcomp2 are valid aliases to the same ESMF GridComp object in memory. For a more general comparison of two ESMF GridComps, going beyond the simple alias test, the ESMF_GridCompMatch() function (not yet implemented) must be used.

The arguments are:

gridcomp1 The `ESMF_GridComp` object on the left hand side of the equality operation.

gridcomp2 The `ESMF_GridComp` object on the right hand side of the equality operation.

17.4.3 ESMF_GridCompOperator(/=) - GridComp not equal operator

INTERFACE:

```
interface operator(/=)
  if (gridcomp1 /= gridcomp2) then ... endif
            OR
  result = (gridcomp1 /= gridcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in) :: gridcomp1
type(ESMF_GridComp), intent(in) :: gridcomp2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Test whether gridcomp1 and gridcomp2 are *not* valid aliases to the same ESMF GridComp object in memory. For a more general comparison of two ESMF GridComps, going beyond the simple alias test, the ESMF_GridCompMatch() function (not yet implemented) must be used.

The arguments are:

gridcomp1 The ESMF_GridComp object on the left hand side of the non-equality operation.

gridcomp2 The ESMF_GridComp object on the right hand side of the non-equality operation.

17.4.4 ESMF_GridCompCreate - Create a GridComp

INTERFACE:

```
recursive function ESMF_GridCompCreate(grid, gridList, &
  mesh, meshList, locstream, locstreamList, xgrid, xgridList, &
  config, configFile, clock, petList, contextflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_GridComp) :: ESMF_GridCompCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_Grid),         intent(in),    optional :: grid
  type(ESMF_Grid),         intent(in),    optional :: gridList(:)
  type(ESMF_Mesh),         intent(in),    optional :: mesh
  type(ESMF_Mesh),         intent(in),    optional :: meshList(:)
  type(ESMF_LocStream),    intent(in),    optional :: locstream
  type(ESMF_LocStream),    intent(in),    optional :: locstreamList(:)
  type(ESMF_XGrid),        intent(in),    optional :: xgrid
  type(ESMF_XGrid),        intent(in),    optional :: xgridList(:)
  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(:)
  type(ESMF_Context_Flag), intent(in),    optional :: contextflag
  character(len=*),        intent(in),    optional :: name
  integer,                 intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  7.1.0r Added arguments gridList, mesh, meshList, locstream, locstreamList, xgrid, and xgridList. These arguments add support for holding references to multiple geom objects, either of the same type, or different type, in the same ESMF_GridComp object.

DESCRIPTION:

This interface creates an ESMF_GridComp object. By default, a separate VM context will be created for each component. This implies creating a new MPI communicator and allocating additional memory to manage the VM resources. When running on a large number of processors, creating a separate VM for each component could be both time and memory inefficient. If the application is sequential, i.e., each component is running on all the PETs of the global VM, it will be more efficient to use the global VM instead of creating a new one. This can be done by setting contextflag to ESMF_CONTEXT_PARENT_VM.

The return value is the new ESMF_GridComp.

The arguments are:

[grid] Associate an ESMF_Grid object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the grid object. The grid argument is mutually exclusive with the gridList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither grid nor gridList are provided, no ESMF_Grid objects are associated with the component.

[gridList] Associate a list of ESMF_Grid objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the gridList object. The gridList argument is mutually exclusive with the grid argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither grid nor gridList are provided, no ESMF_Grid objects are associated with the component.

[mesh] Associate an ESMF_Mesh object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the mesh object. The mesh argument is mutually exclusive with the meshList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither mesh nor meshList are provided, no ESMF_Mesh objects are associated with the component.

[meshList] Associate a list of ESMF_Mesh objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the meshList object. The meshList argument is mutually exclusive with the mesh argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither mesh nor meshList are provided, no ESMF_Mesh objects are associated with the component.

[locstream] Associate an ESMF_LocStream object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the locstream object. The locstream argument is mutually exclusive with the locstreamList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither locstream nor locstreamList are provided, no ESMF_LocStream objects are associated with the component.

[locstreamList] Associate a list of ESMF_LocStream objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the locstreamList object. The locstreamList argument is mutually exclusive with the locstream argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither locstream nor locstreamList are provided, no ESMF_LocStream objects are associated with the component.

[xgrid] Associate an ESMF_XGrid object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the xgrid object. The xgrid argument is mutually exclusive with the xgridList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, no ESMF_XGrid objects are associated with the component.

[xgridList] Associate a list of ESMF_XGrid objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the xgridList object. The xgridList argument is mutually exclusive with the xgrid argument. If both arguments are provided, the routine will fail, and

an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, no ESMF_XGrid objects are associated with the component.

[config] An already-created ESMF_Config object to be attached to the newly created component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the newly created component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

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

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

[contextflag] Specify the component's VM context. The default context is ESMF_CONTEXT_OWN_VM. See section **??** for a complete list of valid flags.

[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.

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

17.4.5 ESMF_GridCompDestroy - Release resources associated with a GridComp

INTERFACE:

```
recursive subroutine ESMF_GridCompDestroy(gridcomp, &
  timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)          :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(in),   optional :: timeout
  logical,             intent(out),  optional :: timeoutFlag
  integer,             intent(out),  optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Destroys an `ESMF_GridComp`, releasing the resources associated with the object.

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.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

17.4.6 ESMF_GridCompFinalize - Call the GridComp's finalize routine

INTERFACE:

```
recursive subroutine ESMF_GridCompFinalize(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user-supplied finalization routine for an `ESMF_GridComp`.

The arguments are:

gridcomp The `ESMF_GridComp` to call finalize routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

17.4.7 ESMF_GridCompGet - Get GridComp information

INTERFACE:

```
  recursive subroutine ESMF_GridCompGet(gridcomp, &
    gridIsPresent, grid, gridList, meshIsPresent, mesh, meshList, &
    locstreamIsPresent, locstream, locstreamList, xgridIsPresent, &
    xgrid, xgridList, importStateIsPresent, importState, &
    exportStateIsPresent, exportState, configIsPresent, config, &
    configFileIsPresent, configFile, clockIsPresent, clock, localPet, &
    petCount, contextflag, currentMethod, currentPhase, comptype, &
    vmIsPresent, vm, name, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),           intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                       intent(out), optional :: gridIsPresent
    type(ESMF_Grid),               intent(out), optional :: grid
    type(ESMF_Grid), allocatable,  intent(out), optional :: gridList(:)
    logical,                       intent(out), optional :: meshIsPresent
    type(ESMF_Mesh),               intent(out), optional :: mesh
    type(ESMF_Mesh), allocatable,  intent(out), optional :: meshList(:)
    logical,                       intent(out), optional :: locstreamIsPresent
    type(ESMF_LocStream),          intent(out), optional :: locstream
    type(ESMF_LocStream), allocatable, intent(out), optional :: locstreamList(:)
    logical,                       intent(out), optional :: xgridIsPresent
    type(ESMF_XGrid),              intent(out), optional :: xgrid
    type(ESMF_XGrid), allocatable, intent(out), optional :: xgridList(:)
    logical,                       intent(out), optional :: importStateIsPresent
    type(ESMF_State),              intent(out), optional :: importState
    logical,                       intent(out), optional :: exportStateIsPresent
    type(ESMF_State),              intent(out), optional :: exportState
    logical,                       intent(out), optional :: configIsPresent
    type(ESMF_Config),             intent(out), optional :: config
    logical,                       intent(out), optional :: configFileIsPresent
    character(len=*),              intent(out), optional :: configFile
    logical,                       intent(out), optional :: clockIsPresent
    type(ESMF_Clock),              intent(out), optional :: clock
    integer,                       intent(out), optional :: localPet
    integer,                       intent(out), optional :: petCount
    type(ESMF_Context_Flag),       intent(out), optional :: contextflag
    type(ESMF_Method_Flag),        intent(out), optional :: currentMethod
    integer,                       intent(out), optional :: currentPhase
    type(ESMF_CompType_Flag),      intent(out), optional :: comptype
    logical,                       intent(out), optional :: vmIsPresent
    type(ESMF_VM),                 intent(out), optional :: vm
    character(len=*),              intent(out), optional :: name
    integer,                       intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  7.1.0r Added arguments `gridList`, `meshIsPresent`, `mesh`, `meshList`, `locstreamIsPresent`, `locstream`, `locstreamList`, `xgridIsPresent`, `xgrid`, and `xgridList`. These arguments add support for accessing references to multiple geom objects, either of the same type, or different type, associated with the same `ESMF_GridComp` object.

DESCRIPTION:

Get information about an `ESMF_GridComp` object.

The arguments are:

gridcomp The `ESMF_GridComp` object being queried.

[gridIsPresent] Set to `.true.` if at least one `ESMF_Grid` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[grid] Return the `ESMF_Grid` object associated with the `gridcomp` component. If multiple `ESMF_Grid` objects are associated, return the first in the list. It is an error to query for `grid` if no `ESMF_Grid` object is associated with the `gridcomp` component. If unsure, query for `gridIsPresent` first, or use the `gridList` variant.

[gridList] Return a list of all `ESMF_Grid` objects associated with the `gridcomp` component. The size of the returned `gridList` corresponds to the number of `ESMF_Grid` objects associated. If no `ESMF_Grid` object is associated with the `gridcomp` component, the size of the returned `gridList` is zero.

[meshIsPresent] Set to `.true.` if at least one `ESMF_Mesh` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[mesh] Return the `ESMF_Mesh` object associated with the `gridcomp` component. If multiple `ESMF_Mesh` objects are associated, return the first in the list. It is an error to query for `mesh` if no `ESMF_Mesh` object is associated with the `gridcomp` component. If unsure, query for `meshIsPresent` first, or use the `meshList` variant.

[meshList] Return a list of all `ESMF_Mesh` objects associated with the `gridcomp` component. The size of the returned `meshList` corresponds to the number of `ESMF_Mesh` objects associated. If no `ESMF_Mesh` object is associated with the `gridcomp` component, the size of the returned `meshList` is zero.

[locstreamIsPresent] Set to `.true.` if at least one `ESMF_LocStream` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[locstream] Return the `ESMF_LocStream` object associated with the `gridcomp` component. If multiple `ESMF_LocStream` objects are associated, return the first in the list. It is an error to query for `locstream` if no `ESMF_Grid` object is associated with the `gridcomp` component. If unsure, query for `locstreamIsPresent` first, or use the `locstreamList` variant.

[locstreamList] Return a list of all `ESMF_LocStream` objects associated with the `gridcomp` component. The size of the returned `locstreamList` corresponds to the number of `ESMF_LocStream` objects associated. If no `ESMF_LocStream` object is associated with the `gridcomp` component, the size of the returned `locstreamList` is zero.

[xgridIsPresent] Set to `.true.` if at least one `ESMF_XGrid` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[xgrid] Return the ESMF_XGrid object associated with the gridcomp component. If multiple ESMF_XGrid objects are associated, return the first in the list. It is an error to query for xgrid if no ESMF_XGrid object is associated with the gridcomp component. If unsure, query for xgridIsPresent first, or use the xgridList variant.

[xgridList] Return a list of all ESMF_XGrid objects associated with the gridcomp component. The size of the returned xgridList corresponds to the number of ESMF_XGrid objects associated. If no ESMF_XGrid object is associated with the gridcomp component, the size of the returned xgridList is zero.

[importStateIsPresent] .true. if importState was set in GridComp object, .false. otherwise.

[importState] Return the associated import State. It is an error to query for the import State if none is associated with the GridComp. If unsure, get importStateIsPresent first to determine the status.

[exportStateIsPresent] .true. if exportState was set in GridComp object, .false. otherwise.

[exportState] Return the associated export State. It is an error to query for the export State if none is associated with the GridComp. If unsure, get exportStateIsPresent first to determine the status.

[configIsPresent] .true. if config was set in GridComp object, .false. otherwise.

[config] Return the associated Config. It is an error to query for the Config if none is associated with the GridComp. If unsure, get configIsPresent first to determine the status.

[configFileIsPresent] .true. if configFile was set in GridComp object, .false. otherwise.

[configFile] Return the associated configuration filename. It is an error to query for the configuration filename if none is associated with the GridComp. If unsure, get configFileIsPresent first to determine the status.

[clockIsPresent] .true. if clock was set in GridComp object, .false. otherwise.

[clock] Return the associated Clock. It is an error to query for the Clock if none is associated with the GridComp. If unsure, get clockIsPresent first to determine the status.

[localPet] Return the local PET id within the ESMF_GridComp object.

[petCount] Return the number of PETs in the the ESMF_GridComp object.

[contextflag] Return the ESMF_Context_Flag for this ESMF_GridComp. See section **??** for a complete list of valid flags.

[currentMethod] Return the current ESMF_Method_Flag of the ESMF_GridComp execution. See section **??** for a complete list of valid options.

[currentPhase] Return the current phase of the ESMF_GridComp execution.

[comptype] Return the Component type. See section **??** for a complete list of valid flags.

[vmIsPresent] .true. if vm was set in GridComp object, .false. otherwise.

[vm] Return the associated VM. It is an error to query for the VM if none is associated with the GridComp. If unsure, get vmIsPresent first to determine the status.

[name] Return the name of the ESMF_GridComp.

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

17.4.8 ESMF_GridCompGetInternalState - Get private data block pointer

INTERFACE:

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

ARGUMENTS:

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

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Available to be called by an `ESMF_GridComp` at any time after `ESMF_GridCompSetInternalState` has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block and the address of that block can be registered with the framework and retrieved by this call. When running multiple instantiations of an `ESMF_GridComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMF_GridCompSetInternalState` call sets the data pointer to this block, and this call retrieves the data pointer. Note that the `wrappedDataPointer` argument needs to be a derived type which contains only a pointer of the type of the data block defined by the user. When making this call the pointer needs to be unassociated. When the call returns, the pointer will now reference the original data block which was set during the previous call to `ESMF_GridCompSetInternalState`.

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

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

gridcomp An `ESMF_GridComp` object.

wrappedDataPointer A derived type (wrapper), containing only an unassociated pointer to the private data block. The framework will fill in the pointer. When this call returns, the pointer is set to the same address set during the last `ESMF_GridCompSetInternalState` call. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

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

17.4.9 ESMF_GridCompInitialize - Call the GridComp's initialize routine

INTERFACE:

```
recursive subroutine ESMF_GridCompInitialize(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user initialization routine for an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to call initialize routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

17.4.10 ESMF_GridCompIsCreated - Check whether a GridComp object has been created

INTERFACE:

```
function ESMF_GridCompIsCreated(gridcomp, rc)
```

RETURN VALUE:

```
logical :: ESMF_GridCompIsCreated
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the `gridcomp` has been created. Otherwise return `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

gridcomp `ESMF_GridComp` queried.

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

17.4.11 ESMF_GridCompIsPetLocal - Inquire if this GridComp is to execute on the calling PET

INTERFACE:

```
recursive function ESMF_GridCompIsPetLocal(gridcomp, rc)
```

RETURN VALUE:

```
logical :: ESMF_GridCompIsPetLocal
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

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

The arguments are:

gridcomp `ESMF_GridComp` queried.

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

17.4.12 ESMF_GridCompPrint - Print GridComp information

INTERFACE:

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

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

gridcomp `ESMF_GridComp` to print.

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

17.4.13 ESMF_GridCompReadRestart - Call the GridComp's read restart routine

INTERFACE:

```
recursive subroutine ESMF_GridCompReadRestart(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user read restart routine for an ESMF_GridComp.

The arguments are:

gridcomp ESMF_GridComp to call run routine for.

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

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

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

17.4.14 ESMF_GridCompRun - Call the GridComp's run routine

INTERFACE:

```
recursive subroutine ESMF_GridCompRun(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_State),     intent(inout), optional :: importState
    type(ESMF_State),     intent(inout), optional :: exportState
    type(ESMF_Clock),     intent(inout), optional :: clock
    type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
    integer,              intent(in),    optional :: phase
    integer,              intent(in),    optional :: timeout
    logical,              intent(out),   optional :: timeoutFlag
    integer,              intent(out),   optional :: userRc
    integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user run routine for an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to call run routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

17.4.15 ESMF_GridCompServiceLoop - Call the GridComp's service loop routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompServiceLoop(gridcomp, &
     importState, exportState, clock, syncflag, port, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp),  intent(inout)           :: gridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_State),     intent(inout), optional :: importState
     type(ESMF_State),     intent(inout), optional :: exportState
     type(ESMF_Clock),     intent(inout), optional :: clock
     type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
     integer,              intent(in),    optional :: port
     integer,              intent(in),    optional :: timeout
     logical,              intent(out),   optional :: timeoutFlag
     integer,              intent(out),   optional :: rc
```

DESCRIPTION:

Call the ServiceLoop routine for an `ESMF_GridComp`. This tries to establish a "component tunnel" between the *actual* Component (calling this routine) and a `dual` Component connecting to it through a matching SetServices call.

The arguments are:

gridcomp `ESMF_GridComp` to call service loop routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[port] In case a port number is provided, the "component tunnel" is established using sockets. The actual component side, i.e. the side that calls into `ESMF_GridCompServiceLoop()`, starts to listen on the specified port as the server. The valid port range is [1024, 65535]. In case the `port` argument is *not* specified, the "component tunnel" is established within the same executable using local communication methods (e.g. MPI).

[timeout] The maximum period in seconds that this call will wait for communications with the dual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. (NOTE: Currently this option is only available for socket based component tunnels.)

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

17.4.16 ESMF_GridCompSet - Set or reset information about the GridComp

INTERFACE:

```
  subroutine ESMF_GridCompSet(gridcomp, grid, gridList, &
    mesh, meshList, locstream, locstreamList, xgrid, xgridList, &
    config, configFile, clock, name, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),    intent(inout)         :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Grid),        intent(in),  optional :: grid
    type(ESMF_Grid),        intent(in),  optional :: gridList(:)
    type(ESMF_Mesh),        intent(in),  optional :: mesh
    type(ESMF_Mesh),        intent(in),  optional :: meshList(:)
    type(ESMF_LocStream),   intent(in),  optional :: locstream
    type(ESMF_LocStream),   intent(in),  optional :: locstreamList(:)
    type(ESMF_XGrid),       intent(in),  optional :: xgrid
    type(ESMF_XGrid),       intent(in),  optional :: xgridList(:)
    type(ESMF_Config),      intent(in),  optional :: config
    character(len=*),       intent(in),  optional :: configFile
    type(ESMF_Clock),       intent(in),  optional :: clock
    character(len=*),       intent(in),  optional :: name
    integer,                intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  7.1.0r Added arguments `gridList`, `mesh`, `meshList`, `locstream`, `locstreamList`, `xgrid`, and `xgridList`. These arguments add support for holding references to multiple geom objects, either of the same type, or different type, in the same `ESMF_GridComp` object.

DESCRIPTION:

Sets or resets information about an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to change.

[grid] Associate an `ESMF_Grid` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `grid` object. The `grid` argument is mutually exclusive with the `gridList` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `grid` nor `gridList` are provided, the `ESMF_Grid` association of the incoming `gridcomp` component remains unchanged.

[gridList] Associate a list of `ESMF_Grid` objects with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `gridList` object. The `gridList` argument is mutually exclusive with the `grid` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `grid` nor `gridList` are provided, the `ESMF_Grid` association of the incoming `gridcomp` component remains unchanged.

[mesh] Associate an `ESMF_Mesh` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `mesh` object. The `mesh` argument is mutually exclusive with the `meshList` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `mesh` nor `meshList` are provided, the `ESMF_Mesh` association of the incoming `gridcomp` component remains unchanged.

[meshList] Associate a list of `ESMF_Mesh` objects with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `meshList` object. The `meshList` argument is mutually exclusive with the `mesh` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `mesh` nor `meshList` are provided, the `ESMF_Mesh` association of the incoming `gridcomp` component remains unchanged.

[locstream] Associate an `ESMF_LocStream` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `locstream` object. The `locstream` argument is mutually exclusive with the `locstreamList` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `locstream` nor `locstreamList` are provided, the `ESMF_LocStream` association of the incoming `gridcomp` component remains unchanged.

[locstreamList] Associate a list of `ESMF_LocStream` objects with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `locstreamList` object. The `locstreamList` argument is mutually exclusive with the `locstream` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `locstream` nor `locstreamList` are provided, the `ESMF_LocStream` association of the incoming `gridcomp` component remains unchanged.

[xgrid] Associate an `ESMF_XGrid` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `xgrid` object. The `xgrid` argument is mutually exclusive

with the xgridList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, the ESMF_XGrid association of the incoming gridcomp component remains unchanged.

[xgridList] Associate a list of ESMF_XGrid objects with the gridcomp component. This is simply a convenience feature for the user. The ESMF library code does not access the xgridList object. The xgridList argument is mutually exclusive with the xgrid argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, the ESMF_XGrid association of the incoming gridcomp component remains unchanged.

[config] An already-created ESMF_Config object to be attached to the component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

[clock] Set the private clock for this ESMF_GridComp.

[name] Set the name of the ESMF_GridComp.

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

17.4.17 ESMF_GridCompSetEntryPoint - Set user routine as entry point for standard GridComp method

INTERFACE:

```
   recursive subroutine ESMF_GridCompSetEntryPoint(gridcomp, methodflag, &
     userRoutine, phase, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp),    intent(inout)         :: gridcomp
     type(ESMF_Method_Flag), intent(in)            :: methodflag
     interface
       subroutine userRoutine(gridcomp, importState, exportState, clock, rc)
         use ESMF_CompMod
         use ESMF_StateMod
         use ESMF_ClockMod
         implicit none
         type(ESMF_GridComp)         :: gridcomp     ! must not be optional
         type(ESMF_State)            :: importState  ! must not be optional
         type(ESMF_State)            :: exportState  ! must not be optional
         type(ESMF_Clock)            :: clock        ! must not be optional
         integer, intent(out)        :: rc           ! must not be optional
       end subroutine
     end interface
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),  optional :: phase
     integer,                intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

gridcomp An `ESMF_GridComp` object.

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

userRoutine The user-supplied subroutine to be associated for this Component `method`. Argument types, intent and order must match the interface signature, and must not have the `optional` attribute. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[phase] The `phase` number for multi-phase methods. For single phase methods the `phase` argument can be omitted. The default setting is 1.

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

17.4.18 ESMF_GridCompSetInternalState - Set private data block pointer

INTERFACE:

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

ARGUMENTS:

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

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

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.

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

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

gridcomp An ESMF_GridComp object.

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

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

17.4.19 ESMF_GridCompSetServices - Call user routine to register GridComp methods

INTERFACE:

```
recursive subroutine ESMF_GridCompSetServices(gridcomp, &
  userRoutine, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)         :: gridcomp
  interface
    subroutine userRoutine(gridcomp, rc)
      use ESMF_CompMod
      implicit none
      type(ESMF_GridComp)        :: gridcomp ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(out), optional :: userRc
  integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

gridcomp Gridded Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

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

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

17.4.20 ESMF_GridCompSetServices - Call user routine through name lookup, to register GridComp methods

INTERFACE:

```
! Private name; call using ESMF_GridCompSetServices()
recursive subroutine ESMF_GridCompSetServicesShObj(gridcomp, userRoutine, &
  sharedObj, userRoutineFound, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)          :: gridcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  logical,             intent(out), optional :: userRoutineFound
  integer,             intent(out), optional :: userRc
  integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  6.3.0r Added argument `userRoutineFound`. The new argument provides a way to test availability without causing error conditions.

DESCRIPTION:

Call into a user provided routine which is responsible for setting Component's Initialize(), Run(), and Finalize() services. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

gridcomp Gridded Component.

userRoutine Name of routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(gridcomp, rc)
    type(ESMF_GridComp)  :: gridcomp   ! must not be optional
    integer, intent(out) :: rc         ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

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

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

[userRoutineFound] Report back whether the specified `userRoutine` was found and executed, or was not available. If this argument is present, not finding the `userRoutine` will not result in returning an error in `rc`. The default is to return an error if the `userRoutine` cannot be found.

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

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

17.4.21 ESMF_GridCompSetServices - Set to serve as Dual Component for an Actual Component

INTERFACE:

```
   ! Private name; call using ESMF_GridCompSetServices()
   recursive subroutine ESMF_GridCompSetServicesComp(gridcomp, &
     actualGridcomp, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp), intent(inout)         :: gridcomp
     type(ESMF_GridComp), intent(in)            :: actualGridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set the services of a Gridded Component to serve a "dual" Component for an "actual" Component. The component tunnel is VM based.

