Earth System Modeling Framework

ESMF Reference Manual for Fortran

Version 5.2

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, Brian Eaton, Bob Hallberg, Tom Henderson, Chris Hill, Mark Iredell, Rob Jacob, Phil Jones, Erik Kluzek, Brian Kauffman, Jay Larson, Peggy Li, Fei Liu, John Michalakes, Sylvia Murphy, David Neckels, Ryan O Kuinghttons, Bob Oehmke, Chuck Panaccione, Jim Rosinski, Will Sawyer, Earl Schwab, Shepard Smithline, Walter Spector, Don Stark, Max Suarez, Spencer Swift, Gerhard Theurich, Atanas Trayanov, Silverio Vasquez, Jon Wolfe, Weiyu Yang, Mike Young, Leonid Zaslavsky

July 15, 2011

---

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 insitutional and national bounds.

2 The ESMF Reference Manual for Fortran

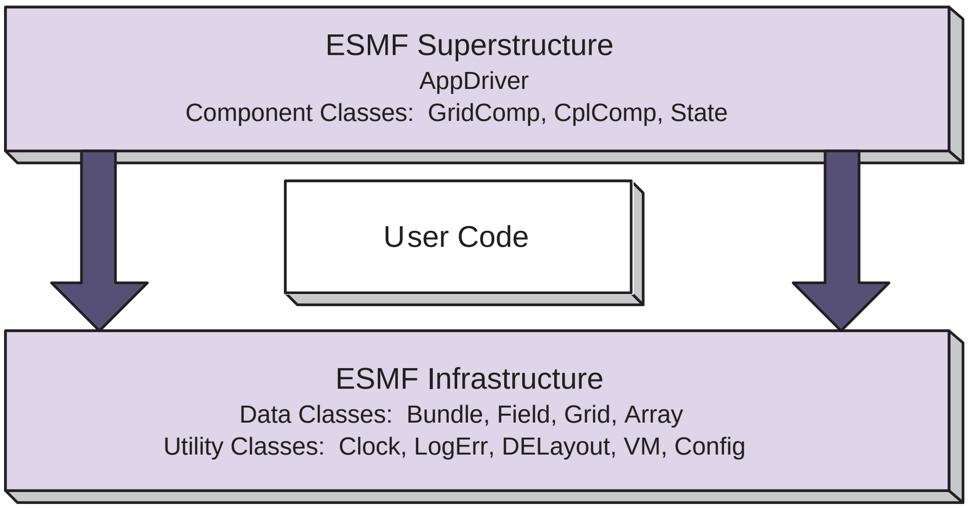
ESMF has a complete set of Fortran interfaces and some C interfaces. This *ESMF Reference Manual* is a listing of ESMF interfaces for Fortran.[1]

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.

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.

[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.



3 How to Contact User Support and Find Additional Information

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

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@list.woc.noaa.gov.

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.* [20]

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 (involving 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 10, 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 an hierarchy of classes that contain model data. The operations that are performed on model data, such as regridding, redistribution, and halo updates, are methods of these classes.

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

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

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

- **DistGrid** A DistGrid represents the index space associated with a grid. It is a useful abstraction because often a full specification of grid coordinates is not necessary to define data communication patterns. The DistGrid contains information about the sequence and connectivity of data points, which is sufficient information for many operations. Arrays are defined on DistGrids.

- **Array** An Array defines how the index space described in the DistGrid is associated with the VAS of each PET. This association considers the type, kind and rank of the indexed data. Fields are defined on Arrays.

- **Grid** A Grid is an abstraction of a physical space. It associates a coordinate system, a set of coordinates, and a topology to a collection of grid cells. Grids in ESMF are comprised of DistGrids plus additional coordinate information.

- **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 Overall Rules and Behavior

7.1 Local and Global Views and Associated Conventions

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

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

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

7.2 Allocation Rules

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

ESMF methods that allocate their own space for data will deallocate that space when the object is destroyed. Methods which accept a user-allocated buffer, for example `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.

7.3 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 assignemnt 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 on 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.

7.4 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 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.

8.1 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.

9 Master List of Constants

9.1 ESMF_ALARMLIST

This flag is documented in section 41.2.1.

9.2 ESMF_DIM_ARB

DESCRIPTION:
An integer named constant which is used to indicate that a particular dimension is arbitrarily distributed.

9.3 ESMF_ATTGETCOUNT

This flag is documented in section 35.5.1.

9.4 ESMF_ATTRECONCILE

DESCRIPTION:
Indicate whether or not to handle metadata (Attributes) in `ESMF_StateReconcile()`.