The arguments are:

gridcomp Dual Gridded Component.

actualGridcomp Actual Gridded Component.

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

17.4.22 ESMF_GridCompSetServices - Set to serve as Dual Component for an Actual Component through sockets

INTERFACE:

```
   ! Private name; call using ESMF_GridCompSetServices()
   recursive subroutine ESMF_GridCompSetServicesSock(gridcomp, port, &
     server, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp), intent(inout)         :: gridcomp
     integer,             intent(in)            :: port
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),    intent(in),  optional :: server
     integer,             intent(in),  optional :: timeout
     logical,             intent(out), optional :: timeoutFlag
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set the services of a Gridded Component to serve a "dual" Component for an "actual" Component. The component tunnel is socket based.

The arguments are:

gridcomp Dual Gridded Component.

port Port number under which the actual component is being served. The valid port range is [1024, 65535].

[server] Server name where the actual component is being served. The default, i.e. if the `server` argument was not provided, is `localhost`.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

17.4.23 ESMF_GridCompSetVM - Call user routine to set GridComp VM properties

INTERFACE:

```
recursive subroutine ESMF_GridCompSetVM(gridcomp, userRoutine, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)         :: gridcomp
  interface
    subroutine userRoutine(gridcomp, rc)
      use ESMF_CompMod
      implicit none
      type(ESMF_GridComp)        :: gridcomp ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(out), optional :: userRc
  integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties.

The arguments are:

gridcomp Gridded Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

The subroutine, when called by the framework, is expected to use any of the `ESMF_GridCompSetVMxxx()` methods to set the properties of the VM associated with the Gridded Component.

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

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

17.4.24 ESMF_GridCompSetVM - Call user routine through name lookup, to set GridComp VM properties

INTERFACE:

```
! Private name; call using ESMF_GridCompSetVM()
recursive subroutine ESMF_GridCompSetVMShObj(gridcomp, userRoutine, &
  sharedObj, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)          :: gridcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  integer,             intent(out), optional :: userRc
  integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

gridcomp Gridded Component.

userRoutine Routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(gridcomp, rc)
    type(ESMF_GridComp)  :: gridcomp    ! must not be optional
    integer, intent(out) :: rc          ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

The subroutine, when called by the framework, is expected to use any of the `ESMF_GridCompSetVMxxx()` methods to set the properties of the VM associated with the Gridded Component.

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

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

17.4.25 ESMF_GridCompSetVMMaxPEs - Associate PEs with PETs in GridComp VM

INTERFACE:

```
subroutine ESMF_GridCompSetVMMaxPEs(gridcomp, &
  maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
  pthreadMinStackSize, openMpHandling, openMpNumThreads, &
  forceChildPthreads, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)         :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(in),  optional :: maxPeCountPerPet
  integer,             intent(in),  optional :: prefIntraProcess
  integer,             intent(in),  optional :: prefIntraSsi
  integer,             intent(in),  optional :: prefInterSsi
  integer,             intent(in),  optional :: pthreadMinStackSize
  character(*),        intent(in),  optional :: openMpHandling
  integer,             intent(in),  optional :: openMpNumThreads
  logical,             intent(in),  optional :: forceChildPthreads
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the ESMF_VM for this ESMF_GridComp. Attempts to associate up to maxPeCountPerPet PEs with each PET. Only PEs that are located on the same single system image (SSI) can be associated with the same PET. Within this constraint the call tries to get as close as possible to the number specified by maxPeCountPerPet.

The other constraint to this call is that the number of PEs is preserved. This means that the child Component in the end is associated with as many PEs as the parent Component provided to the child. The number of child PETs however is adjusted according to the above rule.

The typical use of ESMF_GridCompSetVMMaxPEs() is to allocate multiple PEs per PET in a Component for user-level threading, e.g. OpenMP.

The arguments are:

gridcomp ESMF_GridComp to set the ESMF_VM for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if forceChildPthreads is .true., child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical OMP_STACKSIZE environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by OMP_STACKSIZE. It is the master's stack that can be sized via the pthreadMinStackSize argument, and a large enough size is often critical.

When pthreadMinStackSize is absent, the default is to use the system default set by the limit or ulimit command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and pthreadMinStackSize must be used to allocate sufficient stack space.

[openMpHandling] Handling of OpenMP threads. Supported options are:

- "none" - OpenMP handling is completely left to the user.
- "set" - ESMF uses the omp_set_num_threads() API to set the number of OpenMP threads in each team.
- "init" - ESMF sets the number of OpenMP threads in each team, and triggers the instantiation of the team.
- "pin" (default) - ESMF sets the number of OpenMP threads in each team, triggers the instantiation of the team, and pins each OpenMP thread to the corresponding PE.

[openMpNumThreads] Number of OpenMP threads in each OpenMP thread team. This can be any positive number. By default, or if openMpNumThreads is negative, each PET sets the number of OpenMP threads to its local peCount.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

17.4.26 ESMF_GridCompSetVMMaxThreads - Set multi-threaded PETs in GridComp VM

INTERFACE:

```
   subroutine ESMF_GridCompSetVMMaxThreads(gridcomp, &
     maxPetCountPerVas, prefIntraProcess, prefIntraSsi, prefInterSsi, &
     pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp), intent(inout)         :: gridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(in),  optional :: maxPetCountPerVas
     integer,             intent(in),  optional :: prefIntraProcess
     integer,             intent(in),  optional :: prefIntraSsi
     integer,             intent(in),  optional :: prefInterSsi
     integer,             intent(in),  optional :: pthreadMinStackSize
     logical,             intent(in),  optional :: forceChildPthreads
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the `ESMF_VM` for this `ESMF_GridComp`. Attempts to provide `maxPetCountPerVas` threaded PETs in each virtual address space (VAS). Only as many threaded PETs as there are PEs located on the single system image (SSI) can be associated with the VAS. Within this constraint the call tries to get as close as possible to the number specified by `maxPetCountPerVas`.

The other constraint to this call is that the number of PETs is preserved. This means that the child Component in the end is associated with as many PETs as the parent Component provided to the child. The threading level of the child PETs however is adjusted according to the above rule.

The typical use of `ESMF_GridCompSetVMMaxThreads()` is to run a Component multi-threaded with groups of PETs executing within a common virtual address space.

The arguments are:

gridcomp `ESMF_GridComp` to set the `ESMF_VM` for.

[maxPetCountPerVas] Maximum number of threaded PETs in each virtual address space (VAS). Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When `pthreadMinStackSize` is absent, the default is to use the system default set by the `limit` or `ulimit` command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and `pthreadMinStackSize` must be used to allocate sufficient stack space.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

17.4.27 ESMF_GridCompSetVMMinThreads - Set a reduced threading level in GridComp VM

INTERFACE:

```
  subroutine ESMF_GridCompSetVMMinThreads(gridcomp, &
    maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
    pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp), intent(inout)         :: gridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(in),  optional :: maxPeCountPerPet
    integer,             intent(in),  optional :: prefIntraProcess
    integer,             intent(in),  optional :: prefIntraSsi
    integer,             intent(in),  optional :: prefInterSsi
    integer,             intent(in),  optional :: pthreadMinStackSize
    logical,             intent(in),  optional :: forceChildPthreads
    integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the `ESMF_VM` for this `ESMF_GridComp`. Reduces the number of threaded PETs in each VAS. The `max` argument may be specified to limit the maximum number of PEs that a single PET can be associated with.

Several constraints apply: 1) the number of PEs cannot change, 2) PEs cannot migrate between single system images (SSIs), 3) the number of PETs cannot increase, only decrease, 4) PETs cannot migrate between virtual address spaces (VASs), nor can VASs migrate between SSIs.

The typical use of `ESMF_GridCompSetVMMinThreads()` is to run a Component across a set of single-threaded PETs.

The arguments are:

gridcomp `ESMF_GridComp` to set the `ESMF_VM` for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When `pthreadMinStackSize` is absent, the default is to use the system default set by the `limit` or `ulimit` command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and `pthreadMinStackSize` must be used to allocate sufficient stack space.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

17.4.28 ESMF_GridCompValidate - Check validity of a GridComp

INTERFACE:

```
subroutine ESMF_GridCompValidate(gridcomp, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

gridcomp `ESMF_GridComp` to validate.

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

17.4.29 ESMF_GridCompWait - Wait for a GridComp to return

INTERFACE:

```
  subroutine ESMF_GridCompWait(gridcomp, syncflag, &
    timeout, timeoutFlag, userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),  intent(inout)         :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Sync_Flag), intent(in),  optional :: syncflag
    integer,              intent(in),  optional :: timeout
    logical,              intent(out), optional :: timeoutFlag
    integer,              intent(out), optional :: userRc
    integer,              intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

When executing asynchronously, wait for an ESMF_GridComp to return.

The arguments are:

gridcomp ESMF_GridComp to wait for.

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

[timeout] The maximum period in seconds the actual component is allowed to execute a previously invoked component method before it must communicate back to the dual component. If the actual component does not communicate back in the specified time, a timeout condition is raised on the dual side (this side). The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

17.4.30 ESMF_GridCompWriteRestart - Call the GridComp's write restart routine

INTERFACE:

```
recursive subroutine ESMF_GridCompWriteRestart(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user write restart routine for an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to call run routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple sub-routines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

18 CplComp Class

18.1 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 17.1).

A Coupler Component, or ESMF_CplComp, arranges and executes the data transformations between the Gridded Components. Ideally, Coupler Components should contain all the information about inter-component communication for an application. This enables the Gridded Components in the application to be used in multiple contexts; that is, used in different coupled configurations without changes to their source code. For example, the same atmosphere might in one case be coupled to an ocean in a hurricane prediction model, and to a data assimilation system for numerical weather prediction in another. A single Coupler Component can couple two or more Gridded Components.

Like Gridded Components, Coupler Components have two parts, one that is provided by the user and another that is part of the framework. The user-written portion of the software is the coupling code necessary for a particular exchange between Gridded Components. This portion of the Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.

The term "user-written" is somewhat misleading here, since within a Coupler Component the user can leverage ESMF infrastructure software for regridding, redistribution, lower-level communications, calendar management, and other functions. However, ESMF is unlikely to offer all the software necessary to customize a data transfer between Gridded Components. For instance, ESMF does not currently offer tools for unit tranformations or time averaging operations, so users must manage those operations themselves.

The second part of a Coupler Component is the 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. [2]

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

18.2 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.

18.2.1 Implement a user-code SetServices routine

Every ESMF_CplComp is required to provide and document a public set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an ESMF_CplComp as the first argument, and an integer return code as the second. Both arguments are required and must *not* be declared as optional. If an intent is specified in the interface it must be intent(inout) for the first and intent(out) for the second argument.

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

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

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

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

```
! ESMF Framework module
use ESMF
implicit none
public CPL_SetServices

contains

subroutine CPL_SetServices(comp, rc)
  type(ESMF_CplComp)    :: comp   ! must not be optional
  integer, intent(out)  :: rc     ! must not be optional

  ! Set the entry points for standard ESMF Component methods
  call ESMF_CplCompSetEntryPoint(comp, ESMF_METHOD_INITIALIZE, &
                       userRoutine=CPL_Init, rc=rc)
  call ESMF_CplCompSetEntryPoint(comp, ESMF_METHOD_RUN, &
                       userRoutine=CPL_Run, rc=rc)
  call ESMF_CplCompSetEntryPoint(comp, ESMF_METHOD_FINALIZE, &
                       userRoutine=CPL_Final, rc=rc)

  rc = ESMF_SUCCESS
end subroutine
```

18.2.2 Implement 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 interface shown below. Arguments must not be declared as 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                ! must not be optional
  type(ESMF_State)      :: importState         ! must not be optional
  type(ESMF_State)      :: exportState         ! must not be optional
  type(ESMF_Clock)      :: clock               ! must not be optional
  integer, intent(out)  :: rc                  ! must not be optional

  print *, "Coupler Init starting"

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

  rc = ESMF_SUCCESS

  print *, "Coupler Init returning"
```

```
end subroutine CPL_Init
```

18.2.3 Implement 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 interface shown below. Arguments must not be declared as 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              ! must not be optional
  type(ESMF_State)      :: importState       ! must not be optional
  type(ESMF_State)      :: exportState       ! must not be optional
  type(ESMF_Clock)      :: clock             ! must not be optional
  integer, intent(out)  :: rc                ! must not be optional

  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
```

18.2.4 Implement 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 component writer must supply a subroutine with the exact interface shown below. Arguments must not be declared as optional, and the types and order must match.

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                ! must not be optional
  type(ESMF_State)      :: importState         ! must not be optional
```

```
  type(ESMF_State)      :: exportState         ! must not be optional
  type(ESMF_Clock)      :: clock               ! must not be optional
  integer, intent(out)  :: rc                  ! must not be optional

  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
```

18.2.5 Implement a user-code `SetVM` routine

Every `ESMF_CplComp` can optionally provide and document a public set vm routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_CplComp` as the first argument, and an integer return code as the second. Both arguments are required and must *not* be declared as `optional`. If an intent is specified in the interface it must be `intent(inout)` for the first and `intent(out)` for the second argument.

The set vm routine is the only place where the child component can use the `ESMF_CplCompSetVMMaxPEs()`, or `ESMF_CplCompSetVMMaxThreads()`, or `ESMF_CplCompSetVMMinThreads()` call to modify aspects of its own VM.

A component's VM is started up right before its set services routine is entered. `ESMF_CplCompSetVM()` is executing in the parent VM, and must be called *before* `ESMF_CplCompSetServices()`.

```
subroutine GComp_SetVM(comp, rc)
  type(ESMF_CplComp)    :: comp   ! must not be optional
  integer, intent(out)  :: rc     ! must not be optional

  type(ESMF_VM) :: vm
  logical :: pthreadsEnabled

  ! Test for Pthread support, all SetVM calls require it
  call ESMF_VMGetGlobal(vm, rc=rc)
  call ESMF_VMGet(vm, pthreadsEnabledFlag=pthreadsEnabled, rc=rc)

  if (pthreadsEnabled) then
    ! run PETs single-threaded
    call ESMF_CplCompSetVMMinThreads(comp, rc=rc)
  endif

  rc = ESMF_SUCCESS

end subroutine

end module ESMF_CouplerEx
```

18.3 Restrictions and Future Work

1. **No optional arguments.** User-written routines called by SetServices, and registered for Initialize, Run and Finalize, *must not* declare any of the arguments as optional.
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.
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.

18.4 Class API

18.4.1 ESMF_CplCompAssignment(=) - CplComp assignment

INTERFACE:

```
interface assignment(=)
cplcomp1 = cplcomp2
```

ARGUMENTS:

```
type(ESMF_CplComp) :: cplcomp1
type(ESMF_CplComp) :: cplcomp2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Assign cplcomp1 as an alias to the same ESMF CplComp object in memory as cplcomp2. If cplcomp2 is invalid, then cplcomp1 will be equally invalid after the assignment.

The arguments are:

cplcomp1 The `ESMF_CplComp` object on the left hand side of the assignment.

cplcomp2 The `ESMF_CplComp` object on the right hand side of the assignment.

18.4.2 ESMF_CplCompOperator(==) - CplComp equality operator

INTERFACE:

```
interface operator(==)
  if (cplcomp1 == cplcomp2) then ... endif
            OR
  result = (cplcomp1 == cplcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in) :: cplcomp1
type(ESMF_CplComp), intent(in) :: cplcomp2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Test whether cplcomp1 and cplcomp2 are valid aliases to the same ESMF CplComp object in memory. For a more general comparison of two ESMF CplComps, going beyond the simple alias test, the ESMF_CplCompMatch() function (not yet implemented) must be used.

The arguments are:

cplcomp1 The `ESMF_CplComp` object on the left hand side of the equality operation.

cplcomp2 The `ESMF_CplComp` object on the right hand side of the equality operation.

18.4.3 ESMF_CplCompOperator(/=) - CplComp not equal operator

INTERFACE:

```
interface operator(/=)
  if (cplcomp1 /= cplcomp2) then ... endif
            OR
  result = (cplcomp1 /= cplcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in) :: cplcomp1
type(ESMF_CplComp), intent(in) :: cplcomp2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Test whether cplcomp1 and cplcomp2 are *not* valid aliases to the same ESMF CplComp object in memory. For a more general comparison of two ESMF CplComps, going beyond the simple alias test, the ESMF_CplCompMatch() function (not yet implemented) must be used.

The arguments are:

cplcomp1 The `ESMF_CplComp` object on the left hand side of the non-equality operation.

cplcomp2 The `ESMF_CplComp` object on the right hand side of the non-equality operation.

18.4.4 ESMF_CplCompCreate - Create a CplComp

INTERFACE:

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

RETURN VALUE:

```
type(ESMF_CplComp) :: ESMF_CplCompCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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(:)
    type(ESMF_Context_Flag), intent(in),  optional :: contextflag
    character(len=*),        intent(in),  optional :: name
    integer,                 intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

This interface creates an ESMF_CplComp object. By default, a separate VM context will be created for each component. This implies creating a new MPI communicator and allocating additional memory to manage the VM resources. When running on a large number of processors, creating a separate VM for each component could be both time and memory inefficient. If the application is sequential, i.e., each component is running on all the PETs of the global VM, it will be more efficient to use the global VM instead of creating a new one. This can be done by setting contextflag to ESMF_CONTEXT_PARENT_VM.

The return value is the new ESMF_CplComp.

The arguments are:

[config] An already-created ESMF_Config object to be attached to the newly created component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the newly created component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

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

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

[contextflag] Specify the component's VM context. The default context is ESMF_CONTEXT_OWN_VM. See section **??** for a complete list of valid flags.

[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.

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

18.4.5 ESMF_CplCompDestroy - Release resources associated with a CplComp

INTERFACE:

```
recursive subroutine ESMF_CplCompDestroy(cplcomp, &
  timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(inout)          :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(in),   optional :: timeout
    logical,            intent(out),  optional :: timeoutFlag
    integer,            intent(out),  optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Destroys an `ESMF_CplComp`, releasing the resources associated with the object.

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.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

18.4.6 ESMF_CplCompFinalize - Call the CplComp's finalize routine

INTERFACE:

```
recursive subroutine ESMF_CplCompFinalize(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user-supplied finalization routine for an `ESMF_CplComp`.

The arguments are:

cplcomp The `ESMF_CplComp` to call finalize routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

18.4.7 ESMF_CplCompGet - Get CplComp information

INTERFACE:

```
  subroutine ESMF_CplCompGet(cplcomp, configIsPresent, config, &
    configFileIsPresent, configFile, clockIsPresent, clock, localPet, &
    petCount, contextflag, currentMethod, currentPhase, vmIsPresent, &
    vm, name, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),      intent(in)            :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                 intent(out), optional :: configIsPresent
    type(ESMF_Config),       intent(out), optional :: config
    logical,                 intent(out), optional :: configFileIsPresent
    character(len=*),        intent(out), optional :: configFile
    logical,                 intent(out), optional :: clockIsPresent
    type(ESMF_Clock),        intent(out), optional :: clock
    integer,                 intent(out), optional :: localPet
    integer,                 intent(out), optional :: petCount
    type(ESMF_Context_Flag), intent(out), optional :: contextflag
    type(ESMF_Method_Flag),  intent(out), optional :: currentMethod
    integer,                 intent(out), optional :: currentPhase
    logical,                 intent(out), optional :: vmIsPresent
    type(ESMF_VM),           intent(out), optional :: vm
    character(len=*),        intent(out), optional :: name
    integer,                 intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Get information about an `ESMF_CplComp` object.

The arguments are:

cplcomp The `ESMF_CplComp` object being queried.

[configIsPresent] `.true.` if `config` was set in CplComp object, `.false.` otherwise.

[config] Return the associated Config. It is an error to query for the Config if none is associated with the CplComp. If unsure, get `configIsPresent` first to determine the status.

[configFileIsPresent] `.true.` if `configFile` was set in CplComp object, `.false.` otherwise.

[configFile] Return the associated configuration filename. It is an error to query for the configuration filename if none is associated with the CplComp. If unsure, get `configFileIsPresent` first to determine the status.

[clockIsPresent] `.true.` if `clock` was set in CplComp object, `.false.` otherwise.

[clock] Return the associated Clock. It is an error to query for the Clock if none is associated with the CplComp. If unsure, get `clockIsPresent` first to determine the status.

[localPet] Return the local PET id within the `ESMF_CplComp` object.

[petCount] Return the number of PETs in the the `ESMF_CplComp` object.

[contextflag] Return the `ESMF_Context_Flag` for this `ESMF_CplComp`. See section **??** for a complete list of valid flags.

[currentMethod] Return the current `ESMF_Method_Flag` of the `ESMF_CplComp` execution. See section **??** for a complete list of valid options.

[currentPhase] Return the current `phase` of the `ESMF_CplComp` execution.

[vmIsPresent] `.true.` if `vm` was set in CplComp object, `.false.` otherwise.

[vm] Return the associated VM. It is an error to query for the VM if none is associated with the CplComp. If unsure, get `vmIsPresent` first to determine the status.

[name] Return the name of the `ESMF_CplComp`.

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

18.4.8 ESMF_CplCompGetInternalState - Get private data block pointer

INTERFACE:

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

ARGUMENTS:

```
type(ESMF_CplComp)              :: cplcomp
type(wrapper)                   :: wrappedDataPointer
integer,            intent(out) :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

When the call returns, the pointer will now reference the original data block which was set during the previous call to ESMF_CplCompSetInternalState.

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

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

cplcomp An ESMF_CplComp object.

wrappedDataPointer A derived type (wrapper), containing only an unassociated pointer to the private data block. The framework will fill in the pointer. When this call returns, the pointer is set to the same address set during the last ESMF_CplCompSetInternalState call. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

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

18.4.9 ESMF_CplCompInitialize - Call the CplComp's initialize routine

INTERFACE:

```
  recursive subroutine ESMF_CplCompInitialize(cplcomp, &
    importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
    userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),   intent(inout)           :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_State),     intent(inout), optional :: importState
    type(ESMF_State),     intent(inout), optional :: exportState
    type(ESMF_Clock),     intent(inout), optional :: clock
    type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
    integer,              intent(in),    optional :: phase
    integer,              intent(in),    optional :: timeout
    logical,              intent(out),   optional :: timeoutFlag
    integer,              intent(out),   optional :: userRc
    integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user initialization routine for an `ESMF_CplComp`.

The arguments are:

cplcomp `ESMF_CplComp` to call initialize routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

18.4.10 ESMF_CplCompIsCreated - Check whether a CplComp object has been created

INTERFACE:

```
function ESMF_CplCompIsCreated(cplcomp, rc)
```

RETURN VALUE:

```
    logical :: ESMF_CplCompIsCreated
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(in)            :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the `cplcomp` has been created. Otherwise return `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

cplcomp `ESMF_CplComp` queried.

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

18.4.11 ESMF_CplCompIsPetLocal - Inquire if this CplComp is to execute on the calling PET

INTERFACE:

```
  recursive function ESMF_CplCompIsPetLocal(cplcomp, rc)
```

RETURN VALUE:

```
    logical :: ESMF_CplCompIsPetLocal
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(in)            :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

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

The arguments are:

cplcomp `ESMF_CplComp` queried.

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

18.4.12 ESMF_CplCompPrint - Print CplComp information

INTERFACE:

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

ARGUMENTS:

```
    type(ESMF_CplComp), intent(in)            :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

cplcomp `ESMF_CplComp` to print.

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

18.4.13 ESMF_CplCompReadRestart – Call the CplComp's read restart routine

INTERFACE:

```
recursive subroutine ESMF_CplCompReadRestart(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_State),     intent(inout), optional :: importState
    type(ESMF_State),     intent(inout), optional :: exportState
    type(ESMF_Clock),     intent(inout), optional :: clock
    type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
    integer,              intent(in),    optional :: phase
    integer,              intent(in),    optional :: timeout
    logical,              intent(out),   optional :: timeoutFlag
    integer,              intent(out),   optional :: userRc
    integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user read restart routine for an `ESMF_CplComp`.

The arguments are:

cplcomp `ESMF_CplComp` to call run routine for.

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

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

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

18.4.14 ESMF_CplCompRun - Call the CplComp's run routine

INTERFACE:

```
recursive subroutine ESMF_CplCompRun(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user run routine for an `ESMF_CplComp`.

The arguments are:

cplcomp ESMF_CplComp to call run routine for.

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

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

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

18.4.15 ESMF_CplCompServiceLoop - Call the CplComp's service loop routine

INTERFACE:

```
recursive subroutine ESMF_CplCompServiceLoop(cplcomp, &
  importState, exportState, clock, syncflag, port, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: port
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: rc
```

DESCRIPTION:

Call the ServiceLoop routine for an ESMF_CplComp. This tries to establish a "component tunnel" between the *actual* Component (calling this routine) and a dual Component connecting to it through a matching SetServices call.

The arguments are:

cplcomp ESMF_CplComp to call service loop routine for.

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

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

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[port] In case a port number is provided, the "component tunnel" is established using sockets. The actual component side, i.e. the side that calls into ESMF_CplCompServiceLoop(), starts to listen on the specified port as the server. The valid port range is [1024, 65535]. In case the port argument is *not* specified, the "component tunnel" is established within the same executable using local communication methods (e.g. MPI).

[timeout] The maximum period in seconds that this call will wait for communications with the dual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. (NOTE: Currently this option is only available for socket based component tunnels.)

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

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

18.4.16 ESMF_CplCompSet - Set or reset information about the CplComp

INTERFACE:

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

ARGUMENTS:

```
    type(ESMF_CplComp), intent(inout)         :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Config),  intent(in),  optional :: config
    character(len=*),   intent(in),  optional :: configFile
```

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

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Sets or resets information about an ESMF_CplComp.

The arguments are:

cplcomp ESMF_CplComp to change.

[name] Set the name of the ESMF_CplComp.

[config] An already-created ESMF_Config object to be attached to the component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

[clock] Set the private clock for this ESMF_CplComp.

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

18.4.17 ESMF_CplCompSetEntryPoint - Set user routine as entry point for standard Component method

INTERFACE:

```
recursive subroutine ESMF_CplCompSetEntryPoint(cplcomp, methodflag, &
  userRoutine, phase, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),     intent(inout)         :: cplcomp
type(ESMF_Method_Flag), intent(in)            :: methodflag
interface
  subroutine userRoutine(cplcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type(ESMF_CplComp)          :: cplcomp      ! must not be optional
    type(ESMF_State)            :: importState  ! must not be optional
```

```
        type(ESMF_State)            :: exportState  ! must not be optional
        type(ESMF_Clock)            :: clock        ! must not be optional
        integer, intent(out)        :: rc           ! must not be optional
      end subroutine
    end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                intent(in),  optional :: phase
    integer,                intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

cplcomp An `ESMF_CplComp` object.

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

userRoutine The user-supplied subroutine to be associated for this `methodflag`. The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[phase] The `phase` number for multi-phase methods. For single phase methods the `phase` argument can be omitted. The default setting is 1.

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

18.4.18 ESMF_CplCompSetInternalState - Set private data block pointer

INTERFACE:

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

ARGUMENTS:

```
    type(ESMF_CplComp)              :: cplcomp
    type(wrapper)                   :: wrappedDataPointer
    integer,            intent(out) :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

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.

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

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

cplcomp An ESMF_CplComp object.

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

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

18.4.19 ESMF_CplCompSetServices - Call user routine to register CplComp methods

INTERFACE:

```
recursive subroutine ESMF_CplCompSetServices(cplcomp, userRoutine, &
   userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(inout)         :: cplcomp
    interface
      subroutine userRoutine(cplcomp, rc)
        use ESMF_CompMod
        implicit none
        type(ESMF_CplComp)         :: cplcomp  ! must not be optional
        integer, intent(out)       :: rc       ! must not be optional
      end subroutine
    end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(out), optional :: userRc
    integer,            intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

cplcomp Coupler Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

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

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

18.4.20 ESMF_CplCompSetServices - Call user routine through name lookup, to register CplComp methods

INTERFACE:

```
! Private name; call using ESMF_CplCompSetServices()
recursive subroutine ESMF_CplCompSetServicesShObj(cplcomp, userRoutine, &
  sharedObj, userRoutineFound, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(inout)          :: cplcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  logical,             intent(out), optional :: userRoutineFound
  integer,             intent(out), optional :: userRc
  integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  6.3.0r Added argument `userRoutineFound`. The new argument provides a way to test availability without causing error conditions.

DESCRIPTION:

Call into a user provided routine which is responsible for setting Component's Initialize(), Run(), and Finalize() services. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

cplcomp Coupler Component.

userRoutine Name of routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(cplcomp, rc)
    type(ESMF_CplComp)   :: cplcomp    ! must not be optional
    integer, intent(out) :: rc         ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

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

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

[userRoutineFound] Report back whether the specified `userRoutine` was found and executed, or was not available. If this argument is present, not finding the `userRoutine` will not result in returning an error in `rc`. The default is to return an error if the `userRoutine` cannot be found.

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

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

18.4.21 ESMF_CplCompSetServices - Set to serve as Dual Component for an Actual Component

INTERFACE:

```
   ! Private name; call using ESMF_CplCompSetServices()
   recursive subroutine ESMF_CplCompSetServicesComp(cplcomp, &
     actualCplcomp, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp), intent(inout)         :: cplcomp
     type(ESMF_CplComp), intent(in)            :: actualCplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(out), optional :: rc
```

DESCRIPTION:

Set the services of a Coupler Component to serve a "dual" Component for an "actual" Component. The component tunnel is VM based.

The arguments are:

cplcomp Dual Coupler Component.

actualCplcomp Actual Coupler Component.

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

18.4.22 ESMF_CplCompSetServices - Set to serve as Dual Component for an Actual Component through sockets

INTERFACE:

```
   ! Private name; call using ESMF_CplCompSetServices()
   recursive subroutine ESMF_CplCompSetServicesSock(cplcomp, port, &
     server, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp), intent(inout)         :: cplcomp
     integer,            intent(in)            :: port
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),   intent(in),  optional :: server
     integer,            intent(in),  optional :: timeout
     logical,            intent(out), optional :: timeoutFlag
     integer,            intent(out), optional :: rc
```

DESCRIPTION:

Set the services of a Coupler Component to serve a "dual" Component for an "actual" Component. The component tunnel is socket based.

The arguments are:

cplcomp Dual Coupler Component.

port Port number under which the actual component is being served. The valid port range is [1024, 65535].

[server] Server name where the actual component is being served. The default, i.e. if the `server` argument was not provided, is `localhost`.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

18.4.23 ESMF_CplCompSetVM - Call user routine to set CplComp VM properties

INTERFACE:

```
recursive subroutine ESMF_CplCompSetVM(cplcomp, userRoutine, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp), intent(inout)         :: cplcomp
  interface
    subroutine userRoutine(cplcomp, rc)
      use ESMF_CompMod
      implicit none
      type(ESMF_CplComp)         :: cplcomp  ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,            intent(out), optional :: userRc
  integer,            intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties.

The arguments are:

cplcomp Coupler Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

The subroutine, when called by the framework, is expected to use any of the `ESMF_CplCompSetVMxxx()` methods to set the properties of the VM associated with the Coupler Component.

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

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

18.4.24 ESMF_CplCompSetVM - Call user routine through name lookup, to set CplComp VM properties

INTERFACE:

```
! Private name; call using ESMF_CplCompSetVM()
recursive subroutine ESMF_CplCompSetVMShObj(cplcomp, userRoutine, &
  sharedObj, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(inout)          :: cplcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  integer,             intent(out), optional :: userRc
  integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

cplcomp Coupler Component.

userRoutine Routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(cplcomp, rc)
    type(ESMF_CplComp)   :: cplcomp     ! must not be optional
    integer, intent(out) :: rc          ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

The subroutine, when called by the framework, is expected to use any of the `ESMF_CplCompSetVMxxx()` methods to set the properties of the VM associated with the Coupler Component.

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

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

18.4.25 ESMF_CplCompSetVMMaxPEs - Associate PEs with PETs in CplComp VM

INTERFACE:

```
subroutine ESMF_CplCompSetVMMaxPEs(cplcomp, &
  maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
  pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(inout)         :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(in),  optional :: maxPeCountPerPet
  integer,             intent(in),  optional :: prefIntraProcess
  integer,             intent(in),  optional :: prefIntraSsi
  integer,             intent(in),  optional :: prefInterSsi
  integer,             intent(in),  optional :: pthreadMinStackSize
  logical,             intent(in),  optional :: forceChildPthreads
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the ESMF_VM for this ESMF_CplComp. Attempts to associate up to maxPeCountPerPet PEs with each PET. Only PEs that are located on the same single system image (SSI) can be associated with the same PET. Within this constraint the call tries to get as close as possible to the number specified by maxPeCountPerPet.

The other constraint to this call is that the number of PEs is preserved. This means that the child Component in the end is associated with as many PEs as the parent Component provided to the child. The number of child PETs however is adjusted according to the above rule.

The typical use of ESMF_CplCompSetVMMaxPEs() is to allocate multiple PEs per PET in a Component for user-level threading, e.g. OpenMP.

The arguments are:

cplcomp ESMF_CplComp to set the ESMF_VM for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if forceChildPthreads is .true., child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical OMP_STACKSIZE environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by OMP_STACKSIZE. It is the master's stack that can be sized via the pthreadMinStackSize argument, and a large enough size is often critical.

When pthreadMinStackSize is absent, the default is to use the system default set by the limit or ulimit command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and pthreadMinStackSize must be used to allocate sufficient stack space.

[forceChildPthreads] For .true., force each child PET to execute in its own Pthread. By default, .false., single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

18.4.26 ESMF_CplCompSetVMMaxThreads - Set multi-threaded PETs in CplComp VM

INTERFACE:

```
  subroutine ESMF_CplCompSetVMMaxThreads(cplcomp, &
    maxPetCountPerVas, prefIntraProcess, prefIntraSsi, prefInterSsi, &
    pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),  intent(inout)         :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(in),  optional :: maxPetCountPerVas
    integer,             intent(in),  optional :: prefIntraProcess
    integer,             intent(in),  optional :: prefIntraSsi
    integer,             intent(in),  optional :: prefInterSsi
    integer,             intent(in),  optional :: pthreadMinStackSize
    logical,             intent(in),  optional :: forceChildPthreads
    integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the `ESMF_VM` for this `ESMF_CplComp`. Attempts to provide `maxPetCountPerVas` threaded PETs in each virtual address space (VAS). Only as many threaded PETs as there are PEs located on the single system image (SSI) can be associated with the VAS. Within this constraint the call tries to get as close as possible to the number specified by `maxPetCountPerVas`.

The other constraint to this call is that the number of PETs is preserved. This means that the child Component in the end is associated with as many PETs as the parent Component provided to the child. The threading level of the child PETs however is adjusted according to the above rule.

The typical use of `ESMF_CplCompSetVMMaxThreads()` is to run a Component multi-threaded with groups of PETs executing within a common virtual address space.

The arguments are:

cplcomp `ESMF_CplComp` to set the `ESMF_VM` for.

[maxPetCountPerVas] Maximum number of threaded PETs in each virtual address space (VAS). Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When pthreadMinStackSize is absent, the default is to use the system default set by the limit or ulimit command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and pthreadMinStackSize must be used to allocate sufficient stack space.

[forceChildPthreads] For .true., force each child PET to execute in its own Pthread. By default, .false., single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

18.4.27 ESMF_CplCompSetVMMinThreads - Set a reduced threading level in CplComp VM

INTERFACE:

```
   subroutine ESMF_CplCompSetVMMinThreads(cplcomp, &
     maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
     pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp),  intent(inout)         :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(in),  optional :: maxPeCountPerPet
     integer,             intent(in),  optional :: prefIntraProcess
     integer,             intent(in),  optional :: prefIntraSsi
     integer,             intent(in),  optional :: prefInterSsi
     integer,             intent(in),  optional :: pthreadMinStackSize
     logical,             intent(in),  optional :: forceChildPthreads
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the ESMF_VM for this ESMF_CplComp. Reduces the number of threaded PETs in each VAS. The max argument may be specified to limit the maximum number of PEs that a single PET can be associated with.

Several constraints apply: 1) the number of PEs cannot change, 2) PEs cannot migrate between single system images (SSIs), 3) the number of PETs cannot increase, only decrease, 4) PETs cannot migrate between virtual address spaces (VASs), nor can VASs migrate between SSIs.

The typical use of ESMF_CplCompSetVMMinThreads() is to run a Component across a set of single-threaded PETs.

The arguments are:

cplcomp ESMF_CplComp to set the ESMF_VM for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When `pthreadMinStackSize` is absent, the default is to use the system default set by the `limit` or `ulimit` command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and `pthreadMinStackSize` must be used to allocate sufficient stack space.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

18.4.28 ESMF_CplCompValidate – Ensure the CplComp is internally consistent

INTERFACE:

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

ARGUMENTS:

```
     type(ESMF_CplComp), intent(in)             :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

cplcomp ESMF_CplComp to validate.

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

18.4.29 ESMF_CplCompWait - Wait for a CplComp to return

INTERFACE:

```
   subroutine ESMF_CplCompWait(cplcomp, syncflag, &
     timeout, timeoutFlag, userRc, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp),   intent(inout)         :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Sync_Flag), intent(in),  optional :: syncflag
     integer,              intent(in),  optional :: timeout
     logical,              intent(out), optional :: timeoutFlag
     integer,              intent(out), optional :: userRc
     integer,              intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument timeout. Added argument timeoutFlag. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

When executing asynchronously, wait for an ESMF_CplComp to return.

The arguments are:

cplcomp ESMF_CplComp to wait for.

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

[timeout] The maximum period in seconds the actual component is allowed to execute a previously invoked component method before it must communicate back to the dual component. If the actual component does not communicate back in the specified time, a timeout condition is raised on the dual side (this side). The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

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

18.4.30 ESMF_CplCompWriteRestart – Call the CplComp's write restart routine

INTERFACE:

```
recursive subroutine ESMF_CplCompWriteRestart(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  integer,              intent(out),   optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user write restart routine for an `ESMF_CplComp`.

The arguments are:

cplcomp `ESMF_CplComp` to call run routine for.

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

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

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

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

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

19 SciComp Class

19.1 Description

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

Unlike Gridded and Coupler Components, Science Components are not associated with software; they don't include execution routines such as initialize, run and finalize. The main purpose of a Science Component is to provide a container for Attributes within a Component hierarchy.

19.2 Use and Examples

A Science Component is a container object intended to represent scientific domains, or realms, in an Earth Science Model. It's primary purpose is to provide a means for representing Component metadata within a hierarchy of Components, and it does this by being a container for Attributes as well as other Components.

19.2.1 Use ESMF_SciComp and Attach Attributes

This example illustrates the use of the ESMF_SciComp to attach Attributes within a Component hierarchy. The hierarchy includes Coupler, Gridded, and Science Components and Attributes are attached to the Science Components. For demonstrable purposes, we'll add some CIM Component attributes to the Gridded Component.

Create the top 2 levels of the Component hierarchy. This example creates a parent Coupler Component and 2 Gridded Components as children.

```
! Create top-level Coupler Component
cplcomp = ESMF_CplCompCreate(name="coupler_component", rc=rc)

! Create Gridded Component for Atmosphere
atmcomp = ESMF_GridCompCreate(name="Atmosphere", rc=rc)

! Create Gridded Component for Ocean
ocncomp = ESMF_GridCompCreate(name="Ocean", rc=rc)
```

Now add CIM Attribute packages to the Component. Also, add a CIM Component Properties package, to contain two custom attributes.

```
convCIM = 'CIM 1.5'
purpComp = 'ModelComp'
purpProp = 'CompProp'
purpField = 'Inputs'
purpPlatform = 'Platform'

convISO = 'ISO 19115'
purpRP = 'RespParty'
purpCitation = 'Citation'

! Add CIM Attribute package to the Science Component
call ESMF_AttributeAdd(atmcomp, convention=convCIM, &
  purpose=purpComp, attpack=attpack, rc=rc)
```

The Attribute package can also be retrieved in a multi-Component setting like this:

```
call ESMF_AttributeGetAttPack(atmcomp, convCIM, purpComp, &
                              attpack=attpack, rc=rc)
```

Now, add some CIM Component attributes to the Atmosphere Grid Component.

```
!
! Top-level model component attributes, set on gridded component
!
call ESMF_AttributeSet(atmcomp, 'ShortName', 'EarthSys_Atmos', &
  attpack=attpack, rc=rc)
```

```
call ESMF_AttributeSet(atmcomp, 'LongName', &
  'Earth System High Resolution Global Atmosphere Model', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'Description', &
  'EarthSys brings together expertise from the global ' // &
  'community in a concerted effort to develop coupled ' // &
  'climate models with increased horizontal resolutions.  ' // &
  'Increasing the horizontal resolution of coupled climate ' // &
  'models will allow us to capture climate processes and ' // &
  'weather systems in much greater detail.', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'Version', '2.0', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'ReleaseDate', '2009-01-01T00:00:00Z', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'ModelType', 'aerosol', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'URL', &
  'www.earthsys.org', attpack=attpack, rc=rc)
```

Now create a set of Science Components as a children of the Atmosphere Gridded Component. The hierarchy is as follows:

- Atmosphere
  - AtmosDynamicalCore
    - AtmosAdvection
  - AtmosRadiation

After each Component is created, we need to link it with its parent Component. We then add some standard CIM Component properties as well as Scientific Properties to each of these components.

```
!
! Atmosphere Dynamical Core Science Component
!
dc_scicomp = ESMF_SciCompCreate(name="AtmosDynamicalCore", rc=rc)
```

```
call ESMF_AttributeAdd(dc_scicomp,  &
                       convention=convCIM, purpose=purpComp, &
                       attpack=attpack, rc=rc)

call ESMF_AttributeSet(dc_scicomp, "ShortName", "AtmosDynamicalCore", &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(dc_scicomp, "LongName", &
                       "Atmosphere Dynamical Core", &
                       attpack=attpack, rc=rc)


purpSci = 'SciProp'

dc_sciPropAtt(1) = 'TopBoundaryCondition'
dc_sciPropAtt(2) = 'HeatTreatmentAtTop'
dc_sciPropAtt(3) = 'WindTreatmentAtTop'

call ESMF_AttributeAdd(dc_scicomp,  &
                       convention=convCIM, purpose=purpSci, &
                       attrList=dc_sciPropAtt, &
                       attpack=attpack, rc=rc)

call ESMF_AttributeSet(dc_scicomp, 'TopBoundaryCondition', &
                       'radiation boundary condition', &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(dc_scicomp, 'HeatTreatmentAtTop', &
                       'some heat treatment', &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(dc_scicomp, 'WindTreatmentAtTop', &
                       'some wind treatment', &
                       attpack=attpack, rc=rc)



!
! Atmosphere Advection Science Component
!
adv_scicomp = ESMF_SciCompCreate(name="AtmosAdvection", rc=rc)


call ESMF_AttributeAdd(adv_scicomp,  &
                       convention=convCIM, purpose=purpComp, &
                       attpack=attpack, rc=rc)

call ESMF_AttributeSet(adv_scicomp, "ShortName", "AtmosAdvection", &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(adv_scicomp, "LongName", "Atmosphere Advection", &
                       attpack=attpack, rc=rc)


adv_sciPropAtt(1) = 'TracersSchemeName'
adv_sciPropAtt(2) = 'TracersSchemeCharacteristics'
adv_sciPropAtt(3) = 'MomentumSchemeName'

call ESMF_AttributeAdd(adv_scicomp,  &
```

```
                        convention=convCIM, purpose=purpSci, &
                        attrList=adv_sciPropAtt, &
                        attpack=attpack, rc=rc)

    call ESMF_AttributeSet(adv_scicomp, 'TracersSchemeName', 'Prather', &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(adv_scicomp, 'TracersSchemeCharacteristics', &
                        'modified Euler', &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(adv_scicomp, 'MomentumSchemeName', 'Van Leer', &
                        attpack=attpack, rc=rc)



    !
    ! Atmosphere Radiation Science Component
    !
    rad_scicomp = ESMF_SciCompCreate(name="AtmosRadiation", rc=rc)


    call ESMF_AttributeAdd(rad_scicomp,  &
                        convention=convCIM, purpose=purpComp, &
                        attpack=attpack, rc=rc)

    call ESMF_AttributeSet(rad_scicomp, "ShortName", "AtmosRadiation", &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(rad_scicomp, "LongName", &
                        "Atmosphere Radiation", &
                        attpack=attpack, rc=rc)



    rad_sciPropAtt(1) = 'LongwaveSchemeType'
    rad_sciPropAtt(2) = 'LongwaveSchemeMethod'

    call ESMF_AttributeAdd(rad_scicomp,  &
                        convention=convCIM, purpose=purpSci, &
                        attrList=rad_sciPropAtt, &
                        attpack=attpack, rc=rc)

    call ESMF_AttributeSet(rad_scicomp, &
                        'LongwaveSchemeType', &
                        'wide-band model', &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(rad_scicomp, &
                        'LongwaveSchemeMethod', &
                        'two-stream', &
                        attpack=attpack, rc=rc)
```

Finally, destroy all of the Components.

```
  call ESMF_SciCompDestroy(rad_scicomp, rc=rc)
  call ESMF_SciCompDestroy(adv_scicomp, rc=rc)
  call ESMF_SciCompDestroy(dc_scicomp, rc=rc)
```

```
call ESMF_GridCompDestroy(atmcomp, rc=rc)
call ESMF_GridCompDestroy(ocncomp, rc=rc)
call ESMF_CplCompDestroy(cplcomp, rc=rc)
```

19.3 Restrictions and Future Work

1. **None.**

19.4 Class API

19.4.1 ESMF_SciCompAssignment(=) - SciComp assignment

INTERFACE:

```
interface assignment(=)
scicomp1 = scicomp2
```

ARGUMENTS:

```
type(ESMF_SciComp) :: scicomp1
type(ESMF_SciComp) :: scicomp2
```

DESCRIPTION:

Assign scicomp1 as an alias to the same ESMF SciComp object in memory as scicomp2. If scicomp2 is invalid, then scicomp1 will be equally invalid after the assignment.