The type of this flag is:
`type(ESMF_AttReconcileFlag)`
The valid values are:

ESMF_ATTRECONCILE_ON Attribute reconciliation will be turned on.

ESMF_ATTRECONCILE_OFF Attribute reconciliation will be turned off.

9.5 ESMF_ATTTREE

This flag is documented in section 35.5.2.

9.6 ESMF_ATTWRITE

This flag is documented in section 35.5.3.

9.7 ESMF_CALKIND

This flag is documented in section 37.2.1.

9.8 ESMF_COMPTYPE

DESCRIPTION:
Indicate the type of a Component.
The type of this flag is:
`type(ESMF_CompType_Flag)`
The valid values are:

ESMF_COMPTYPE_GRID A `ESMF_GridComp` object.

ESMF_COMPTYPE_CPL A `ESMF_CplComp` objects.

9.9 ESMF_CONTEXT

DESCRIPTION:
Indicates the type of VM context in which a Component will be executing its standard methods.
The type of this flag is:
`type(ESMF_Context_Flag)`
The valid values are:

ESMF_CONTEXT_OWN_VM The component is running in its own, separate VM context. Resources are inherited from the parent but can be arranged to fit the component's requirements.

ESMF_CONTEXT_PARENT_VM The component uses the parent's VM for resource management. Compared to components that use their own VM context components that run in the parent's VM context are more light-weight with respect to the overhead of calling into their initialize, run and finalize methods. Furthermore, VM-specific properties remain unchanged when going from the parent component to the child component. These properties include the MPI communicator, the number of PETs, the PET labeling, communication attributes, threading-level.

9.10 ESMF_COORDSYS

This flag is documented in section 27.2.1.

9.11 ESMF_COPY

DESCRIPTION:
Indicates which type of copy behavior is used when copying ESMF objects.
The type of this flag is:
`type(ESMF_Copy_Flag)`
The valid values are:

ESMF_COPY_ALIAS The destination of the copy becomes an alias of the source object.

ESMF_COPY_REFERENCE All local data of the object is coped by value, and pointer valued data is referenced.

ESMF_COPY_VALUE All data and pointers are copied to the destination object, nothing is referenced.

9.12 ESMF_DATACOPY

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

The type of this flag is:
`type(ESMF_DataCopy_Flag)`
The valid values are:

ESMF_DATACOPY_VALUE Copy the data item to another buffer.

ESMF_DATACOPY_REFERENCE Reference the data item.

9.13 ESMF_DECOMP

DESCRIPTION:
Indicates how DistGrid elements are decomposed over DEs.

The type of this flag is:
`type(ESMF_Decomp_Flag)`
The valid values are:

ESMF_DECOMP_BALANCED Decompose elements as balanced as possible across DEs. The maximum difference in number of elements per DE is 1, with the extra elements on the lower DEs.

ESMF_DECOMP_CYCLIC Decompose elements cyclically across DEs.

ESMF_DECOMP_DEFAULT Use default decomposition behavior. Currently equal to `ESMF_DECOMP_BALANCED`.

ESMF_DECOMP_RESTFIRST Divide elements over DEs. Assign the rest of this division to the first DE.

ESMF_DECOMP_RESTLAST Divide elements over DEs. Assign the rest of this division to the last DE.

9.14 ESMF_DIRECTION

This flag is documented in section 40.2.1.

9.15 ESMF_DISTGRIDMATCH

This flag is documented in section 31.2.1.

9.16 ESMF_END

This flag is documented in section 14.2.1.

9.17 ESMF_FIELDSTATUS

This flag is documented in section 22.2.1.

9.18 ESMF_FILEFORMAT

This flag is documented in section 29.2.2.

9.19 ESMF_GEOMTYPE

DESCRIPTION:
Different types of geometries upon which an ESMF Field or ESMF Fieldbundle may be built.

The type of this flag is:

`type(ESMF_GeomType_Flag)`

The valid values are:

ESMF_GEOMTYPE_GRID An ESMF_Grid, a structured grid composed of one or more logically rectangular tiles

ESMF_GEOMTYPE_MESH An ESMF_Mesh, an unstructured grid

ESMF_GEOMTYPE_XGRID An ESMF_XGrid, an exchange grid

ESMF_TYPEKIND_LOCSTREAM An ESMF_LocStream, a disconnected series of points with associated key values

9.20 ESMF_GRIDCONN

This flag is documented in section 27.2.2.