The arguments are:

scicomp1 The `ESMF_SciComp` object on the left hand side of the assignment.

scicomp2 The `ESMF_SciComp` object on the right hand side of the assignment.

19.4.2 ESMF_SciCompOperator(==) - SciComp equality operator

INTERFACE:

```
interface operator(==)
  if (scicomp1 == scicomp2) then ... endif
            OR
  result = (scicomp1 == scicomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_SciComp), intent(in) :: scicomp1
type(ESMF_SciComp), intent(in) :: scicomp2
```

DESCRIPTION:

Test whether scicomp1 and scicomp2 are valid aliases to the same ESMF SciComp object in memory. For a more general comparison of two ESMF SciComps, going beyond the simple alias test, the ESMF_SciCompMatch() function (not yet implemented) must be used.

The arguments are:

scicomp1 The `ESMF_SciComp` object on the left hand side of the equality operation.

scicomp2 The `ESMF_SciComp` object on the right hand side of the equality operation.

19.4.3 ESMF_SciCompOperator(/=) - SciComp not equal operator

INTERFACE:

```
interface operator(/=)
  if (scicomp1 /= scicomp2) then ... endif
            OR
  result = (scicomp1 /= scicomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_SciComp), intent(in) :: scicomp1
type(ESMF_SciComp), intent(in) :: scicomp2
```

DESCRIPTION:

Test whether scicomp1 and scicomp2 are *not* valid aliases to the same ESMF SciComp object in memory. For a more general comparison of two ESMF SciComps, going beyond the simple alias test, the ESMF_SciCompMatch() function (not yet implemented) must be used.

The arguments are:

scicomp1 The `ESMF_SciComp` object on the left hand side of the non-equality operation.

scicomp2 The `ESMF_SciComp` object on the right hand side of the non-equality operation.

19.4.4 ESMF_SciCompCreate - Create a SciComp

INTERFACE:

```
recursive function ESMF_SciCompCreate(name, rc)
```

RETURN VALUE:

```
type(ESMF_SciComp) :: ESMF_SciCompCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    character(len=*),        intent(in),    optional :: name
    integer,                 intent(out),   optional :: rc
```

DESCRIPTION:

This interface creates an ESMF_SciComp object. The return value is the new ESMF_SciComp.

The arguments are:

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

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

19.4.5 ESMF_SciCompDestroy - Release resources associated with a SciComp

INTERFACE:

```
subroutine ESMF_SciCompDestroy(scicomp, rc)
```

ARGUMENTS:

```
    type(ESMF_SciComp), intent(inout)           :: scicomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(out),  optional :: rc
```

DESCRIPTION:

Destroys an ESMF_SciComp, releasing the resources associated with the object.

The arguments are:

scicomp Release all resources associated with this ESMF_SciComp 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.

19.4.6 ESMF_SciCompGet - Get SciComp information

INTERFACE:

```
   subroutine ESMF_SciCompGet(scicomp, name, rc)
```

ARGUMENTS:

```
     type(ESMF_SciComp),       intent(in)            :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),         intent(out), optional :: name
     integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Get information about an `ESMF_SciComp` object.

The arguments are:

scicomp The `ESMF_SciComp` object being queried.

[name] Return the name of the `ESMF_SciComp`.

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

19.4.7 ESMF_SciCompIsCreated - Check whether a SciComp object has been created

INTERFACE:

```
   function ESMF_SciCompIsCreated(scicomp, rc)
```

RETURN VALUE:

```
     logical :: ESMF_SciCompIsCreated
```

ARGUMENTS:

```
     type(ESMF_SciComp), intent(in)            :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the `scicomp` has been created. Otherwise return `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

scicomp ESMF_SciComp queried.

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

19.4.8 ESMF_SciCompPrint - Print SciComp information

INTERFACE:

```
   subroutine ESMF_SciCompPrint(scicomp, rc)
```

ARGUMENTS:

```
     type(ESMF_SciComp), intent(in)             :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Prints information about an ESMF_SciComp to stdout.

The arguments are:

scicomp ESMF_SciComp to print.

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

19.4.9 ESMF_SciCompSet - Set or reset information about the SciComp

INTERFACE:

```
   subroutine ESMF_SciCompSet(scicomp, name, rc)
```

ARGUMENTS:

```
     type(ESMF_SciComp), intent(inout)          :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),    intent(in),  optional :: name
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Sets or resets information about an ESMF_SciComp.

The arguments are:

scicomp `ESMF_SciComp` to change.

[name] Set the name of the `ESMF_SciComp`.

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

19.4.10 ESMF_SciCompValidate - Check validity of a SciComp

INTERFACE:

```
   subroutine ESMF_SciCompValidate(scicomp, rc)
```

ARGUMENTS:

```
     type(ESMF_SciComp), intent(in)             :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Currently all this method does is to check that the `scicomp` was created.

The arguments are:

scicomp `ESMF_SciComp` to validate.

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

20 Fault-tolerant Component Tunnel

20.1 Description

For ensemble runs with many ensemble members, fault-tolerance becomes an issue of very critical practical impact. The meaning of *fault-tolerance* in this context refers to the ability of an ensemble application to continue with normal execution after one or more ensemble members have experienced catastrophic conditions, from which they cannot recover. ESMF implements this type of fault-tolerance on the Component level via a **timeout** paradigm: A timeout parameter is specified for all interactions that need to be fault-tolerant. When a connection to a component times out, maybe because it has become inaccessible due to some catastrophic condition, the driver application can react to this condition, for example by not further interacting with the component during the otherwise normal continuation of the model execution.

The fault-tolerant connection between a driver application and a Component is established through a **Component Tunnel**. There are two sides to a Component Tunnel: the "actual" side is where the component is actually executing, and the "dual" side is the portal through which the Component becomes accessible on the driver side. Both the actual and the dual side of a Component Tunnel are implemented in form of a regular ESMF Gridded or Coupler Component.

Component Tunnels between Components can be based on a number of low level implementations. The only implementation that currently provides fault-tolerance is *socket* based. In this case an actual Component typically runs as

a separate executable, listening to a specific port for connections from the driver application. The dual Component is created on the driver side. It connects to the actual Component during the SetServices() call.

20.2 Use and Examples

A Component Tunnel connects a *dual* Component to an *actual* Component. This connection can be based on a number of different low level implementations, e.g. VM-based or socket-based. VM-based Component Tunnels require that both dual and actual Components run within the same application (i.e. execute under the same MPI_COMM_WORLD). Fault-tolerant Component Tunnels require that dual and actual Components run in separate applications, under different MPI_COMM_WORLD communicators. This mode is implemented in the socket-based Component Tunnels.

20.2.1 Creating an *actual* Component

The creation process of an *actual* Gridded Component, which will become one of the two end points of a Component Tunnel, is identical to the creation of a regular Gridded Component. On the actual side, an actual Component is very similar to a regular Component. Here the actual Component is created with a custom `petList`.

```
petList = (/0,1,2/)
actualComp = ESMF_GridCompCreate(petList=petList, name="actual", rc=rc)
```

20.2.2 Creating a *dual* Component

The same way an actual Component appears as a regular Component in the context of the actual side application, a *dual* Component is created as a regular Component on the dual side. A dual Gridded Component with custom `petList` is created using the regular create call.

```
petList = (/4,3,5/)
dualComp = ESMF_GridCompCreate(petList=petList, name="dual", rc=rc)
```

20.2.3 Setting up the *actual* side of a Component Tunnel

After creation, the regular procedure for registering the standard Component methods is followed for the actual Gridded Component.

```
call ESMF_GridCompSetServices(actualComp, userRoutine=setservices, &
  userRc=userRc, rc=rc)
```

So far the `actualComp` object is no different from a regular Gridded Component. In order to turn it into the *actual* end point of a Component Tunnel the `ServiceLoop()` method is called. Here the socket-based implementation is chosen.

```
call ESMF_GridCompServiceLoop(actualComp, port=61010, timeout=20, rc=rc)
```

This call opens the actual side of the Component Tunnel in form of a socket-based server, listening on `port` 61010. The `timeout` argument specifies how long the actual side will wait for the dual side to connect, before the actual side returns with a time out condition. The time out is set to 20 seconds.

At this point, before a dual Component connects to the other side of the Component Tunnel, it is possible to manually connect to the waiting actual Component. This can be useful when debugging connection issues. A convenient tool for this is the standard `telnet` application. Below is a transcript of such a connection. The manually typed commands are separate from the previous responses by a blank line.

```
$ telnet localhost 61010
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello from ESMF Actual Component server!

date
Tue Apr  3 21:53:03 2012

version
ESMF_VERSION_STRING: 5.3.0
```

If at any point the `telnet` session is manually shut down, the `ServiceLoop()` on the actual side will return with an error condition. The clean way to disconnect the `telnet` session, and to have the `ServiceLoop()` wait for a new connection, e.g. from a dual Component, is to send the `reconnect` command. This will automatically shut down the `telnet` connection.

```
reconnect
Actual Component server will reconnect now!
Connection closed by foreign host.
$
```

At this point the actual Component is back in listening mode, with a time out of 20 seconds, as specified during the ServiceLoop() call.

Before moving on to the dual side of the GridComp based Component Tunnel example, it should be pointed out that the exact same procedure is used to set up the actual side of a *CplComp* based Component Tunnel. Assuming that `actualCplComp` is a CplComp object for which SetServices has already been called, the actual side uses `ESMF_CplCompServiceLoop()` to start listening for connections from the dual side.

```
call ESMF_CplCompServiceLoop(actualCplComp, port=61011, timeout=2, &
  timeoutFlag=timeoutFlag, rc=rc)
```

Here the `timeoutFlag` is specified in order to prevent the expected time-out condition to be indicated through the return code. Instead, when `timeoutFlag` is present, the return code is still `ESMF_SUCCESS`, but `timeoutFlag` is set to `.true.` when a time-out occurs.

20.2.4 Setting up the *dual* side of a Component Tunnel

On the dual side, the `dualComp` object needs to be connected to the actual Component in order to complete the Component Tunnel. Instead of registering standard Component methods locally, a special variant of the `SetServices()` call is used to connect to the actual Component.

```
call ESMF_GridCompSetServices(dualComp, port=61010, server="localhost", &
  timeout=10, timeoutFlag=timeoutFlag, rc=rc)
```

The `port` and `server` arguments are used to connect to the desired actual Component. The time out of 10 seconds ensures that if the actual Component is not available, a time out condition is returned instead of resulting in a hang. The `timeoutFlag` argument further absorbs the time out condition, either returning as `.true.` or `.false.`. In this mode the standard `rc` will indicate success even when a time out condition was reached.

20.2.5 Invoking standard Component methods through a Component Tunnel

Once a Component Tunnel is established, the actual Component is fully under the control of the dual Component. A standard Component method invoked on the dual Component is not executed by the dual Component itself, but by the actual Component instead. In fact, it is the entry points registered with the actual Component that are executed when standard methods are invoked on the dual Component. The connected `dualComp` object serves as a portal through which the connected `actualComp` becomes accessible on the dual side.

Typically the first standard method called is the `CompInitialize()` routine.

```
call ESMF_GridCompInitialize(dualComp, timeout=10, timeoutFlag=timeoutFlag, &
  userRc=userRc, rc=rc)
```

Again, the `timeout` argument serves to prevent the dual side from hanging if the actual Component application has experienced a catastrophic condition and is no longer available, or takes longer than expected. The presence of the `timeoutFlag` allows time out conditions to be caught gracefully, so the dual side can deal with it in an orderly fashion, instead of triggering an application abort due to an error condition.

The `CompRun()` and `CompFinalize()` methods follow the same format.

```
call ESMF_GridCompRun(dualComp, timeout=10, timeoutFlag=timeoutFlag, &
  userRc=userRc, rc=rc)
```

```
call ESMF_GridCompFinalize(dualComp, timeout=10, timeoutFlag=timeoutFlag, &
  userRc=userRc, rc=rc)
```

20.2.6 The non-blocking option to invoke standard Component methods through a Component Tunnel

Standard Component methods called on a connected dual Component are executed on the actual side, across the PETs of the actual Component. By default the dual Component PETs are blocked until the actual Component has finished executing the invoked Component method, or until a time out condition has been reached. In many practical applications a more loose synchronization between dual and actual Components is useful. Having the PETs of a dual Component return immediately from a standard Component method allows multiple dual Component, on the same PETs, to control multiple actual Components. If the actual Components are executing in separate executables, or the same executable but on exclusive sets of PETs, they can execute concurrently, even with the controlling dual Components all running on the same PETs. The non-blocking dual side regains control over the actual Component by synchronizing through the CompWait() call.

Any of the standard Component methods can be called in non-blocking mode by setting the optional `syncflag` argument to `ESMF_SYNC_NONBLOCKING`.

```
call ESMF_GridCompInitialize(dualComp, syncflag=ESMF_SYNC_NONBLOCKING, rc=rc)
```

If communication between the dual and the actual Component was successful, this call will return immediately on all of the dual Component PETs, while the actual Component continues to execute the invoked Component method. However, if the dual Component has difficulties reaching the actual Component, the call will block on all dual PETs until successful contact was made, or the default time out (3600 seconds, i.e. 1 hour) has been reached. In most cases a shorter time out condition is desired with the non-blocking option, as shown below.

First the dual Component must wait for the outstanding method.

```
call ESMF_GridCompWait(dualComp, rc=rc)
```

Now the same non-blocking CompInitialize() call is issued again, but this time with an explicit 10 second time out.

```
call ESMF_GridCompInitialize(dualComp, syncflag=ESMF_SYNC_NONBLOCKING, &
  timeout=10, timeoutFlag=timeoutFlag, rc=rc)
```

This call is guaranteed to return within 10 seconds, or less, on the dual Component PETs, either without time out condition, indicating that the actual Component has been contacted successfully, or with time out condition, indicating that the actual Component was unreachable at the time. Either way, the dual Component PETs are back under user control quickly.

Calling the CompWait() method on the dual Component causes the dual Component PETs to block until the actual Component method has returned, or a time out condition has been reached.

```
call ESMF_GridCompWait(dualComp, userRc=userRc, rc=rc)
```

The default time out for CompWait() is 3600 seconds, i.e. 1 hour, just like for the other Component methods. However, the semantics of a time out condition under CompWait() is different from the other Component methods. Typically the `timeout` is simply the maximum time that any communication between dual and actual Component is allowed to take before a time out condition is raised. For CompWait(), the `timeout` is the maximum time that an actual Component is allowed to execute before reporting back to the dual Component. Here, even with the default time out, the dual Component would return from CompWait() immediately with a time out condition if the actual Component has already been executing for over 1 hour, and is not already waiting to report back when the dual Component calls CompWait(). On the other hand, if it has only been 30 minutes since CompInitialize() was called on the dual Component, then the actual Component still has 30 minutes before CompWait() returns with a time out condition. During this time (or until the actual Component returns) the dual Component PETs are blocked.

A standard Component method is invoked in non-blocking mode.

```
call ESMF_GridCompRun(dualComp, syncflag=ESMF_SYNC_NONBLOCKING, &
  timeout=10, timeoutFlag=timeoutFlag, rc=rc)
```

Once the user code on the dual side is ready to regain control over the actual Component it calls CompWait() on the dual Component. Here a `timeout` of 60s is specified, meaning that the total execution time the actual Component spends in the registered Run() routine may not exceed 60s before CompWait() returns with a time out condition.

```
call ESMF_GridCompWait(dualComp, timeout=60, userRc=userRc, rc=rc)
```

20.2.7 Destroying a connected *dual* Component

A dual Component that is connected to an actual Component through a Component Tunnel is destroyed the same way a regular Component is. The only difference is that a connected dual Component may specify a `timeout` argument to the `CompDestroy()` call.

```
call ESMF_GridCompDestroy(dualComp, timeout=10, rc=rc)
```

The `timeout` argument again ensures that the dual side does not hang indefinitely in case the actual Component has become unavailable. If the actual Component is available, the destroy call will indicate to the actual Component that it should break out of the `ServiceLoop()`. Either way, the local dual Component is destroyed.

20.2.8 Destroying a connected *actual* Component

An actual Component that is in a `ServiceLoop()` must first return from that call before it can be destroyed. This can either happen when a connected dual Component calls its `CompDestroy()` method, or if the `ServiceLoop()` reaches the specified time out condition. Either way, once control has been returned to the user code, the actual Component is destroyed in the same way a regular Component is, by calling the destroy method.

```
call ESMF_GridCompDestroy(actualComp, rc=rc)
```

20.3 Restrictions and Future Work

1. **No data flow through States.** The current implementation does not support data flow (Fields, FieldBundles, etc.) between actual and dual Components. The current work-around is to employ user controlled, file based transfer methods. The next implementation phase will offer transparent data flow through the Component Tunnel, where the user code interacts with the States on the actual and dual side in the same way as if they were the same Component.

21 State Class

21.1 Description

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

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

States can contain Arrays, ArrayBundles, Fields, FieldBundles, and other States. They cannot directly contain native language arrays (i.e. Fortran or C style arrays). Objects in a State must span the VM on which they are running. For sequentially executing components which run on the same set of PETs this happens by calling the object create methods on each PET, creating the object in unison. For concurrently executing components which are running on

subsets of PETs, an additional method, called ESMF_StateReconcile(), is provided by ESMF to broadcast information about objects which were created in sub-components.

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

21.2 Constants

21.2.1 ESMF_STATEINTENT

DESCRIPTION:
Specifies whether a ESMF_State contains data to be imported into a component or exported from a component.

The type of this flag is:

type(ESMF_StateIntent_Flag)

The valid values are:

ESMF_STATEINTENT_IMPORT Contains data to be imported into a component.

ESMF_STATEINTENT_EXPORT Contains data to be exported out of a component.

ESMF_STATEINTENT_UNSPECIFIED The intent has not been specified.

21.2.2 ESMF_STATEITEM

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

The type of this flag is:

type(ESMF_StateItem_Flag)

The valid values are:

ESMF_STATEITEM_ARRAY Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_ARRAYBUNDLE Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_FIELD Refers to a ESMF_Field within an ESMF_State.

ESMF_STATEITEM_FIELDBUNDLE Refers to a ESMF_FieldBundle within an ESMF_State.

ESMF_STATEITEM_ROUTEHANDLE Refers to a ESMF_RouteHandle within an ESMF_State.

ESMF_STATEITEM_STATE Refers to a ESMF_State within an ESMF_State.

21.3 Use and Examples

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

the parent component into the Gridded Component's initialize method. This is where the States get prepared for use and the import State is first filled with data.

States can be filled with data items that do not yet have data allocated. Fields, FieldBundles, Arrays, and ArrayBundles each have methods that support their creation without actual data allocation - the Grid and Attributes 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, FieldBundles, and ArrayBundles within the State can allocate or reference data inside the initialize method.

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

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

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

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

21.3.1 State create and destroy

States can be created and destroyed at any time during application execution. The `ESMF_StateCreate()` routine can take many different combinations of optional arguments. Refer to the API description for all possible methods of creating a State. An empty State can be created by providing only a name and type for the intended State:

```
state = ESMF_StateCreate(name, stateintent=ESMF_STATEINTENT_IMPORT, rc=rc)
```

When finished with an `ESMF_State`, the `ESMF_StateDestroy` method removes it. However, the objects inside the `ESMF_State` created externally should be destroyed separately, since objects can be added to more than one `ESMF_State`.

21.3.2 Add items to a State

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

```
statename = "Ocean"
state2 = ESMF_StateCreate(name=statename,  &
                          stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)


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


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


call ESMF_StateDestroy(state2, rc=rc)


call ESMF_FieldBundleDestroy(bundle1, rc=rc)
```

21.3.3 Add 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 that is required by another component.

```
statename = "Ocean"
state3 = ESMF_StateCreate(name=statename,  &
                          stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)


dataname = "Downward wind:needed"
call ESMF_AttributeSet (state3, dataname, .false., rc=rc)


dataname = "Humidity:needed"
call ESMF_AttributeSet (state3, dataname, .false., rc=rc)
```

21.3.4 Mark an item NEEDED

How to set the NEEDED state of an item.

```
    dataname = "Downward wind:needed"
    call ESMF_AttributeSet (state3, name=dataname, value=.true., rc=rc)
```

21.3.5 Create 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:needed"
    call ESMF_AttributeGet (state3, dataname, value=neededFlag, rc=rc)


    if (rc == ESMF_SUCCESS .and. neededFlag) then
        bundlename = dataname
        bundle2 = ESMF_FieldBundleCreate(name=bundlename, rc=rc)


        call ESMF_StateAdd(state3, (/bundle2/), rc=rc)


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

21.3.6 ESMF_StateReconcile() usage

The set services routines are used to tell ESMF which routine hold the user code for the initialize, run, and finalize blocks of user level Components. These are the separate subroutines called by the code below.

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

    type(ESMF_Field) :: field1
    integer :: localrc

    print *, "i am comp1_init"

    field1 = ESMF_FieldEmptyCreate(name="Comp1 Field", rc=localrc)

    call ESMF_StateAdd(istate, (/field1/), rc=localrc)

    rc = localrc
```

```
end subroutine comp1_init

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

    type(ESMF_Field) :: field2
    integer :: localrc

    print *, "i am comp2_init"

    field2 = ESMF_FieldEmptyCreate(name="Comp2 Field", rc=localrc)

    call ESMF_StateAdd(istate, (/field2/), rc=localrc)

    rc = localrc

end subroutine comp2_init

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

   rc = ESMF_SUCCESS
end subroutine comp_dummy


! !PROGRAM: ESMF_StateReconcileEx - State reconciliation
!
! !DESCRIPTION:
!
! This program shows examples of using the State Reconcile function
!-------------------------------------------------------------------------
#include "ESMF.h"

    ! ESMF Framework module
    use ESMF
    use ESMF_TestMod
    use ESMF_StateReconcileEx_Mod
    implicit none

    ! Local variables
    integer :: rc, petCount
    type(ESMF_State) :: state1
    type(ESMF_GridComp) :: comp1, comp2
    type(ESMF_VM) :: vm
    character(len=ESMF_MAXSTR) :: comp1name, comp2name, statename
```

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

```
! Get the global VM for this job.
call ESMF_VMGetGlobal(vm=vm, rc=rc)

comp1name = "Atmosphere"
comp1 = ESMF_GridCompCreate(name=comp1name, petList=(/ 0, 1 /), rc=rc)
print *, "GridComp Create returned, name = ", trim(comp1name)



comp2name = "Ocean"
comp2 = ESMF_GridCompCreate(name=comp2name, petList=(/ 2, 3 /), rc=rc)
print *, "GridComp Create returned, name = ", trim(comp2name)



statename = "Ocn2Atm"
state1 = ESMF_StateCreate(name=statename, rc=rc)
```

Here we register the subroutines which should be called for initialization. Then we call ESMF_GridCompInitialize() on all PETs, but the code runs only on the PETs given in the petList when the Component was created.

Because this example is so short, we call the entry point code directly instead of the normal procedure of nesting it in a separate SetServices() subroutine.