9.21 ESMF_GRIDITEM

This flag is documented in section 27.2.3.

9.22 ESMF_GRIDMATCH

This flag is documented in section 27.2.4.

9.23 ESMF_GRIDSTATUS

This flag is documented in section 27.2.5.

9.24 ESMF_INDEX

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

The type of this flag is:

`type(ESMF_Index_Flag)`

The valid values are:

ESMF_INDEX_DELOCAL Indicates that DE-local index space starts at lower bound 1 for each DE.

ESMF_INDEX_GLOBAL Indicates that global indices are used. This means that DE-local index space starts at the global lower bound for each DE.

ESMF_INDEX_USER Indicates that the DE-local index bounds are explicitly set by the user.

9.25 ESMF_IOFMT

DESCRIPTION:
Indicates IO format options that are currently supported.

The type of this flag is:

`type(ESMF_IOFmtFlag)`

The valid values are:

ESMF_IOFMT_BIN Binary format.

ESMF_IOFMT_NETCDF NETCDF and PNETCDF format.

9.26 ESMF_KIND

DESCRIPTION:
Named constants to be used as *kind-parameter* in Fortran variable declarations. For example:

```
integer(ESMF_KIND_I4)       :: integerVariable
integer(kind=ESMF_KIND_I4)  :: integerVariable
real(ESMF_KIND_R4)          :: realVariable
real(kind=ESMF_KIND_R4)     :: realVariable
```

The Fortran standard does not mandate what numeric values correspond to actual number of bytes allocated for the various kinds. The following constants are defined by ESMF to be correct across the supported Fortran compilers. Note that not all compilers support every kind listed below; in particular 1 and 2 byte integers can be problematic.

The type of these named constants is:

`integer`

The named constants are:

ESMF_KIND_I1 Kind-parameter for 1 byte integer.

ESMF_KIND_I2 Kind-parameter for 2 byte integer.

ESMF_KIND_I4 Kind-parameter for 4 byte integer.

ESMF_KIND_I8 Kind-parameter for 8 byte integer.

ESMF_KIND_R4 Kind-parameter for 4 byte real.

ESMF_KIND_R8 Kind-parameter for 8 byte real.

9.27 ESMF_LOGERR

This flag is documented in section 43.2.1.

9.28 ESMF_LOGKIND

This flag is documented in section 43.2.2.

9.29 ESMF_LOGMSG

This flag is documented in section 43.2.3.

9.30 ESMF_MESHELEMTYPE

This flag is documented in section 29.2.1.

9.31 ESMF_METHOD

DESCRIPTION:
Specify standard ESMF Component method.

The type of this flag is:

`type(ESMF_Method_Flag)`

The valid values are:

ESMF_METHOD_FINALIZE Finalize method.

ESMF_METHOD_INITIALIZE Initialize method.

ESMF_METHOD_READRESTART ReadRestart method.

ESMF_METHOD_RUN Run method.

ESMF_METHOD_WRITERESTART WriteRestart method.

9.32 ESMF_PIN

This flag is documented in section 44.2.1.

9.33 ESMF_POLEMETHOD

This flag is documented in section 22.2.2.

9.34 ESMF_POLEKIND

This flag is documented in section 27.2.6.

9.35 ESMF_REDUCE

DESCRIPTION:
Indicates reduce operation

The type of this flag is:

`type(ESMF_Reduce_Flag)`

The valid values are:

ESMF_REDUCE_SUM Use arithmetic sum to add all data elements.

ESMF_REDUCE_MIN Determine the minimum of all data elements.

ESMF_REDUCE_MAX Determine the maximum of all data elements.

9.36 ESMF_REGION

DESCRIPTION:
Specifies various regions in the data layout of an Array or Field object.

The type of this flag is:

`type(ESMF_Region_Flag)`

The valid values are:

ESMF_REGION_TOTAL Total allocated memory.

ESMF_REGION_SELECT Region of operation-specific elements.

ESMF_REGION_EMPTY The empty region contains no elements.

9.37 ESMF_REGRIDMETHOD

This flag is documented in section 22.2.3.

9.38 ESMF_ROUTESYNC

DESCRIPTION:

Switch between blocking and non-blocking execution of RouteHandle based communication calls. Every RouteHandle based communication method contains an optional argument `routesyncflag` that is of type `ESMF_RouteSync_Flag`.

The type of this flag is:

`type(ESMF_RouteSync_Flag)`

The valid values are:

ESMF_ROUTESYNC_BLOCKING Execute a precomputed communication pattern in blocking mode. This mode guarantees that when the method returns all PET-local data transfers, both in-bound and out-bound, have finished.