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



call ESMF_GridCompSetVM(comp2, comp_dummy, rc=rc)



call ESMF_GridCompSetServices(comp1, userRoutine=comp_dummy, rc=rc)



call ESMF_GridCompSetServices(comp2, userRoutine=comp_dummy, rc=rc)



print *, "ready to set entry point 1"
call ESMF_GridCompSetEntryPoint(comp1, ESMF_METHOD_INITIALIZE, &
     comp1_init, rc=rc)
```

```
    print *, "ready to set entry point 2"
    call ESMF_GridCompSetEntryPoint(comp2, ESMF_METHOD_INITIALIZE, &
         comp2_init, rc=rc)




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



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

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

```
    print *, "State before calling StateReconcile()"
    call ESMF_StatePrint(state1, rc=rc)




    call ESMF_StateReconcile(state1, vm=vm, rc=rc)




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


end program ESMF_StateReconcileEx
```

21.3.7 Read Arrays from a NetCDF file and add to a State

This program shows an example of reading and writing Arrays from a State from/to a NetCDF file.

```
    ! ESMF Framework module
    use ESMF
    use ESMF_TestMod
    implicit none
```

```
! Local variables
type(ESMF_State) :: state
type(ESMF_Array) :: latArray, lonArray, timeArray, humidArray, &
                    tempArray, pArray, rhArray
type(ESMF_VM) :: vm
integer :: localPet, rc
```

The following line of code will read all Array data contained in a NetCDF file, place them in ESMF_Arrays and add them to an ESMF_State. Only PET 0 reads the file; the States in the other PETs remain empty. Currently, the data is not decomposed or distributed; each PET has only 1 DE and only PET 0 contains data after reading the file. Future versions of ESMF will support data decomposition and distribution upon reading a file.

Note that the third party NetCDF library must be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, NetCDF" and the website http://www.unidata.ucar.edu/software/netcdf.

```
! Read the NetCDF data file into Array objects in the State on PET 0
call ESMF_StateRead(state, "io_netcdf_testdata.nc", rc=rc)

! If the NetCDF library is not present (on PET 0), cleanup and exit
if (rc == ESMF_RC_LIB_NOT_PRESENT) then
  call ESMF_StateDestroy(state, rc=rc)
  goto 10
endif
```

Only reading data into ESMF_Arrays is supported at this time; ESMF_ArrayBundles, ESMF_Fields, and ESMF_FieldBundles will be supported in future releases of ESMF.

21.3.8 Print Array data from a State

To see that the State now contains the same data as in the file, the following shows how to print out what Arrays are contained within the State and to print the data contained within each Array. The NetCDF utility "ncdump" can be used to view the contents of the NetCDF file. In this example, only PET 0 will contain data.

```
if (localPet == 0) then
  ! Print the names and attributes of Array objects contained in the State
  call ESMF_StatePrint(state, rc=rc)

  ! Get each Array by name from the State
  call ESMF_StateGet(state, "lat",  latArray,   rc=rc)
  call ESMF_StateGet(state, "lon",  lonArray,   rc=rc)
  call ESMF_StateGet(state, "time", timeArray,  rc=rc)
  call ESMF_StateGet(state, "Q",    humidArray, rc=rc)
  call ESMF_StateGet(state, "TEMP", tempArray,  rc=rc)
  call ESMF_StateGet(state, "p",    pArray,     rc=rc)
  call ESMF_StateGet(state, "rh",   rhArray,    rc=rc)

  ! Print out the Array data
  call ESMF_ArrayPrint(latArray,   rc=rc)
  call ESMF_ArrayPrint(lonArray,   rc=rc)
```

```
  call ESMF_ArrayPrint(timeArray,  rc=rc)
  call ESMF_ArrayPrint(humidArray, rc=rc)
  call ESMF_ArrayPrint(tempArray,  rc=rc)
  call ESMF_ArrayPrint(pArray,     rc=rc)
  call ESMF_ArrayPrint(rhArray,    rc=rc)
endif
```

Note that the Arrays "lat", "lon", and "time" hold spatial and temporal coordinate data for the dimensions latitude, longitude and time, respectively. These will be used in future releases of ESMF to create `ESMF_Grids`.

21.3.9 Write Array data within a State to a NetCDF file

All the Array data within the State on PET 0 can be written out to a NetCDF file as follows:

```
! Write Arrays within the State on PET 0 to a NetCDF file
call ESMF_StateWrite(state, "io_netcdf_testdata_out.nc", rc=rc)
```

Currently writing is limited to PET 0; future versions of ESMF will allow parallel writing, as well as parallel reading.

21.4 Restrictions and Future Work

1. **No synchronization of object IDs at object create time.** Object IDs are used during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time.

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

21.5 Design and Implementation Notes

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

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

   When running components on subsets of the original VM all the PETs can create consistent objects but then when they are put into a State and passed to a component with a different VM and a different set of PETs, a communication call (reconcile) must be made to communicate the missing information to the PETs which were not involved in the original object creation. The reconcile call broadcasts object lists; those PETs which are

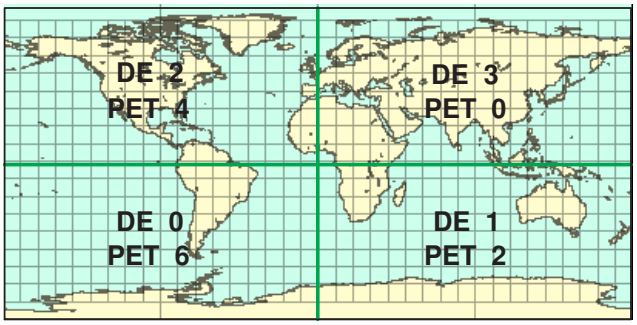


Figure 7: The mapping of PETs (processors) to DEs (data) in the source grid created by `user_model1.F90` in the FieldExcl system test.

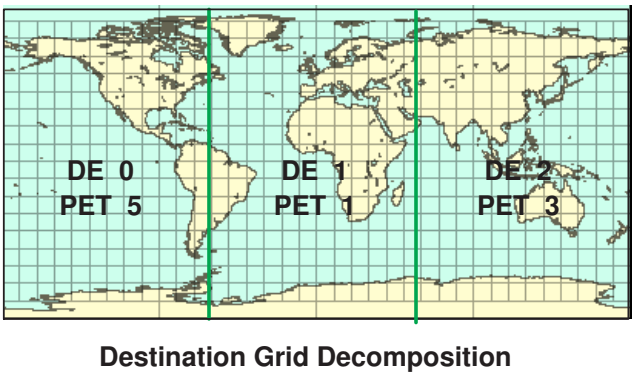


Figure 8: The mapping of PETs (processors) to DEs (data) in the destination grid created by `user_model2.F90` in the FieldExcl system test.

missing any objects in the total list can receive enough information to reconstruct a proxy object which contains all necessary information about that object, with no local data, on that PET. These proxy objects can be queried by ESMF routines to determine the amount of data and what PETs contain data which is destined to be moved to the local PET (for receiving data) and conversely, can determine which other PETs are going to receive data and how much (for sending data).

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

In the coupler code, all PETs must make the reconcile call before accessing data in the State. On PETs which already contain data, the objects are unchanged. On PETs which were not involved during the creation of the FieldBundles or Fields, the reconcile call adds an object to the State which contains all the same metadata

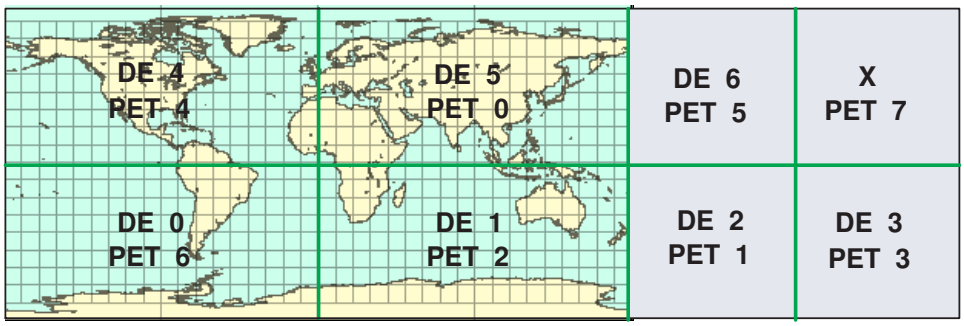


Figure 9: The mapping of PETs (processors) to DEs (data) in the source grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

associated with the object, but creates a slightly different Grid object, called a Proxy Grid. These PETs contain no local data, so the Array object is empty, and the DELayout for the Grid is like this:

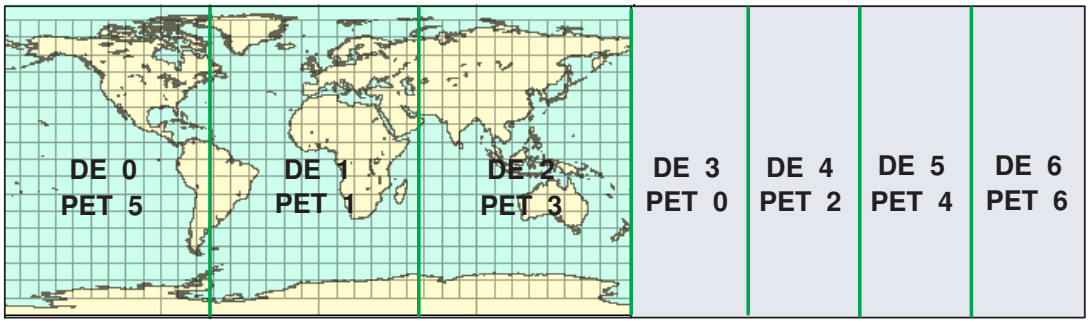
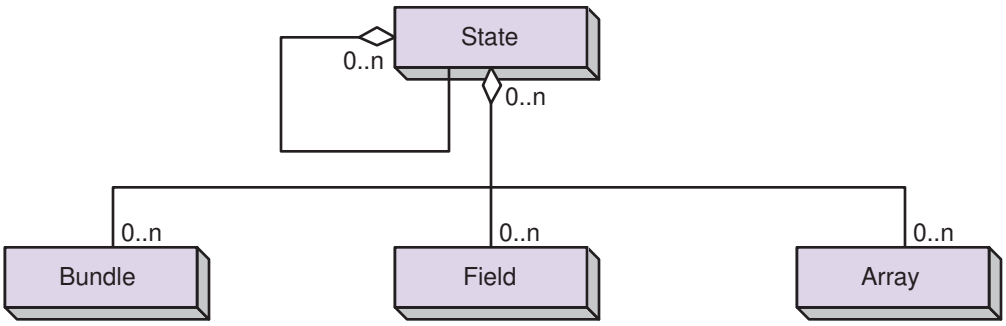


Figure 10: The mapping of PETs (processors) to DEs (data) in the destination grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

21.6 Object Model

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



21.7 Class API

21.7.1 ESMF_StateAssignment(=) - State assignment

INTERFACE:

```
interface assignment(=)
state1 = state2
```

ARGUMENTS:

```
type(ESMF_State) :: state1
type(ESMF_State) :: state2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Assign state1 as an alias to the same ESMF State object in memory as state2. If state2 is invalid, then state1 will be equally invalid after the assignment.

The arguments are:

state1 The `ESMF_State` object on the left hand side of the assignment.

state2 The `ESMF_State` object on the right hand side of the assignment.

21.7.2 ESMF_StateOperator(==) - State equality operator

INTERFACE:

```
interface operator(==)
if (state1 == state2) then ... endif
OR
result = (state1 == state2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state1
type(ESMF_State), intent(in) :: state2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Test whether state1 and state2 are valid aliases to the same ESMF State object in memory. For a more general comparison of two ESMF States, going beyond the simple alias test, the ESMF_StateMatch() function (not yet implemented) must be used.

The arguments are:

state1 The ESMF_State object on the left hand side of the equality operation.

state2 The ESMF_State object on the right hand side of the equality operation.

21.7.3 ESMF_StateOperator(/=) - State not equal operator

INTERFACE:

```
interface operator(/=)
if (state1 /= state2) then ... endif
OR
result = (state1 /= state2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state1
type(ESMF_State), intent(in) :: state2
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Test whether state1 and state2 are *not* valid aliases to the same ESMF State object in memory. For a more general comparison of two ESMF States, going beyond the simple alias test, the ESMF_StateMatch() function (not yet implemented) must be used.

The arguments are:

state1 The ESMF_State object on the left hand side of the non-equality operation.

state2 The ESMF_State object on the right hand side of the non-equality operation.

21.7.4 ESMF_StateAdd - Add a list of items to a State

INTERFACE:

```
subroutine ESMF_StateAdd(state, <itemList>, relaxedFlag, rc)
```

ARGUMENTS:

```
   type(ESMF_State), intent(inout) :: state
   <itemList>, see below for supported values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   logical, intent(in), optional :: relaxedFlag
   integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Add a list of items to a `ESMF_State`. It is an error if any item in <itemlist> already matches, by name, an item already contained in `state`.

Supported values for <itemList> are:

type(ESMF_Array), intent(in) :: arrayList(:)

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

type(ESMF_Field), intent(in) :: fieldList(:)

type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)

type(ESMF_RouteHandle), intent(in) :: routehandleList(:)

type(ESMF_State), intent(in) :: nestedStateList(:)

The arguments are:

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

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

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add", where it is *not* an error if `<itemList>` contains items with names that are found in `state`. The `State` is left unchanged for these items. For `.false.` this is treated as an error condition. The default setting is `.false.`.

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

21.7.5 ESMF_StateAddReplace - Add or replace a list of items to a State

INTERFACE:

```
subroutine ESMF_StateAddReplace(state, <itemList>, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
<itemList>, see below for supported values
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Add or replace a list of items to an `ESMF_State`. If an item in <itemList> does not match any items already present in `state`, it is added. Items with names already present in the `state` replace the existing item.

Supported values for <itemList> are:

type(ESMF_Array), intent(in) :: arrayList(:)

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

type(ESMF_Field), intent(in) :: fieldList(:)

type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)

type(ESMF_RouteHandle), intent(in) :: routehandleList(:)

type(ESMF_State), intent(in) :: nestedStateList(:)

The arguments are:

state An `ESMF_State` to which the <itemList> will be added or replaced.

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

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

21.7.6 ESMF_StateCreate - Create a new State

INTERFACE:

```
function ESMF_StateCreate(stateintent, &
             arrayList, arraybundleList, &
             fieldList, fieldbundleList, &
             nestedStateList, &
             routehandleList, name, vm, rc)
```

RETURN VALUE:

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

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_StateIntent_Flag), intent(in), optional :: stateintent
    type(ESMF_Array), intent(in), optional :: arrayList(:)
    type(ESMF_ArrayBundle), intent(in), optional :: arraybundleList(:)
    type(ESMF_Field), intent(in), optional :: fieldList(:)
    type(ESMF_FieldBundle), intent(in), optional :: fieldbundleList(:)
    type(ESMF_State), intent(in), optional :: nestedStateList(:)
    type(ESMF_RouteHandle), intent(in), optional :: routehandleList(:)
    character(len=*), intent(in), optional :: name
    type(ESMF_VM), intent(in), optional :: vm
    integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  8.1.0 Added argument `vm` to support object creation on a different VM than that of the current context.

DESCRIPTION:

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

The arguments are:

[stateintent] Import or Export `ESMF_State`. Valid values are `ESMF_STATEINTENT_IMPORT`, `ESMF_STATEINTENT_EXPORT`, or `ESMF_STATEINTENT_UNSPECIFIED` The default is `ESMF_STATEINTENT_UNSPECIFIED`.

[arrayList] A list (Fortran array) of `ESMF_Array`s.

[arraybundleList] A list (Fortran array) of `ESMF_ArrayBundle`s.

[fieldList] A list (Fortran array) of `ESMF_Field`s.

[fieldbundleList] A list (Fortran array) of `ESMF_FieldBundle`s.

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

[routehandleList] A list (Fortran array) of `ESMF_RouteHandle`s.

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

[vm] If present, the State object is created on the specified `ESMF_VM` object. The default is to create on the VM of the current component context.

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

21.7.7 ESMF_StateDestroy - Release resources for a State

INTERFACE:

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

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
logical, intent(in), optional :: noGarbage
integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  8.1.0 Added argument noGarbage. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Releases resources associated with this ESMF_State. Actual objects added to ESMF_States will not be destroyed, it remains the responsibility of the user to destroy these objects in the correct context.

The arguments are:

state Destroy contents of this ESMF_State.

[noGarbage] If set to .TRUE. the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

21.7.8 ESMF_StateGet - Get object-wide information from a State

INTERFACE:

```
! Private name; call using ESMF_StateGet()
subroutine ESMF_StateGetInfo(state, &
      itemSearch, itemorderflag, nestedFlag, &
      stateintent, itemCount, itemNameList, itemTypeList, name, rc)
```

ARGUMENTS:

```
   type(ESMF_State), intent(in) :: state
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
   character (len=*), intent(in), optional :: itemSearch
   type(ESMF_ItemOrder_Flag), intent(in), optional :: itemorderflag
   logical, intent(in), optional :: nestedFlag
   type(ESMF_StateIntent_Flag), intent(out), optional :: stateintent
   integer, intent(out), optional :: itemCount
   character (len=*), intent(out), optional :: itemNameList(:)
   type(ESMF_StateItem_Flag), intent(out), optional :: itemTypeList(:)
   character (len=*), intent(out), optional :: name
   integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.
- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  6.1.0 Added argument `itemorderflag`. The new argument gives the user control over the order in which the items are returned.

DESCRIPTION:

Returns the requested information about this `ESMF_State`. The optional `itemSearch` argument may specify the name of an individual item to search for. When used in conjunction with the `nestedFlag`, nested States will also be searched.

Typically, an `ESMF_StateGet()` information request will be performed twice. The first time, the `itemCount` argument will be used to query the size of arrays that are needed. Arrays can then be allocated to the correct size for `itemNameList` and `itemtypeList` as needed. A second call to `ESMF_StateGet()` will then fill in the values.

The arguments are:

state An `ESMF_State` object to be queried.

[itemSearch] Query objects by name in the State. When the `nestedFlag` option is set to .true., all nested States will also be searched for the specified name.

[itemorderflag] Specifies the order of the returned items in the `itemNameList` and `itemTypeList`. The default is `ESMF_ITEMORDER_ABC`. See **??** for a full list of options.

[nestedFlag] When set to `.false.`, returns information at the current State level only (default) When set to `.true.`, additionally returns information from nested States

[stateintent] Returns the type, e.g., Import or Export, of this `ESMF_State`. Possible values are listed in Section 21.2.1.

[itemCount] Count of items in this `ESMF_State`. When the `nestedFlag` option is set to `.true.`, the count will include items present in nested States. When using `itemSearch`, it will count the number of items matching the specified name.

[itemNameList] Array of item names in this `ESMF_State`. When the `nestedFlag` option is set to `.true.`, the list will include items present in nested States. When using `itemSearch`, it will return the names of items matching the specified name. `itemNameList` must be at least `itemCount` long.

[itemTypeList] Array of possible item object types in this `ESMF_State`. When the `nestedFlag` option is set to `.true.`, the list will include items present in nested States. When using `itemSearch`, it will return the types of items matching the specified name. Must be at least `itemCount` long. Return values are listed in Section 21.2.2.

[name] Returns the name of this `ESMF_State`.

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

21.7.9 ESMF_StateGet - Get information about an item in a State by item name

INTERFACE:

```
! Private name; call using ESMF_StateGet()
subroutine ESMF_StateGetItemInfo(state, itemName, itemType, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: itemName
type(ESMF_StateItem_Flag), intent(out) :: itemType
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

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

The arguments are:

state ESMF_State to be queried.

itemName Name of the item to return information about.

itemType Returned item types for the item with the given name, including placeholder names. Options are listed in Section 21.2.2. If no item with the given name is found, ESMF_STATEITEM_NOTFOUND will be returned and rc will **not** be set to an error.

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

21.7.10 ESMF_StateGet - Get an item from a State by item name

INTERFACE:

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

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: itemName
<item>, see below for supported values
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Returns an <item> from an ESMF_State by item name. If the ESMF_State contains the <item> directly, only itemName is required.

If the state contains nested ESMF_States, the itemName argument may specify a fully qualified name to access the desired item with a single call. This is performed using the '/' character to separate the names of the intermediate State names leading to the desired item. (E.g., itemName='state1/state12/item').

Supported values for <item> are:

type(ESMF_Array), intent(out) :: array

type(ESMF_ArrayBundle), intent(out) :: arraybundle

type(ESMF_Field), intent(out) :: field

type(ESMF_FieldBundle), intent(out) :: fieldbundle

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

type(ESMF_State), intent(out) :: nestedState

The arguments are:

state State to query for an <item> named `itemName`.

itemName Name of <item> to be returned. This name may be fully qualified in order to access nested State items.

<item> Returned reference to the <item>.

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

21.7.11 ESMF_StateIsCreated - Check whether an State object has been created

INTERFACE:

```
   function ESMF_StateIsCreated(state, rc)
```

RETURN VALUE:

```
     logical :: ESMF_StateIsCreated
```

ARGUMENTS:

```
     type(ESMF_State), intent(in) :: state
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the `state` has been created. Otherwise return `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

state `ESMF_State` queried.

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

21.7.12 ESMF_StatePrint - Print State information

INTERFACE:

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

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character(len=*), intent(in), optional :: options
logical, intent(in), optional :: nestedFlag
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: " ", or "brief" - print names and types of the objects within the state (default), "long" - print additional information, such as proxy flags

[nestedFlag] When set to `.false.`, prints information about the current State level only (default), When set to `.true.`, additionally prints information from nested States

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

21.7.13 ESMF_StateRead – Read data items from a file into a State

INTERFACE:

```
subroutine ESMF_StateRead(state, fileName, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Currently limited to read in all Arrays from a NetCDF file and add them to a State object. Future releases will enable more items of a State to be read from a file of various formats.

Only PET 0 reads the file; the States in other PETs remain empty. Currently, the data is not decomposed or distributed; each PET has only 1 DE and only PET 0 contains data after reading the file. Future versions of ESMF will support data decomposition and distribution upon reading a file. See Section 21.3.7 for an example.

Note that the third party NetCDF library must be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, NetCDF" and the website http://www.unidata.ucar.edu/software/netcdf.

The arguments are:

state The `ESMF_State` to add items read from file. Currently only Arrays are supported.

fileName File to be read.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors. Equals `ESMF_RC_LIB_NOT_PRESENT` if the NetCDF library is not present.

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

INTERFACE:

```
subroutine ESMF_StateReconcile(state, vm, rc)
```

ARGUMENTS:

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

DESCRIPTION:

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

This call is collective across the specified VM.

The arguments are:

state `ESMF_State` to reconcile.

[vm] `ESMF_VM` for this `ESMF_Component`. By default, it is set to the current vm.

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

21.7.15 ESMF_StateRemove - Remove an item from a State - (DEPRECATED METHOD)

INTERFACE:

```
! Private name; call using ESMF_StateRemove ()
subroutine ESMF_StateRemoveOneItem (state, itemName, &
    relaxedFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_State), intent(inout) :: state
  character(*), intent(in) :: itemName
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical, intent(in), optional :: relaxedFlag
  integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- **DEPRECATED METHOD** as of ESMF 5.3.1. Please use ESMF_StateRemove, section 21.7.16 instead. Rationale: The list version is consistent with other ESMF container operations which use lists.

DESCRIPTION:

Remove an existing reference to an item from a State.

The arguments are:

state The ESMF_State within which itemName will be removed.

itemName The name of the item to be removed. This is a reference only. The item itself is unchanged.

If the state contains nested ESMF_States, the itemName argument may specify a fully qualified name to remove the desired item with a single call. This is performed using the "/" character to separate the names of the intermediate State names leading to the desired item. (E.g., itemName="state1/state12/item".

Since an item could potentially be referenced by multiple containers, it remains the responsibility of the user to manage its destruction when it is no longer in use.

[relaxedflag] A setting of .true. indicates a relaxed definition of "remove", where it is *not* an error if itemName is not present in the state. For .false. this is treated as an error condition. The default setting is .false..

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

21.7.16 ESMF_StateRemove - Remove a list of items from a State

INTERFACE:

```
! Private name; call using ESMF_StateRemove ()
subroutine ESMF_StateRemoveList (state, itemNameList, relaxedFlag, rc)
```

ARGUMENTS:

```
    type(ESMF_State), intent(inout) :: state
    character(*), intent(in) :: itemNameList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical, intent(in), optional :: relaxedFlag
    integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.3.1. If code using this interface compiles with any version of ESMF starting with 5.3.1, then it will compile with the current version.

DESCRIPTION:

Remove existing references to items from a State.

The arguments are:

state The ESMF_State within which itemName will be removed.

itemNameList The name of the items to be removed. This is a reference only. The items themselves are unchanged.

If the state contains nested ESMF_States, the itemName arguments may specify fully qualified names to remove the desired items with a single call. This is performed using the "/" character to separate the names of the intermediate State names leading to the desired items. (E.g., itemName="state1/state12/item".

Since items could potentially be referenced by multiple containers, it remains the responsibility of the user to manage their destruction when they are no longer in use.

[relaxedflag] A setting of .true. indicates a relaxed definition of "remove", where it is *not* an error if an item in the itemNameList is not present in the state. For .false. this is treated as an error condition. The default setting is .false..

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

21.7.17 ESMF_StateReplace - Replace a list of items within a State

INTERFACE:

```
   subroutine ESMF_StateReplace(state, <itemList>, relaxedflag, rc)
```

ARGUMENTS:

```
   type(ESMF_State), intent(inout) :: state
   <itemList>, see below for supported values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   logical, intent(in), optional :: relaxedflag
   integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Replace a list of items with a ESMF_State. If an item in <itemList> does not match any items already present in state, an error is returned.

Supported values for <itemList> are:

type(ESMF_Array), intent(in) :: arrayList(:)

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

type(ESMF_Field), intent(in) :: fieldList(:)

type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)

type(ESMF_RouteHandle), intent(in) :: routehandleList(:)

type(ESMF_State), intent(in) :: nestedStateList(:)

The arguments are:

state An ESMF_State within which the <itemList> items will be replaced.

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

[relaxedflag] A setting of .true. indicates a relaxed definition of "replace", where it is *not* an error if <itemList> contains items with names that are not found in state. The State is left unchanged for these items. For .false. this is treated as an error condition. The default setting is .false..

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

21.7.18 ESMF_StateSet - Set State aspects

INTERFACE:

```
   subroutine ESMF_StateSet(state, stateIntent, rc)
```

ARGUMENTS:

```
     type(ESMF_State),             intent(inout)         :: state
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_StateIntent_Flag), intent(in),  optional :: stateIntent
     integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Set the info in the state object.

The arguments are:

state The ESMF_State to set.

stateIntent Intent, e.g. Import or Export, of this ESMF_State. Possible values are listed in Section 21.2.1.

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

21.7.19 ESMF_StateValidate - Check validity of a State

INTERFACE:

```
       subroutine ESMF_StateValidate(state, nestedFlag, rc)
```

ARGUMENTS:

```
      type(ESMF_State), intent(in) :: state
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      logical,          intent(in),  optional :: nestedFlag
      integer,          intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

Validates that the `state` is internally consistent. Currently this method determines if the `State` is uninitialized or already destroyed. The method returns an error code if problems are found.

The arguments are:

state The `ESMF_State` to validate.

[nestedFlag] `.false.` - validates at the current State level only (default) `.true.` - recursively validates any nested States

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

21.7.20 ESMF_StateWrite – Write items from a State to file

INTERFACE:

```
      subroutine ESMF_StateWrite(state, fileName, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Currently limited to write out all Arrays of a State object to a netCDF file. Future releases will enable more item types of a State to be written to files of various formats.

Writing is currently limited to PET 0; future versions of ESMF will allow parallel writing, as well as parallel reading.

See Section 21.3.7 for an example.

Note that the third party NetCDF library must be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, NetCDF" and the website http://www.unidata.ucar.edu/software/netcdf.

The arguments are:

state The ESMF_State from which to write items. Currently limited to Arrays.

fileName File to be written.

[rc] Return code; equals ESMF_SUCCESS if there are no errors. Equals ESMF_RC_LIB_NOT_PRESENT if the NetCDF library is not present.

22 Attachable Methods

22.1 Description

ESMF allows user methods to be attached to Components and States. Providing this capability supports a more object oriented way of model design.

Attachable methods on Components can be used to implement the concept of generic Components where the specialization requires attaching methods with well defined names. This methods are then called by the generic Component code.

Attaching methods to States can be used to supply data operations along with the data objects inside of a State object. This can be useful where a producer Component not only supplies a data set, but also the associated processing functionality. This can be more efficient than providing all of the possible sets of derived data.

22.2 Use and Examples

The following examples demonstrate how a producer Component attaches a user defined method to a State, and how it implements the method. The attached method is then executed by the consumer Component.

22.2.1 Producer Component attaches user defined method

The producer Component attaches a user defined method to exportState during the Component's initialize method. The user defined method is attached with label finalCalculation by which it will become accessible to the consumer Component.

```
subroutine init(gcomp, importState, exportState, clock, rc)
  ! arguments
  type(ESMF_GridComp):: gcomp
  type(ESMF_State):: importState, exportState
  type(ESMF_Clock):: clock
  integer, intent(out):: rc

  rc = ESMF_SUCCESS
  call ESMF_MethodAdd(exportState, label="finalCalculation", &
    userRoutine=finalCalc, rc=rc)
  if (rc /= ESMF_SUCCESS) return

  ! just for testing purposes add the same method with a crazy string label
  call ESMF_MethodAdd(exportState, label="Somewhat of a SILLY @$^@_ label", &
    userRoutine=finalCalc, rc=rc)
  if (rc /= ESMF_SUCCESS) return
```

```
end subroutine !------------------------------------------------------------
```

22.2.2 Producer Component implements user defined method

The producer Component implements the attached, user defined method `finalCalc`. Strict interface rules apply for the user defined method.

```
subroutine finalCalc(state, rc)
  ! arguments
  type(ESMF_State):: state
  integer, intent(out):: rc

  rc = ESMF_SUCCESS

  ! access data objects in state and perform calculation
  print *, "dummy output from attached method "

end subroutine !------------------------------------------------------------
```

22.2.3 Consumer Component executes user defined method

The consumer Component executes the user defined method on the `importState`.

```
subroutine init(gcomp, importState, exportState, clock, rc)
  ! arguments
  type(ESMF_GridComp):: gcomp
  type(ESMF_State):: importState, exportState
  type(ESMF_Clock):: clock
  integer, intent(out):: rc

  integer:: userRc, i
  logical:: isPresent
  character(len=:), allocatable :: labelList(:)

  rc = ESMF_SUCCESS
```

The importState can be queried for a list of *all* the attached methods.

```
  call ESMF_MethodGet(importState, labelList=labelList, rc=rc)
  if (rc /= ESMF_SUCCESS) return

  ! print the labels
  do i=1, size(labelList)
    print *, labelList(i)
  enddo
```

It is also possible to check the importState whether a *specific* method is attached. This allows the consumer code to implement alternatives in case the method is not available.

```
    call ESMF_MethodGet(importState, label="finalCalculation", &
      isPresent=isPresent, rc=rc)
    if (rc /= ESMF_SUCCESS) return
```

Finally call into the attached method from the consumer side.

```
    call ESMF_MethodExecute(importState, label="finalCalculation", &
      userRc=userRc, rc=rc)
    if (rc /= ESMF_SUCCESS) return
    rc = userRc
    if (rc /= ESMF_SUCCESS) return

  end subroutine !------------------------------------------------------------
```

22.3 Restrictions and Future Work

1. **Not reconciled.** Attachable Methods are PET-local settings on an object. Currently Attachable Methods cannot be reconciled (i.e. ignored during `ESMF_StateReconcile()`).
2. **No copy nor move.** Currently Attachable Methods cannot be copied or moved between objects.

22.4 Class API

22.4.1 ESMF_MethodAdd - Attach user method to CplComp

INTERFACE:

```
   ! Private name; call using ESMF_MethodAdd()
   subroutine ESMF_MethodCplCompAdd(cplcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp)                      :: cplcomp
     character(len=*), intent(in)            :: label
     integer,          intent(in),  optional :: index
     interface
       subroutine userRoutine(cplcomp, rc)
         use ESMF_CompMod
         implicit none
         type(ESMF_CplComp)          :: cplcomp      ! must not be optional
         integer, intent(out)        :: rc           ! must not be optional
       end subroutine
     end interface
     integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.2 ESMF_MethodAdd - Attach user method, located in shared object, to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodCplCompAddShObj(cplcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodCplCompAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.3 ESMF_MethodAdd - Attach user method to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodGridCompAdd(gcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(gcomp, rc)
    use ESMF_CompMod
    implicit none
    type(ESMF_GridComp)         :: gcomp        ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.4 ESMF_MethodAdd - Attach user method, located in shared object, to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodGridCompAddShObj(gcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodGridCompAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.5 ESMF_MethodAdd - Attach user method to State

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodStateAdd(state, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(state, rc)
    use ESMF_StateMod
    implicit none
    type(ESMF_State)             :: state        ! must not be optional
    integer, intent(out)         :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.6 ESMF_MethodAdd - Attach user method, located in shared object, to State

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodStateAddShObj(state, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
  type(ESMF_State)                        :: state
  character(len=*), intent(in)            :: label
  integer,          intent(in),  optional :: index
  character(len=*), intent(in)            :: userRoutine
  character(len=*), intent(in),  optional :: sharedObj
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodStateAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.7 ESMF_MethodAddReplace - Attach user method to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodCplCompAddRep(cplcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(cplcomp, rc)
    use ESMF_CompMod
    implicit none
    type(ESMF_CplComp)          :: cplcomp      ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.8 ESMF_MethodAddReplace - Attach user method, located in shared object, to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodCplCompAddRepShObj(cplcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodCplCompAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.9 ESMF_MethodAddReplace - Attach user method to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodGridCompAddRep(gcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(gcomp, rc)
    use ESMF_CompMod
    implicit none
    type(ESMF_GridComp)         :: gcomp        ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

gcomp The ESMF_GridComp to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

userRoutine The user-supplied subroutine to be associated with the label.

The subroutine must have the exact interface shown above for the userRoutine argument. Arguments in userRoutine must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.10 ESMF_MethodAddReplace - Attach user method, located in shared object, to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodGridCompAddRepShObj(gcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach userRoutine. Replacing potential previous attached method under the same label and index.

The arguments are:

gcomp The ESMF_GridComp to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

userRoutine Name of user-supplied subroutine to be associated with the label, specified as a character string.

The subroutine must have the exact interface shown in ESMF_MethodGridCompAdd for the userRoutine argument. Arguments in userRoutine must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another

procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.11 ESMF_MethodAddReplace - Attach user method to State

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodStateAddRep(state, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(state, rc)
    use ESMF_StateMod
    implicit none
    type(ESMF_State)             :: state        ! must not be optional
    integer, intent(out)         :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.12 ESMF_MethodAddReplace - Attach user method, located in shared object, to State

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodStateAddRepShObj(state, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_State)                        :: state
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach userRoutine. Replacing potential previous attached method under the same label and index.

The arguments are:

state The ESMF_State to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

userRoutine Name of user-supplied subroutine to be associated with the label, specified as a character string.

The subroutine must have the exact interface shown in ESMF_MethodStateAdd for the userRoutine argument. Arguments in userRoutine must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains userRoutine. If the sharedObj argument is not provided the executable itself will be searched for userRoutine.

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

22.4.13 ESMF_MethodExecute - Execute user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
recursive subroutine ESMF_MethodCplCompExecute(cplcomp, label, index, existflag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp)                      :: cplcomp
  character(len=*), intent(in)            :: label
  integer,          intent(in),  optional :: index
  logical,          intent(out), optional :: existflag
  integer,          intent(out), optional :: userRc
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Execute attached method.

The arguments are:

cplcomp The `ESMF_CplComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

[existflag] Returned `.true.` indicates that the method specified by `label` exists and was executed. A return value of `.false.` indicates that the method does not exist and consequently was not executed. By default, i.e. if `existflag` was not specified, the latter condition will lead to `rc` not equal `ESMF_SUCCESS` being returned. However, if `existflag` was specified, a method not existing is not an error condition.

[userRc] Return code set by attached method before returning.

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

22.4.14 ESMF_MethodExecute - Execute user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
recursive subroutine ESMF_MethodGridCompExecute(gcomp, label, index, existflag, &
  userRc, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: existflag
integer,          intent(out), optional :: userRc
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Execute attached method.

The arguments are:

gcomp The `ESMF_GridComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

[existflag] Returned `.true.` indicates that the method specified by `label` exists and was executed. A return value of `.false.` indicates that the method does not exist and consequently was not executed. By default, i.e. if `existflag` was not specified, the latter condition will lead to `rc` not equal `ESMF_SUCCESS` being returned. However, if `existflag` was specified, a method not existing is not an error condition.

[userRc] Return code set by attached method before returning.

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

22.4.15 ESMF_MethodExecute - Execute user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
recursive subroutine ESMF_MethodStateExecute(state, label, index, existflag, &
  userRc, rc)
```

ARGUMENTS:

```
type(ESMF_State)                        :: state
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: existflag
integer,          intent(out), optional :: userRc
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Execute attached method.

The arguments are:

state The ESMF_State object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

[existflag] Returned .true. indicates that the method specified by label exists and was executed. A return value of .false. indicates that the method does not exist and consequently was not executed. By default, i.e. if existflag was not specified, the latter condition will lead to rc not equal ESMF_SUCCESS being returned. However, if existflag was specified, a method not existing is not an error condition.

[userRc] Return code set by attached method before returning.

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

22.4.16 ESMF_MethodGet - Get info about user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodCplCompGet(cplcomp, label, index, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: isPresent
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access information about attached method.

The arguments are:

cplcomp The ESMF_CplComp object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

[isPresent] .true. if a method was attached for label/index. .false. otherwise.

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

22.4.17 ESMF_MethodGet - Get info about user methods attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodCplCompGetList(cplcomp, labelList, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                                  :: cplcomp
character(len=:), allocatable, intent(out)          :: labelList(:)
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access labels of all attached methods.

The arguments are:

cplcomp The `ESMF_CplComp` object holding the attachable method.

labelList List of labels of *all* the attached methods. On return, it will be allocated with as many list elements as there are attached methods. The length of each label in `labelList` is that of the largest method label currently attached. Elements with shorter labels are padded with white spaces.

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

22.4.18 ESMF_MethodGet - Get info about user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodGridCompGet(gcomp, label, index, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                    :: gcomp
character(len=*), intent(in)           :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: isPresent
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access information about attached method.

The arguments are:

gcomp The ESMF_GridComp object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

[isPresent] .true. if a method was attached for label/index. .false. otherwise.

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

22.4.19 ESMF_MethodGet - Get info about user methods attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodGridCompGetList(gcomp, labelList, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                                  :: gcomp
character(len=:), allocatable, intent(out)           :: labelList(:)
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access labels of all attached methods.

The arguments are:

gcomp The ESMF_GridComp object holding the attachable method.

labelList List of labels of *all* the attached methods. On return, it will be allocated with as many list elements as there are attached methods. The length of each label in labelList is that of the largest method label currently attached. Elements with shorter labels are padded with white spaces.

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

22.4.20 ESMF_MethodGet - Get info about user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodStateGet(state, label, index, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: isPresent
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access information about attached method.

The arguments are:

state The `ESMF_State` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

[isPresent] `.true.` if a method was attached for `label`/`index`. `.false.` otherwise.

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

22.4.21 ESMF_MethodGet - Get info about user methods attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodStateGetList(state, labelList, rc)
```

ARGUMENTS:

```
type(ESMF_State)                                    :: state
character(len=:), allocatable, intent(out)          :: labelList(:)
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access labels of all attached methods.

The arguments are:

state The `ESMF_State` object holding the attachable method.

labelList List of labels of *all* the attached methods. On return, it will be allocated with as many list elements as there are attached methods. The length of each label in `labelList` is that of the largest method label currently attached. Elements with shorter labels are padded with white spaces.

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

22.4.22 ESMF_MethodRemove - Remove user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodCplCompRemove(cplcomp, label, index, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Remove attached method.

The arguments are:

cplcomp The `ESMF_CplComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

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

22.4.23 ESMF_MethodRemove - Remove user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodGridCompRemove(gcomp, label, index, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                      :: gcomp
character(len=*),  intent(in)            :: label
 integer,          intent(in),  optional :: index
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Remove attached method.

The arguments are:

gcomp The `ESMF_GridComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

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

22.4.24 ESMF_MethodRemove - Remove user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodStateRemove(state, label, index, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Remove attached method.

The arguments are:

state The `ESMF_State` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

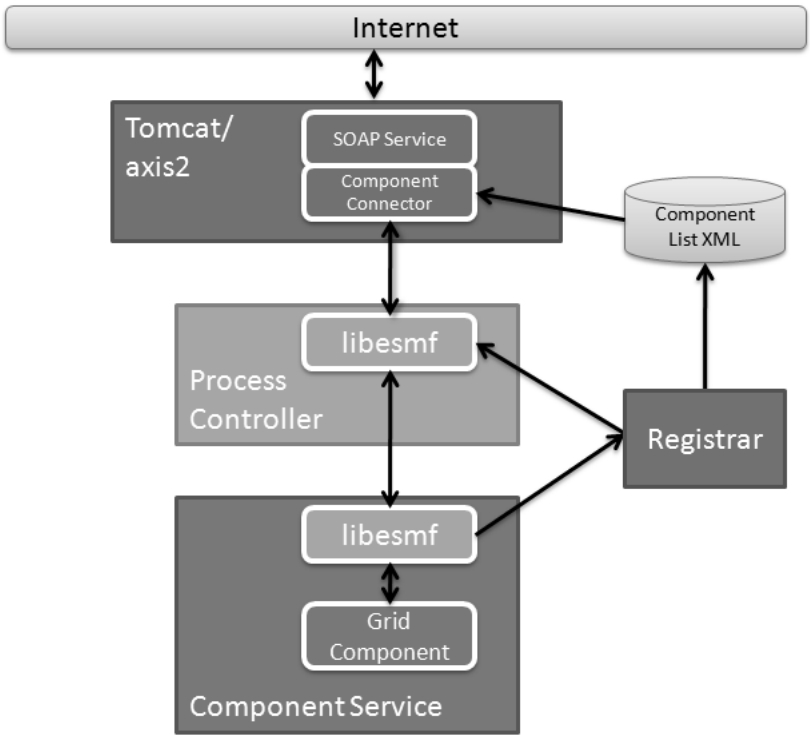
[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

23 Web Services

23.1 Description

The goal of the ESMF Web Services is to provide the tools to allow ESMF Users to make their Components available via a web service. The first step is to make the Component a service, and then make it accessible via the Web.

Figure 11: The diagram describes the ESMF Web Services software architecture. The architecture defines a multi-tiered set of applications that provide a flexible approach for accessing model components.



At the heart of this architecture is the Component Service; this is the application that does the model work. The ESMF Web Services part provides a way to make the model accessible via a network API (Application Programming Interface). ESMF provides the tools to turn a model component into a service as well as the tools to access the service from the network.

The Process Controller is a stand-alone application that provides a control mechanism between the end user and the Component Service. The Process Controller is responsible for managing client information as well as restricting client access to a Component Service. (The role of the Process Controller is expected to expand in the future.)

The tomcat/axis2 application provides the access via the Web using standard SOAP protocols. Part of this application includes the SOAP interface definition (using a WSDL file) as well as some java code that provides the access to the Process Controller application.

Finally, the Registrar maintains a list of Component Services that are currently available; Component Services register themselves with the Registrar when they startup, and unregister themselves when they shutdown. The list of available services is maintained in an XML file and is accessible from the Registrar using its network API.

23.1.1 Creating a Service around a Component

23.1.2 Code Modifications

One of the goals in providing the tools to make Components into services was to make the process as simple and easy as possible. Any model component that has been implemented using the ESMF Component Framework can easily be turned into a Component Services with just a minor change to the Application driver code. (For details on the ESMF Framework, see the ESMF Developers Documentation.)

The primary function in ESMF Web Services is the ESMF_WebServicesLoop routine. This function registers the Component Service with the Registrar and then sets up a network socket service that listens for requests from a client. It starts a loop that waits for incoming requests and manages the routing of these requests to all PETs. It is also responsible for making sure the appropriate ESMF routine (ESMF_Initialize, ESMF_Run or ESMF_Finalize) is called based on the incoming request. When the client has completed its interaction with the Component Service, the loop will be terminated and it will unregister the Component Service from the Registrar.

To make all of this happen, the Application Driver just needs to replace its calls to ESMF_Initialize, ESMF_Run, and ESMF_Finalize with a single call to ESMF_WebServicesLoop.

```
use ESMF_WebServMod
....

call ESMF_WebServicesLoop(gridComponent, portNumber, returnCode)
```

That's all there is to turning an ESMF Component into a network-accessible ESMF Component Service. For a detailed example of an ESMF Component turned into an ESMF Component Service, see the Examples in the Web Services section of the Developer' Guide.

23.1.3 Accessing the Service

Now that the Component is available as a service, it can be accessed remotely by any client that can communicate via TCP sockets. The ESMF library, in addition to providing the service tools, also provides the classes to create C++

clients to access the Component Service via the socket interface.

However, the goal of ESMF Web Services is to make an ESMF Component accessible through a standard web service, which is accomplished through the Process Controller and the Tomcat/Axis2 applications

23.1.4 Client Application via C++ API

Interfacing to a Component service is fairly simple using the ESMF library. The following code is a simple example of how to interface to a Component Service in C++ and request the initialize operation (the entire sample client can be found in the Web Services examples section of the ESMF Distribution):

```
#include "ESMCI_WebServCompSvrClient.h"

int  main(int  argc, char*  argv[])
{
        int    portNum = 27060;
           int    clientId = 101;
        int    rc = ESMF_SUCCESS;

        ESMCI::ESMCI_WebServCompSvrClient
                        client("localhost", portNum, clientId);

        rc = client.init();
        printf("Initialize return code: %d\n", rc);
}
```

To see a complete description of the NetEsmfClient class, refer to the netesmf library section of the Web Services Reference Manual.

23.1.5 Process Controller

The Process Controller is basically just a instance of a C++ client application. It manages client access to the Component Service (only 1 client can access the service at a time), and will eventually be responsible for starting up and shutting down instances of Component Services (planned for a future release). The Process Controller application is built with the ESMF library and is included in the apps section of the distribution.

23.1.6 Tomcat/Axis2

The Tomcat/Axis2 "application" is essentially the Apache Tomcat server using the Apache Axis2 servlet to implement web services using SOAP protocols. The web interface is defined by a WSDL file, and its implementation is handled by the Component Connector java code. Tomcat and Axis2 are both open source projects that should be downloaded from the Apache web site, but the WSDL file, the Component Connector java code, and all required software for supporting the interface can be found next to the ESMF distribution in the web_services_server directory. This code is not included with the ESMF distribution because they can be distributed and installed independent of each other.