ESMF_ROUTESYNC_NBSTART Start executing a precomputed communication pattern in non-blocking mode. When a method returns from being called in this mode, it guarantees that all PET-local out-bound data has been transferred. It is now safe for the user to overwrite out-bound data elements. No guarantees are made for in-bound data elements at this stage. It is unsafe to access these elements until a call in `ESMF_ROUTESYNC_NBTESTFINISH` mode has been issued and has returned with `finishedflag` equal to .true., or a call in `ESMF_ROUTESYNC_NBWAITFINISH` mode has been issued and has returned.

ESMF_ROUTESYNC_NBTESTFINISH Test whether the transfer of data of a precomputed communication pattern, started with `ESMF_ROUTESYNC_NBSTART`, has completed. Finish up as much as possible and set the `finishedflag` to `.true.` if *all* data operations have completed, or `.false.` if there are still outstanding transfers. Only after a `finishedflag` equal to `.true.` has been returned is it safe to access any of the in-bound data elements.

ESMF_ROUTESYNC_NBWAITFINISH Wait (i.e. block) until the transfer of data of a precomputed communication pattern, started with `ESMF_ROUTESYNC_NBSTART`, has completed. Finish up *all* data operations and set the returned `finishedflag` to `.true.`. It is safe to access any of the in-bound data elements once the call has returned.

ESMF_ROUTESYNC_CANCEL Cancel outstanding transfers for a precomputed communication pattern.

9.39 ESMF_SERVICEREPLY

This flag is documented in section 44.2.2.

9.40 ESMF_STAGGERLOC

This flag is documented in section 27.2.7.

9.41 ESMF_STARTREGION

DESCRIPTION:

Specifies the start of the effective halo region of an Array or Field object.

The type of this flag is:

`type(ESMF_StartRegion_Flag)`

The valid values are:

ESMF_STARTREGION_EXCLUSIVE Region of elements that are exclusively owned by the local DE.

ESMF_STARTREGION_COMPUTATIONAL User defined region, greater or equal to the exclusive region.

9.42 ESMF_STATEINTENT

This flag is documented in section 17.2.1.

9.43 ESMF_STATEITEM

This flag is documented in section 17.2.2.

9.44 ESMF_SYNC

DESCRIPTION:
Indicates method blocking behavior and PET synchronization for VM communication methods, as well as for standard Component methods, such as Initialize(), Run() and Finalize().

For VM communication calls the `ESMF_SYNC_BLOCKING` and `ESMF_SYNC_NONBLOCKING` modes provide behavior that is practically identical to the blocking and non-blocking communication calls familiar from MPI.

The details of how the blocking mode setting affects Component methods are more complex. This is a consequence of the fact that ESMF Components can be executed in threaded or non-threaded mode. However, in the default, non-threaded case, where an ESMF application runs as a pure MPI or mpiuni program, most of the complexity is removed.

See the **VM** item in 6.5 for an explanation of the PET and VAS concepts used in the following descriptions.

The type of this flag is:

`type(ESMF_Sync_Flag)`

The valid values are:

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

Component calls: The called method will block until all PETs of the VM have completed the operation.

For a non-threaded, pure MPI component the behavior is identical to calling a barrier before returning from the method. Generally this kind of rigid synchronization is not the desirable mode of operation for an MPI application, but may be useful for application debugging. In the opposite case, where all PETs of the component are running as threads in shared memory, i.e. in a single VAS, strict synchronization of all PETs is required to prevent race conditions.

ESMF_SYNC_VASBLOCKING *Communication calls:* Not available for communication calls.

Component calls: The called method will block each PET until all operations in the PET-local VAS have completed.

This mode is a combination of `ESMF_SYNC_BLOCKING` and `ESMF_SYNC_NONBLOCKING` modes. It provides a default setting that leads to the typically desirable behavior for pure MPI components as well as those that share address spaces between PETs.

For a non-threaded, pure MPI component each PET returns independent of the other PETs. This is generally the expected behavior in the pure MPI case where calling into a component method is practically identical to a subroutine call without extra synchronization between the processes.

In the case where some PETs of the component are running as threads in shared memory `ESMF_SYNC_VASBLOCKING` becomes identical to `ESMF_SYNC_BLOCKING` within thread groups, to prevent race conditions, while there is no synchronization between the thread groups.