23.2 Use and Examples

The following examples demonstrate how to use ESMF Web Services.

23.2.1 Making a Component available through ESMF Web Services

In this example, a standard ESMF Component is made available through the Web Services interface.

The first step is to make sure your callback routines for initialize, run and finalize are setup. This is done by creating a register routine that sets the entry points for each of these callbacks. In this example, we've packaged it all up into a separate module.

```
module ESMF_WebServUserModel

  ! ESMF Framework module
  use ESMF

  implicit none

  public ESMF_WebServUserModelRegister

  contains

  !-------------------------------------------------------------------------
  !  The Registration routine
  !
  subroutine ESMF_WebServUserModelRegister(comp, rc)
    type(ESMF_GridComp)  :: comp
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Register starting"

    ! Register the callback routines.

    call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_INITIALIZE, &
                                    userRoutine=user_init, rc=rc)
    if (rc/=ESMF_SUCCESS) return ! bail out

    call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_RUN, &
                                    userRoutine=user_run, rc=rc)
    if (rc/=ESMF_SUCCESS) return ! bail out

    call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_FINALIZE, &
                                    userRoutine=user_final, rc=rc)
    if (rc/=ESMF_SUCCESS) return ! bail out

    print *, "Registered Initialize, Run, and Finalize routines"
    print *, "User Comp1 Register returning"

  end subroutine
```

```
  !------------------------------------------------------------------------
  !  The Initialization routine
  !
  subroutine user_init(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp)  :: comp
    type(ESMF_State)     :: importState, exportState
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Init"

  end subroutine user_init

  !------------------------------------------------------------------------
  !  The Run routine
  !
  subroutine user_run(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp)  :: comp
    type(ESMF_State)     :: importState, exportState
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Run"

  end subroutine user_run

  !------------------------------------------------------------------------
  !  The Finalization routine
  !
  subroutine user_final(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp)  :: comp
    type(ESMF_State)     :: importState, exportState
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Final"

  end subroutine user_final

end module ESMF_WebServUserModel
```

The actual driver code then becomes very simple; ESMF is initialized, the component is created, the callback functions for the component are registered, and the Web Service loop is started.

```
program WebServicesEx
```

```
#include "ESMF.h"

  ! ESMF Framework module
  use ESMF
  use ESMF_TestMod

  use ESMF_WebServMod
  use ESMF_WebServUserModel

  implicit none

  ! Local variables
  type(ESMF_GridComp) :: comp1     !! Grid Component
  integer             :: rc        !! Return Code
  integer             :: finalrc  !! Final return code
  integer             :: portNum   !! The port number for the listening socket
```

A listening socket will be created on the local machine with the specified port number. This socket is used by the service to wait for and receive requests from the client. Check with your system administrator to determine an appropriate port to use for your service.

```
  finalrc = ESMF_SUCCESS

  call ESMF_Initialize(defaultlogfilename="WebServicesEx.Log", &
                   logkindflag=ESMF_LOGKIND_MULTI, rc=rc)


  ! create the grid component
  comp1 = ESMF_GridCompCreate(name="My Component", rc=rc)


  ! Set up the register routine
  call ESMF_GridCompSetServices(comp1, &
          userRoutine=ESMF_WebServUserModelRegister, rc=rc)


  portNum = 27060

  ! Call the Web Services Loop and wait for requests to come in
  !call ESMF_WebServicesLoop(comp1, portNum, rc=rc)
```

The call to ESMF_WebServicesLoop will setup the listening socket for your service and will wait for requests from a client. As requests are received, the Web Services software will process the requests and then return to the loop to continue to wait.

The 3 main requests processed are INIT, RUN, and FINAL. These requests will then call the appropriate callback routine as specified in your register routine (as specified in the ESMF_GridCompSetServices call). In this example, when the INIT request is received, the user_init routine found in the ESMF_WebServUserModel module is called.

One other request is also processed by the Component Service, and that is the EXIT request. When this request is received, the Web Services loop is terminated and the remainder of the code after the ESMF_WebServicesLoop call is executed.

```
  call ESMF_Finalize(rc=rc)


end program WebServicesEx
```

23.3 Restrictions and Future Work

1. **Manual Control of Process.** Currently, the Component Service must be manually started and stopped. Future plans include having the Process Controller be responsible for controlling the Component Service processes.
2. **Data Streaming.** While data can be streamed from the web server to the client, it is not yet getting the data directly from the Component Service. Instead, the Component Service exports the data to a file which the Process Controller can read and return across the network interface. The data streaming capabilities will be a major component of future improvements to the Web Services architecture.

23.4 Class API

23.4.1 ESMF_WebServicesLoop

INTERFACE:

```
   subroutine ESMF_WebServicesLoop(comp, portNum, clientId, registrarHost, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                          :: comp
     integer,            intent(inout), optional :: portNum
     character(len=*),   intent(in),    optional, target :: clientId
     character(len=*),   intent(in),    optional, target :: registrarHost
     integer,            intent(out),   optional :: rc
```

DESCRIPTION:

Encapsulates all of the functionality necessary to setup a component as a component service. On the root PET, it registers the component service and then enters into a loop that waits for requests on a socket. The loop continues until an "exit" request is received, at which point it exits the loop and unregisters the service. On any PET other than the root PET, it sets up a process block that waits for instructions from the root PET. Instructions will come as requests are received from the socket.

The arguments are:

[comp] `ESMF_CplComp` object that represents the Grid Component for which routine is run.

[portNum] Number of the port on which the component service is listening.

[clientId] Identifier of the client responsible for this component service. If a Process Controller application manages this component service, then the clientId is provided to the component service application in the command line. Otherwise, the clientId is not necessary.

[registrarHost] Name of the host on which the Registrar is running. Needed so the component service can notify the Registrar when it is ready to receive requests from clients.

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

23.4.2 ESMF_WebServicesCplCompLoop

INTERFACE:

```
subroutine ESMF_WebServicesCplCompLoop(comp, portNum, clientId, registrarHost, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                          :: comp
integer,           intent(inout), optional :: portNum
character(len=*),  intent(in),    optional, target :: clientId
character(len=*),  intent(in),    optional, target :: registrarHost
integer,           intent(out),   optional :: rc
```

DESCRIPTION:

Encapsulates all of the functionality necessary to setup a component as a component service. On the root PET, it registers the component service and then enters into a loop that waits for requests on a socket. The loop continues until an "exit" request is received, at which point it exits the loop and unregisters the service. On any PET other than the root PET, it sets up a process block that waits for instructions from the root PET. Instructions will come as requests are received from the socket.

The arguments are:

[comp] `ESMF_CplComp` object that represents the Grid Component for which routine is run.

[portNum] Number of the port on which the component service is listening.

[clientId] Identifier of the client responsible for this component service. If a Process Controller application manages this component service, then the clientId is provided to the component service application in the command line. Otherwise, the clientId is not necessary.

[registrarHost] Name of the host on which the Registrar is running. Needed so the component service can notify the Registrar when it is ready to receive requests from clients.

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

Part IV

Infrastructure: Fields and Grids

24 Overview of Data Classes

The ESMF infrastructure data classes are part of the framework's hierarchy of structures for handling Earth system model data and metadata on parallel platforms. The hierarchy is in complexity; the simplest data class in the infrastructure represents a distributed data array and the most complex data class represents a bundle of physical fields that are discretized on the same grid. Data class methods are called both from user-written code and from other classes internal to the framework.

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

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

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

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

Key Features
Hierarchy of data structures designed specifically for the Earth system domain and high performance, parallel computing.
Multi-use ESMF structures simplify user code overall.
Data objects support incremental construction and deferred allocation.
Native Fortran arrays can be associated with or retrieved from ESMF data objects, for ease of adoption, convenience, and performance.
A variety of operations are provided for manipulating data in data objects such as regridding, redistribution, halo communication, and sparse matrix multiply.

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

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

  Data elements in Arrays are partitioned into categories defined by the role the data element plays in distributed halo operations. Haloing - sometimes called ghosting - is the practice of copying portions of array data to multiple memory locations to ensure that data dependencies can be satisfied quickly when performing a calculation. ESMF Arrays contain an **exclusive** domain, which contains data elements updated exclusively and definitively by a given DE; a **computational** domain, which contains all data elements with values that are updated by the

DE in computations; and a **total** domain, which includes both the computational domain and data elements from other DEs which may be read but are not updated in computations.

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

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

- **FieldBundle** Groups of Fields on the same underlying physical grid can be collected into a single object called a FieldBundle. A FieldBundle provides two major functions: it allows groups of Fields to be manipulated using a single identifier, for example during export or import of data between Components; and it allows data from multiple Fields to be packed together in memory for higher locality of reference and ease in subsetting operations. Packing a set of Fields into a single FieldBundle before performing a data communication allows the set to be transferred at once rather than as a Field at a time. This can improve performance on high-latency platforms.

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

24.1 Bit-for-Bit Considerations

Bit-for-bit reproducibility is at the core of the regression testing schemes of many scientific model codes. The bit-for-bit requirement makes it easy to compare the numerical results of simulation runs using standard binary diff tools.

For the most part, ESMF methods do not modify user data numerically, and thus have no effect on the bit-for-bit characteristics of the model code. The exceptions are the regrid weight generation and the sparse matrix multiplication.

In the case of the regrid weight generation, user data is used to produce interpolation weights following specific numerical schemes. The bit-for-bit reproducibility of the generated weights depends on the implementation details. Section 24.2 provides more details about the bit-for-bit considerations with respect to the regrid weights generated by ESMF.

In the case of the sparse matrix multiplication, which is the typical method that is used to apply the regrid weights, user data is directly manipulated by ESMF. In order to help users with the implementation of their bit-for-bit requirements, while also considering the associated performance impact, the ESMF sparse matrix implementation provides three levels of bit-for-bit support. The strictest level ensures that the numerical results are bit-for-bit identical, even when executing across different numbers of PETs. In the relaxed level, bit-for-bit reproducibility is guaranteed when running across an unchanged number of PETs. The lowest level makes no guarantees about bit-for-bit reproducibility, however, it provides the greatest performance potential for those cases where numerical round-off differences are acceptable. An in-depth discussion of bit-for-bit reproducibility, and the performance aspects of route-based communication methods, such the sparse matrix multiplication, is given in section **??**.

24.2 Regrid

This section describes the regridding methods provided by ESMF. Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Regridding can be broken into two stages. The first stage is generation of an interpolation weight matrix that describes how points in the source grid contribute to points in the destination grid. The second stage is the multiplication of values on the source grid by the interpolation weight matrix to produce values on the destination grid. This is implemented as a parallel sparse matrix multiplication.

There are two options for accessing ESMF regridding functionality: **offline** and **integrated**. Offline regridding is a process whereby interpolation weights are generated by a separate ESMF command line tool, not within the user code. The ESMF offline regridding tool also only generates the interpolation matrix, the user is responsible for reading in this matrix and doing the actual interpolation (multiplication by the sparse matrix) in their code. Please see Section 12 for a description of the offline regridding command line tool and the options it supports. For user convenience, there is also a method interface to the offline regrid tool functionality which is described in Section 24.3.1. In contrast to offline regridding, integrated regridding is a process whereby interpolation weights are generated via subroutine calls during the execution of the user's code. In addition to generating the weights, integrated regridding can also produce a **RouteHandle** (described in Section **??**) which allows the user to perform the parallel sparse matrix multiplication using ESMF methods. In other words, ESMF integrated regridding allows a user to perform the whole process of interpolation within their code.

To see what types of grids and other options are supported in the two types of regridding and their testing status, please see the ESMF Regridding Status webpage for this version of ESMF. Figure 24.2 shows a comparison of different regrid interfaces and where they can be found in the documentation.

The rest of this section further describes the various options available in ESMF regridding.

Name	Access via	Inputs	Outputs		Description
			Weights	RouteHandle	
ESMF_FieldRegridStore()	Subroutine call	Field object	yes	yes	Sec. **??**
ESMF_FieldBundleRegridStore()	Subroutine call	Fieldbundle obj.	no	yes	Sec. **??**
ESMF_RegridWeightGen()	Subroutine call	Grid files	yes	no	Sec. 24.3.1
ESMF_RegridWeightGen	Command Line Tool	Grid files	yes	no	Sec. 12

Table 1: Regrid Interfaces

24.2.1 Interpolation methods: bilinear

Bilinear interpolation calculates the value for the destination point as a combination of multiple linear interpolations, one for each dimension of the Grid. Note that for ease of use, the term bilinear interpolation is used for 3D interpolation in ESMF as well, although it should more properly be referred to as trilinear interpolation.

In 2D, ESMF supports bilinear regridding between any combination of the following:

- Structured grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- A set of disconnected points (`ESMF_LocStream`) may be the destination of the regridding
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports bilinear regridding between any combination of the following:

- Structured grids (`ESMF_Grid`) composed of a single logically rectangular tile

- Unstructured meshes (ESMF_Mesh) composed of hexahedrons
- A set of disconnected points (ESMF_LocStream) may be the destination of the regridding

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are currently ignored
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the bilinear method the user may create their Fields on any stagger location (e.g. ESMF_STAGGERLOC_CENTER) for a Grid, or any Mesh location (e.g. ESMF_MESHLOC_NODE) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a destination Field built on a LocStream that contains coordinates, or with a source or destination Field built on an XGrid.

24.2.2 Interpolation methods: higher-order patch

Patch (or higher-order) interpolation is the ESMF version of a technique called "patch recovery" commonly used in finite element modeling [**?**] [**?**]. It typically results in better approximations to values and derivatives when compared to bilinear interpolation. Patch interpolation works by constructing multiple polynomial patches to represent the data in a source cell. For 2D grids, these polynomials are currently 2nd degree 2D polynomials. One patch is constructed for each corner of the source cell, and the patch is constructed by doing a least squares fit through the data in the cells surrounding the corner. The interpolated value at the destination point is then a weighted average of the values of the patches at that point.

The patch method has a larger stencil than the bilinear, for this reason the patch weight matrix can be correspondingly larger than the bilinear matrix (e.g. for a quadrilateral grid the patch matrix is around 4x the size of the bilinear matrix). This can be an issue when performing a regrid operation close to the memory limit on a machine.

The patch method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the patch method, the destination field will contain values less than 0.0.

In 2D, ESMF supports patch regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of a single logically rectangular tile
- Unstructured meshes (ESMF_Mesh) composed of polygons with any number of sides
- A set of disconnected points (ESMF_LocStream) may be the destination of the regridding
- An exchange grid (ESMF_XGrid)

In 3D, ESMF supports patch regridding between any combination of the following:

- NONE

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are currently ignored
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the patch method the user may create their Fields on any stagger location (e.g. `ESMF_STAGGERLOC_CENTER`) for a Grid, or any Mesh location (e.g. `ESMF_MESHLOC_NODE`) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a destination Field built on a LocStream that contains coordinates, or with a source or destination Field built on an XGrid.

24.2.3 Interpolation methods: nearest source to destination

In nearest source to destination interpolation (`ESMF_REGRIDMETHOD_NEAREST_STOD`) each destination point is mapped to the closest source point. A given source point may map to multiple destination points, but no destination point will receive input from more than one source point. If two points are equally close, then the point with the smallest sequence index is arbitrarily used (i.e. the point which would have the smallest index in the weight matrix).

In 2D, ESMF supports nearest source to destination regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- A set of disconnected points (`ESMF_LocStream`)
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports nearest source to destination regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured Meshes (`ESMF_Mesh`) composed of hexahedrons (e.g. cubes) and tetrahedrons
- A set of disconnected points (`ESMF_LocStream`)

Restrictions:
NONE

To use the nearest source to destination method the user may create their Fields on any stagger location (e.g. `ESMF_STAGGERLOC_CENTER`) for a Grid, or any Mesh location (e.g. `ESMF_MESHLOC_NODE`) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a source or destination Field built on a LocStream that contains coordinates, or when the source or destination Field is built on an XGrid.

24.2.4 Interpolation methods: nearest destination to source

In nearest destination to source interpolation (ESMF_REGRIDMETHOD_NEAREST_DTOS) each source point is mapped to the closest destination point. A given destination point may receive input from multiple source points, but no source point will map to more than one destination point. If two points are equally close, then the point with the smallest sequence index is arbitrarily used (i.e. the point which would have the smallest index in the weight matrix). Note that with this method the unmapped destination point detection currently doesn't work, so no error will be returned even if there are destination points that don't map to any source point.

In 2D, ESMF supports nearest destination to source regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of any number of logically rectangular tiles
- Unstructured meshes (ESMF_Mesh) composed of polygons with any number of sides
- A set of disconnected points (ESMF_LocStream)
- An exchange grid (ESMF_XGrid)

In 3D, ESMF supports nearest destination to source regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of any number of logically rectangular tiles
- Unstructured Meshes (ESMF_Mesh) composed of hexahedrons (e.g. cubes) and tetrahedrons
- A set of disconnected points (ESMF_LocStream)

Restrictions:

- The unmapped destination point detection doesn't currently work for this method. Even if there are unmapped points, no error will be returned.

To use the nearest destination to source method the user may create their Fields on any stagger location (e.g. ESMF_STAGGERLOC_CENTER) for a Grid, or any Mesh location (e.g. ESMF_MESHLOC_NODE) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a source or destination Field built on a LocStream that contains coordinates, or when the source or destination Field is built on an XGrid.

24.2.5 Interpolation methods: first-order conservative

The goal of this method is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 24.2.7.) In this method the value across each source cell is treated as a constant, so it will typically have a larger interpolation error than the bilinear or patch methods. The first-order method used here is similar to that described in the following paper [**?**].

In the first-order method, the values for a particular destination cell are a calculated as a combination of the values of the intersecting source cells. The weight of a given source cell's contribution to the total being the amount that that source cell overlaps with the destination cell. In particular, the weight is the ratio of the area of intersection of the source and destination cells to the area of the whole destination cell.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 24.2.8. For Grids, Meshes, or XGrids on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

In 2D, ESMF supports conservative regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of any number of logically rectangular tiles
- Unstructured meshes (ESMF_Mesh) composed of polygons with any number of sides
- An exchange grid (ESMF_XGrid)

In 3D, ESMF supports conservative regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of a single logically rectangular tile
- Unstructured Meshes (ESMF_Mesh) composed of hexahedrons (e.g. cubes) and tetrahedrons

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are optionally (via a flag) either ignored or return an error
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source or destination Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the conservative method the user should create their Fields on the center stagger location (ESMF_STAGGERLOC_CENTER in 2D or ESMF_STAGGERLOC_CENTER_VCENTER in 3D) for Grids or the element location (ESMF_MESHLOC_ELEMENT) for Meshes. For Grids, the corner stagger location (ESMF_STAGGERLOC_CORNER in 2D or ESMF_STAGGERLOC_CORNER_VFACE in 3D) must contain coordinates describing the outer perimeter of the Grid cells. This method will also work when the source or destination Field is built on an XGrid.

24.2.6 Interpolation methods: second-order conservative

Like the first-order conservative method, this method's goal is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 24.2.7.) The difference between the first and second-order conservative methods is that the second-order takes the source gradient into account, so it yields a smoother destination field that typically better matches the source field. This difference between the first and second-order methods is particularly apparent when going from a coarse source grid to a finer destination grid. Another difference is that the second-order method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the second-order method, the destination field will contain values less than 0.0. The implementation of this method is based on the one described in this paper [**?**].

Like the first-order method, the values for a particular destination cell with the second-order method are a combination of the values of the intersecting source cells with the weight of a given source cell's contribution to the total being

the amount that that source cell overlaps with the destination cell. However, with the second-order conservative interpolation there are additional terms that take into account the gradient of the field across the source cell. In particular, the value d for a given destination cell is calculated as:

$d = \sum_{i}^{intersecting-source-cells}(s_i + \nabla s_i \cdot (c_{si} - c_d))$

Where:

s_i is the intersecting source cell value.

∇s_i is the intersecting source cell gradient.

c_{si} is the intersecting source cell centroid.

c_d is the destination cell centroid.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 24.2.8. For Grids, Meshes, or XGrids on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

In 2D, ESMF supports second-order conservative regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports second-order conservative regridding between any combination of the following:

- NONE

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are optionally (via a flag) either ignored or return an error
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source or destination Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the second-order conservative method the user should create their Fields on the center stagger location (`ESMF_STAGGERLOC_CENTER` for Grids or the element location (`ESMF_MESHLOC_ELEMENT`) for Meshes. For Grids, the corner stagger location (`ESMF_STAGGERLOC_CORNER` in 2D must contain coordinates describing the outer perimeter of the Grid cells. This method will also work when the source or destination Field is built on an XGrid.

24.2.7 Conservation

Conservation means that the following equation will hold: $\sum^{all-source-cells}(V_{si} * A_{si}) = \sum^{all-destination-cells}(V_{dj} * A_{dj})$, where V is the variable being regridded and A is the area of a cell. The subscripts s and d refer to source and destination values, and the i and j are the source and destination grid cell indices (flattening the arrays to 1 dimension).

If the user doesn't specify a cell areas in the involved Grids or Meshes, then the areas (A) in the above equation are calculated by ESMF. For Cartesian grids, the area of a grid cell calculated by ESMF is the typical Cartesian area. For grids on a sphere, cell areas are calculated by connecting the corner coordinates of each grid cell with great circles. If the user does specify the areas in the Grid or Mesh, then the conservation will be adjusted to work for the areas provided by the user. This means that the above equation will hold, but with the areas (A) being the ones specified by the user.

The user should be aware that because of the conservation relationship between the source and destination fields, the more the total source area differs from the total destination area the more the values of the source field will differ from the corresponding values of the destination field, likely giving a higher interpolation error. It is best to have the total source and destination areas the same (this will automatically be true if no user areas are specified). For source and destination grids that only partially overlap, the overlapping regions of the source and destination should be the same.

24.2.8 The effect of normalization options on integrals and values produced by conservative methods

It is important to note that by default (i.e. using destination area normalization) conservative regridding doesn't normalize the interpolation weights by the destination fraction. This means that for a destination grid which only partially overlaps the source grid the destination field that is output from the regrid operation should be divided by the corresponding destination fraction to yield the true interpolated values for cells which are only partially covered by the source grid. The fraction also needs to be included when computing the total source and destination integrals. (To include the fraction in the conservative weights, the user can specify the fraction area normalization type. This can be done by specifying `normType=ESMF_NORMTYPE_FRACAREA` when invoking `ESMF_FieldRegridStore()`.)

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying `normType=ESMF_NORMTYPE_DSTAREA`), if a destination field extends outside the unmasked source field, then the values of the cells which extend partway outside the unmasked source field are decreased by the fraction they extend outside. To correct these values, the destination field (`dst_field`) resulting from the `ESMF_FieldRegrid()` call can be divided by the destination fraction `dst_frac` from the `ESMF_FieldRegridStore()` call. The following pseudocode demonstrates how to do this:

```
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_field(i)=dst_field(i)/dst_frac(i)
   end if
end for
```

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying `normType=ESMF_NORMTYPE_DSTAREA`), the following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the `ESMF_FieldRegrid()` call, the destination area (`dst_area`) from the `ESMF_FieldRegridGetArea()` call, and the destination fraction (`dst_frac`) from the `ESMF_FieldRegridStore()` call. As shown in the previous paragraph, it also shows how to adjust the destination field (`dst_field`) resulting from the `ESMF_FieldRegrid()` call by the fraction (`dst_frac`) from the `ESMF_FieldRegridStore()` call:

```
dst_total=0.0
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_total=dst_total+dst_field(i)*dst_area(i)
      dst_field(i)=dst_field(i)/dst_frac(i)
      ! If mass computed here after dst_field adjust, would need to be:
      ! dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
   end if
end for
```

For weights generated using fraction area normalization (by specifying `normType=ESMF_NORMTYPE_FRACAREA`), no adjustment of the destination field is necessary. The following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the `ESMF_FieldRegrid()` call, the destination area (`dst_area`) from the `ESMF_FieldRegridGetArea()` call, and the destination fraction (`dst_frac`) from the `ESMF_FieldRegridStore()` call:

```
dst_total=0.0
for each destination element i
     dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
end for
```

For both normalization types, the following pseudo-code shows how to compute the total source integral (`src_total`) given the source field values (`src_field`), the source area (`src_area`) from the `ESMF_FieldRegridGetArea()` call, and the source fraction (`src_frac`) from the `ESMF_FieldRegridStore()` call:

```
src_total=0.0
for each source element i
   src_total=src_total+src_field(i)*src_area(i)*src_frac(i)
end for
```

24.2.9 Great circle cells

For Grids, Meshes, or XGrids on a sphere some combinations of interpolation options (e.g. first and second-order conservative methods) use cells whose edges are great circles. This section describes some behavior that the user may not expect from these cells and some potential solutions.

A great circle edge isn't necessarily the same as a straight line in latitude longitude space. For small edges, this difference will be small, but for long edges it could be significant. This means if the user expects cell edges as straight lines in latitude longitude space, they should avoid using one large cell with long edges to compute an average over a region (e.g. over an ocean basin).

Also, the user should also avoid using cells that contain one edge that runs half way or more around the earth, because the regrid weight calculation assumes the edge follows the shorter great circle path. There isn't a unique great circle edge defined between points on the exact opposite side of the earth from one another (antipodal points). However, the user can work around both of these problem by breaking the long edge into two smaller edges by inserting an extra node, or by breaking the large target grid cells into two or more smaller grid cells. This allows the application to resolve the ambiguity in edge direction.

24.2.10 Masking

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

The user may mask out points in the source Field or destination Field or both. To do masking the user sets mask information in the Grid (see **??**), Mesh (see **??**), or LocStream (see **??**) upon which the Fields passed into the `ESMF_FieldRegridStore()` call are built. The `srcMaskValues` and `dstMaskValues` arguments to that call can then be used to specify which values in that mask information indicate that a location should be masked out. For example, if `dstMaskValues` is set to (/1,2/), then any location that has a value of 1 or 2 in the mask information of the Grid, Mesh or LocStream upon which the destination Field is built will be masked out.

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

24.2.11 Extrapolation methods: overview

Extrapolation in the ESMF regridding system is a way to automatically fill some or all of the destination points left unmapped by a regridding method. Weights generated by the extrapolation method are merged into the regridding weights to yield one set of weights or routehandle. Currently extrapolation is not supported with conservative regridding methods, because doing so would result in non-conservative weights.

24.2.12 Extrapolation methods: nearest source to destination

In nearest source to destination extrapolation (`ESMF_EXTRAPMETHOD_NEAREST_STOD`) each unmapped destination point is mapped to the closest source point. A given source point may map to multiple destination points, but no destination point will receive input from more than one source point. If two points are equally close, then the point with the smallest sequence index is arbitrarily used (i.e. the point which would have the smallest index in the weight matrix).

If there is at least one unmasked source point, then this method is expected to fill all unmapped points.

24.2.13 Extrapolation methods: inverse distance weighted average

In inverse distance weighted average extrapolation (`ESMF_EXTRAPMETHOD_NEAREST_IDAVG`) each unmapped destination point is the weighted average of the closest N source points. The weight is the reciprocal of the distance

of the source point from the destination point raised to a power P. All the weights contributing to one destination point are normalized so that they sum to 1.0. The user can choose N and P when using this method, but defaults are also provided. For example, when calling `ESMF_FieldRegridStore()` N is specified via the argument `extrapNumSrcPnts` and P is specified via the argument `extrapDistExponent`.

If there is at least one unmasked source point, then this method is expected to fill all unmapped points.

24.2.14 Extrapolation methods: creep fill

In creep fill extrapolation (`ESMF_EXTRAPMETHOD_CREEP`) unmapped destination points are filled by repeatedly moving data from mapped locations to neighboring unmapped locations for a user specified number of levels. More precisely, for each creeped point, its value is the average of the values of the point's immediate neighbors in the previous level. For the first level, the values are the average of the point's immediate neighbors in the destination points mapped by the regridding method. The number of creep levels is specified by the user. For example, in ESMF_FieldRegridStore() this number of levels is specified via the `extrapNumLevels` argument.

Unlike some extrapolation methods, creep fill does not necessarily fill all unmapped destination points. Unfilled destination points are still unmapped with the usual consequences (e.g. they won't be in the resulting regridding matrix, and won't be set by the application of the regridding weights).

Because it depends on the connections in the destination grid, creep fill extrapolation is not supported when the destination Field is built on a Location Stream (ESMF_LocStream). Also, creep fill is currently only supported for 2D Grids, Meshes, or XGrids

24.2.15 Unmapped destination points

If a destination point can't be mapped to a location in the source grid by the combination of regrid method and optional follow on extrapolation method, then the user has two choices. The user may ignore those destination points that can't be mapped by setting the `unmappedaction` argument to `ESMF_UNMAPPEDACTION_IGNORE` (Ignored points won't be included in the sparse matrix or routeHandle). If the user needs the unmapped points, the `ESMF_FieldRegridStore()` method has the capability to return a list of them using the `unmappedDstList` argument. In addition to ignoring them, the user also has the option to return an error if unmapped destination points exist. This is the default behavior, so the user can either not set the `unmappedaction` argument or the user can set it to `ESMF_UNMAPPEDACTION_ERROR`. Currently, the unmapped destination error detection doesn't work with the nearest destination to source regrid method (`ESMF_REGRIDMETHOD_NEAREST_DTOS`), so with this method the regridding behaves as if `ESMF_UNMAPPEDACTION_IGNORE` is always on.

24.2.16 Spherical grids and poles

In the case that the Grid is on a sphere (`coordSys=ESMF_COORDSYS_SPH_DEG` or `ESMF_COORDSYS_SPH_RAD`) then the coordinates given in the Grid are interpreted as latitude and longitude values. The coordinates can either be in degrees or radians as indicated by the `coordSys` flag set during Grid creation. As is true with many global models, this application currently assumes the latitude and longitude refer to positions on a perfect sphere, as opposed to a more complex and accurate representation of the Earth's true shape such as would be used in a GIS system. (ESMF's current user base doesn't require this level of detail in representing the Earth's shape, but it could be added in the future if necessary.)

For Grids on a sphere, the regridding occurs in 3D Cartesian to avoid problems with periodicity and with the pole singularity. This library supports four options for handling the pole region (i.e. the empty area above the top row of

the source grid or below the bottom row of the source grid). Note that all of these pole options currently only work for the Fields build on the Grid class.

The first option is to leave the pole region empty (`polemethod=ESMF_POLEMETHOD_NONE`), in this case if a destination point lies above or below the top row of the source grid, it will fail to map, yielding an error (unless `unmappedaction=ESMF_UNMAPPEDACTION_IGNORE` is specified).

With the next two options (`ESMF_POLEMETHOD_ALLAVG` and `ESMF_POLEMETHOD_NPNTAVG`), the pole region is handled by constructing an artificial pole in the center of the top and bottom row of grid points and then filling in the region from this pole to the edges of the source grid with triangles. The pole is located at the average of the position of the points surrounding it, but moved outward to be at the same radius as the rest of the points in the grid. The difference between the two artificial pole options is what value is used at the pole. The option (`polemethod=ESMF_POLEMETHOD_ALLAVG`) sets the value at the pole to be the average of the values of all of the grid points surrounding the pole. The option (`polemethod=ESMF_POLEMETHOD_NPNTAVG`) allows the user to choose a number N from 1 to the number of source grid points around the pole. The value N is set via the argument `regridPoleNPnts`. For each destination point, the value at the pole is then the average of the N source points surrounding that destination point.

The last option (`polemethod=ESMF_POLEMETHOD_TEETH`) does not construct an artificial pole, instead the pole region is covered by connecting points across the top and bottom row of the source Grid into triangles. As this makes the top and bottom of the source sphere flat, for a big enough difference between the size of the source and destination pole regions, this can still result in unmapped destination points. Only pole option `ESMF_POLEMETHOD_NONE` is currently supported with the conservative interpolation methods (e.g. `regridmethod=ESMF_REGRIDMETHOD_CONSERVE`) and with the nearest neighbor interpolation options (e.g. `regridmethod=ESMF_REGRIDMETHOD_NEAREST_STOD`).

Regrid Method	Line Type	
	ESMF_LINETYPE_CART	ESMF_LINETYPE_GREAT_CIRCLE
ESMF_REGRIDMETHOD_BILINEAR	Y*	Y
ESMF_REGRIDMETHOD_PATCH	Y*	Y
ESMF_REGRIDMETHOD_NEAREST_STOD	Y*	N
ESMF_REGRIDMETHOD_NEAREST_DTOS	Y*	N
ESMF_REGRIDMETHOD_CONSERVE	N/A	Y*
ESMF_REGRIDMETHOD_CONSERVE_2ND	N/A	Y*

Table 2: Line Type Support by Regrid Method (* indicates the default)

Another variation in the regridding supported with spherical grids is **line type**. This is controlled in the `ESMF_FieldRegridStore()` method by the `lineType` argument. This argument allows the user to select the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated, for example in bilinear interpolation the distances are used to calculate the weights and the cell edges are used to determine to which source cell a destination point should be mapped.

ESMF currently supports two line types: ESMF_LINETYPE_CART and ESMF_LINETYPE_GREAT_CIRCLE. The ESMF_LINETYPE_CART option specifies that the line between two points follows a straight path through the 3D Cartesian space in which the sphere is embedded. Distances are measured along this 3D Cartesian line. Under this option cells are approximated by planes in 3D space, and their boundaries are 3D Cartesian lines between their corner points. The ESMF_LINETYPE_GREAT_CIRCLE option specifies that the line between two points follows a great circle path along the sphere surface. (A great circle is the shortest path between two points on a sphere.) Distances are measured along the great circle path. Under this option cells are on the sphere surface, and their boundaries are great circle paths between their corner points.

Figure 24.2.16 shows which line types are supported for each regrid method as well as the defaults (indicated by *).

24.2.17 Troubleshooting guide

The below is a list of problems users commonly encounter with regridding and potential solutions. This is by no means an exhaustive list, so if none of these problems fit your case, or if the solutions don't fix your problem, please feel free to email esmf support (esmf_support@ucar.edu).

Problem: Regridding is too slow.

Possible Cause: The `ESMF_FieldRegridStore()` method is called more than is necessary.
The `ESMF_FieldRegridStore()` operation is a complex one and can be relatively slow for some cases (large Grids, 3D grids, etc.)

Solution: Reduce the number of `ESMF_FieldRegridStore()` calls to the minimum necessary. The routeHandle generated by the `ESMF_FieldRegridStore()` call depends on only four factors: the stagger locations that the input Fields are created on, the coordinates in the Grids the input Fields are built on at those stagger locations, the padding of the input Fields (specified by the `totalWidth` arguments in `FieldCreate`) and the size of the tensor dimensions in the input Fields (specified by the `ungridded` arguments in `FieldCreate`). For any pair of Fields which share these attributes with the Fields used in the `ESMF_FieldRegridStore` call the same routeHandle can be used. Note that the data in the Fields does NOT matter, the same routeHandle can be used no matter how the data in the Fields changes.

In particular:

- If Grid coordinates do not change during a run, then the `ESMF_FieldRegridStore()` call can be done once between a pair of Fields at the beginning and the resulting routeHandle used for each timestep during the run.
- If a pair of Fields was created with exactly the same arguments to `ESMF_FieldCreate()` as the pair of Fields used during an `ESMF_FieldRegridStore()` call, then the resulting routeHandle can also be used between that pair of Fields.

Problem: Distortions in destination Field at periodic boundary.

Possible Cause: The Grid overlaps itself. With a periodic Grid, the regrid system expects the first point to not be a repeat of the last point. In other words, regrid constructs its own connection and overlap between the first and last points of the periodic dimension and so the Grid doesn't need to contain these. If the Grid does, then this can cause problems.

Solution: Define the Grid so that it doesn't contain the overlap point. This typically means simply making the Grid one point smaller in the periodic dimension. If a Field constructed on the Grid needs to contain these overlap points then the user can use the `totalWidth` arguments to include this extra padding in the Field. Note, however, that the regrid won't update these extra points, so the user will have to do a copy to fill the points in the overlap region in the Field.

24.2.18 Restrictions and Future Work

This section contains restrictions that apply to the entire regridding system. For restrictions that apply to just one interpolation method, see the section corresponding to that method above.

- **Regridding doesn't work on a Field created on a Grid with an arbitrary distribution:** Using a Field built on a Grid with an arbitrary distribution will cause the regridding to stop with an error.

24.2.19 Design and implementation notes

The ESMF regrid weight calculation functionality has been designed to enable it to support a wide range of grid and interpolation types without needing to support each individual combination of source grid type, destination grid type, and interpolation method. To avoid the quadratic growth of the number of pairs of grid types, all grids are converted to a common internal format and the regrid weight calculation is performed on that format. This vastly reduces the variety of grids that need to be supported in the weight calculations for each interpolation method. It also has the added benefit of making it straightforward to add new grid types and to allow them to work with all the existing grid types. To hook into the existing weight calculation code, the new type just needs to be converted to the internal format.

The internal grid format used by the ESMF regrid weight calculation is a finite element unstructured mesh. This was chosen because it was the most general format and all the others could be converted to it. The ESMF finite element unstructured mesh (ESMF FEM) is similar in some respects to the SIERRA [**?**] package developed at Sandia National Laboratory. The ESMF code relies on some of the same underlying toolkits (e.g. Zoltan [**?**] library for calculating mesh partitions) and adds a layer on top that allows the calculation of regrid weights and some mesh operations (e.g. mesh redistribution) that ESMF needs. The ESMF FEM has similar notions to SIERRA about the basic structure of the mesh entities, fields, iteration and a similar notion of parallel distribution.

Currently we use the ESMF FEM internal mesh to hold the structure of our Mesh class and in our regrid weight calculation. The parts of the internal FEM code that are used/tested by ESMF are the following:

- The creation of a mesh composed of triangles and quadrilaterals or hexahedrons and tetrahedrons.
- The object relations data base to store the connections between objects (e.g. which element contains which nodes).
- The fields to hold data (e.g. coordinates). We currently only build fields on nodes and elements (2D and 3D).
- Iteration to move through mesh entities.
- The parallel code to maintain information about the distribution of the mesh across processors and to communicate data between parts of the mesh on different processors (i.e. halos).

24.3 File-based Regrid API

24.3.1 ESMF_RegridWeightGen - Generate regrid weight file from grid files

INTERFACE:

```
! Private name; call using ESMF_RegridWeightGen()
subroutine ESMF_RegridWeightGenFile(srcFile, dstFile, &
  weightFile, rhFile, regridmethod, polemethod, regridPoleNPnts, lineType, normType, &
  extrapMethod, extrapNumSrcPnts, extrapDistExponent, extrapNumLevels, &
  unmappedaction, ignoreDegenerate, srcFileType, dstFileType, &
  srcRegionalFlag, dstRegionalFlag, srcMeshname, dstMeshname,  &
```

```
    srcMissingvalueFlag, srcMissingvalueVar, &
    dstMissingvalueFlag, dstMissingvalueVar, &
    useSrcCoordFlag, srcCoordinateVars, &
    useDstCoordFlag, dstCoordinateVars, &
    useSrcCornerFlag, useDstCornerFlag, &
    useUserAreaFlag, largefileFlag, &
    netcdf4fileFlag, weightOnlyFlag, &
    tileFilePath, &
    verboseFlag, checkFlag, rc)
```

ARGUMENTS:

```
  character(len=*),             intent(in)            :: srcFile
  character(len=*),             intent(in)            :: dstFile
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),             intent(in),  optional :: weightFile
  character(len=*),             intent(in),  optional :: rhFile
  type(ESMF_RegridMethod_Flag), intent(in),  optional :: regridmethod
  type(ESMF_PoleMethod_Flag),   intent(in),  optional :: polemethod
  integer,                      intent(in),  optional :: regridPoleNPnts
  type(ESMF_LineType_Flag),     intent(in),  optional :: lineType
  type(ESMF_NormType_Flag),     intent(in),  optional :: normType
  type(ESMF_ExtrapMethod_Flag),   intent(in),    optional :: extrapMethod
  integer,                        intent(in),    optional :: extrapNumSrcPnts
  real,                           intent(in),    optional :: extrapDistExponent
  integer,                      intent(in), optional :: extrapNumLevels
  type(ESMF_UnmappedAction_Flag),intent(in), optional :: unmappedaction
  logical,                      intent(in),  optional :: ignoreDegenerate
  type(ESMF_FileFormat_Flag),   intent(in),  optional :: srcFileType
  type(ESMF_FileFormat_Flag),   intent(in),  optional :: dstFileType
  logical,                      intent(in),  optional :: srcRegionalFlag
  logical,                      intent(in),  optional :: dstRegionalFlag
  character(len=*),             intent(in),  optional :: srcMeshname
  character(len=*),             intent(in),  optional :: dstMeshname
  logical,                      intent(in),  optional :: srcMissingValueFlag
  character(len=*),             intent(in),  optional :: srcMissingValueVar
  logical,                      intent(in),  optional :: dstMissingValueFlag
  character(len=*),             intent(in),  optional :: dstMissingValueVar
  logical,                      intent(in),  optional :: useSrcCoordFlag
  character(len=*),             intent(in),  optional :: srcCoordinateVars(:)
  logical,                      intent(in),  optional :: useDstCoordFlag
  character(len=*),             intent(in),  optional :: dstCoordinateVars(:)
  logical,                      intent(in),  optional :: useSrcCornerFlag
  logical,                      intent(in),  optional :: useDstCornerFlag
  logical,                      intent(in),  optional :: useUserAreaFlag
  logical,                      intent(in),  optional :: largefileFlag
  logical,                      intent(in),  optional :: netcdf4fileFlag
  logical,                      intent(in),  optional :: weightOnlyFlag
  logical,                      intent(in),  optional :: verboseFlag
  character(len=*),             intent(in),  optional :: tileFilePath
  logical,                      intent(in),  optional :: checkFlag
  integer,                      intent(out), optional :: rc
```

DESCRIPTION:

This subroutine provides the same function as the ESMF_RegridWeightGen application described in Section 12. It takes two grid files in NetCDF format and writes out an interpolation weight file also in NetCDF format. The interpolation weights can be generated with the bilinear (24.2.1), higher-order patch (24.2.2), or first order conservative (24.2.5) methods. The grid files can be in one of the following four formats:

- The SCRIP format (12.8.1)
- The native ESMF format for an unstructured grid (12.8.2)
- The CF Convention Single Tile File format (12.8.3)
- The proposed CF Unstructured grid (UGRID) format (12.8.4)
- The GRIDSPEC Mosaic File format (12.8.5)

The weight file is created in SCRIP format (12.9). The optional arguments allow users to specify various options to control the regrid operation, such as which pole option to use, whether to use user-specified area in the conservative regridding, or whether ESMF should generate masks using a given variable's missing value. There are also optional arguments specific to a certain type of the grid file. All the optional arguments are similar to the command line arguments for the ESMF_RegridWeightGen application (12.6). The acceptable values and the default value for the optional arguments are listed below.

The arguments are:

srcFile The source grid file name.

dstFile The destination grid file name.

weightFile The interpolation weight file name.

[rhFile] The RouteHandle file name.

[regridmethod] The type of interpolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_REGRIDMETHOD_BILINEAR.

[polemethod] A flag to indicate which type of artificial pole to construct on the source Grid for regridding. Please see Section **??** for a list of valid options. The default value varies depending on the regridding method and the grid type and format.

[regridPoleNPnts] If polemethod is set to ESMF_POLEMETHOD_NPNTAVG, this argument is required to specify how many points should be averaged over at the pole.

[lineType] This argument controls the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated. As would be expected, this argument is only applicable when srcField and dstField are built on grids which lie on the surface of a sphere. Section **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** has the defaults by line type. Figure 24.2.16 shows which line types are supported for each regrid method as well as showing the default line type by regrid method.

[normType] This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with regridmethod=ESMF_REGRIDMETHOD_CONSERVE. Please see Section **??** for a list of valid options. If not specified normType defaults to ESMF_NORMTYPE_DSTAREA.

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

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

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

[unmappedaction] Specifies what should happen if there are destination points that can't be mapped to a source cell. Please see Section **??** for a list of valid options. If not specified, `unmappedaction` defaults to `ESMF_UNMAPPEDACTION_ERROR`.

[ignoreDegenerate] Ignore degenerate cells when checking the input Grids or Meshes for errors. If this is set to true, then the regridding proceeds, but degenerate cells will be skipped. If set to false, a degenerate cell produces an error. If not specified, `ignoreDegenerate` defaults to false.

[srcFileType] The file format of the source grid. Please see Section **??** for a list of valid options. If not specifed, the program will determine the file format automatically.

[dstFileType] The file format of the destination grid. Please see Section **??** for a list of valid options. If not specifed, the program will determine the file format automatically.

[srcRegionalFlag] If .TRUE., the source grid is a regional grid, otherwise, it is a global grid. The default value is .FALSE.

[dstRegionalFlag] If .TRUE., the destination grid is a regional grid, otherwise, it is a global grid. The default value is .FALSE.

[srcMeshname] If the source file is in UGRID format, this argument is required to define the dummy variable name in the grid file that contains the mesh topology info.

[dstMeshname] If the destination file is in UGRID format, this argument is required to define the dummy variable name in the grid file that contains the mesh topology info.

[srcMissingValueFlag] If .TRUE., the source grid mask will be constructed using the missing values of the variable defined in `srcMissingValueVar`. This flag is only used for the grid defined in the GRIDSPEC or the UGRID file formats. The default value is .FALSE..

[srcMissingValueVar] If `srcMissingValueFlag` is .TRUE., the argument is required to define the variable name whose missing values will be used to construct the grid mask. It is only used for the grid defined in the GRIDSPEC or the UGRID file formats.

[dstMissingValueFlag] If .TRUE., the destination grid mask will be constructed using the missing values of the variable defined in `dstMissingValueVar`. This flag is only used for the grid defined in the GRIDSPEC or the UGRID file formats. The default value is .FALSE..

[dstMissingValueVar] If `dstMissingValueFlag` is .TRUE., the argument is required to define the variable name whose missing values will be used to construct the grid mask. It is only used for the grid defined in the GRIDSPEC or the UGRID file formats.

[useSrcCoordFlag] If .TRUE., the coordinate variables defined in `srcCoordinateVars` will be used as the longitude and latitude variables for the source grid. This flag is only used for the GRIDSPEC file format. The default is .FALSE.

[srcCoordinateVars] If `useSrcCoordFlag` is .TRUE., this argument defines the longitude and ! latitude variables in the source grid file to be used for the regrid. This argument is only used when the grid file is in GRIDSPEC format. `srcCoordinateVars` should be a array of 2 elements.

[useDstCoordFlag] If .TRUE., the coordinate variables defined in `dstCoordinateVars` will be used as the longitude and latitude variables for the destination grid. This flag is only used for the GRIDSPEC file format. The default is .FALSE.