ESMF_SYNC_NONBLOCKING *Communication calls:* The called method will not block but returns immediately after initiating the requested operation. It is unsafe to modify or use participating local data before all local operations have completed. Use the `ESMF_VMCommWait()` or `ESMF_VMCommQueueWait()` method to block the local PET until local data access is safe again.

Component calls: The behavior of this mode is fundamentally different for threaded and non-threaded components, independent on whether the components use shared memory or not. The `ESMF_SYNC_NONBLOCKING` mode is the most complex mode for calling component methods and should only be used if the extra control, described below, is absolutely necessary.

For non-threaded components (the ESMF default) calling a component method with `ESMF_SYNC_NONBLOCKING` is identical to calling it with `ESMF_SYNC_VASBLOCKING`. However, different than for `ESMF_SYNC_VASBLOCKING`, a call to `ESMF_GridCompWait()` or `ESMF_CplCompWait()` is required in order to deallocate memory internally allocated for the `ESMF_SYNC_NONBLOCKING` mode.

For threaded components the calling PETs of the parent component will not be blocked and return immediately after initiating the requested child component method. In this scenario parent and child components will run concurrently in identical VASs. This is the most complex mode of operation. It is unsafe to modify or use VAS local data that may be accessed by concurrently running components until the child component method has completed. Use the appropriate `ESMF_GridCompWait()` or `ESMF_CplCompWait()` method to block the local parent PET until the child component method has completed in the local VAS.

9.45 ESMF_TYPEKIND

DESCRIPTION:

Named constants used to indicate type and kind combinations supported by the overloaded ESMF interfaces. The corresponding Fortran kind-parameter constants are described in section 9.26.

The type of these named constants is:

`type(ESMF_TypeKind_Flag)`

The named constants are:

ESMF_TYPEKIND_I1 Indicates 1 byte integer.

ESMF_TYPEKIND_I2 Indicates 2 byte integer.

ESMF_TYPEKIND_I4 Indicates 4 byte integer.

ESMF_TYPEKIND_I8 Indicates 8 byte integer.

ESMF_TYPEKIND_R4 Indicates 4 byte real.

ESMF_TYPEKIND_R8 Indicates 8 byte real.

9.46 ESMF_UNMAPPEDACTION

DESCRIPTION:

Indicates what action to take with respect to unmapped destination points and the entries of the sparse matrix that correspond to these points.

The type of this flag is:

`type(ESMF_UnmappedAction_Flag)`

The valid values are:

ESMF_UNMAPPEDACTION_ERROR An error is issued when there exist destination points in a regridding operation that are not mapped by corresponding source points.

ESMF_UNMAPPEDACTION_IGNORE Destination points which do not have corresponding source points are ignored and zeros are used for the entries of the sparse matrix that is generated.

9.47 ESMF_VERSION

DESCRIPTION:
The following named constants define the precise version of ESMF in use.

ESMF_VERSION_BETASNAPSHOT Constant of type `logical` indicating beta snapshot phase (`.true.` for any version during the pre-release development phase, `.false.` for any released version of the software).

ESMF_VERSION_MAJOR Constant of type `integer` indicating the major version number (e.g. 5 for v5.2.0r).

ESMF_VERSION_MINOR Constant of type `integer` indicating the minor version number (e.g. 2 for v5.2.0r).

ESMF_VERSION_PATCHLEVEL Constant of type `integer` indicating the patch level of a specific revision (e.g. 0 for v5.2.0r, or 1 for v5.2.0rp1).

ESMF_VERSION_PUBLIC Constant of type `logical` indicating public vs. internal release status (e.g. `.true.` for v5.2.0r, or `.false.` for v5.2.0).

ESMF_VERSION_REVISION Constant of type `integer` indicating the revision number (e.g. 0 for v5.2.0r).

ESMF_VERSION_STRING Constant of type `character` holding the exact release version string (e.g. "5.2.0r").

9.48 ESMF_XGRIDSIDE

This flag is documented in section 30.2.1.

10 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.

Part II
Applications

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

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

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

11 ESMF_Info

11.1 Description

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

The application usage is as follows:

```
ESMF_Info  [--help]

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

12 ESMF_RegridWeightGen

12.1 Description

In addition to the online regridding functionality, the ESMF distribution also contains an exectuable for generating regridding weights. This tool reads in two grid files and outputs weights for interpolation between the two grids. The input and output files are all in NetCDF format. The grid files are either in the same format as is used as an input to SCRIP [13], or in the ESMF unstructured grid format 29.3.5. The weight file is the same format as is output by SCRIP. The interpolation weights can be generated with the bilinear, patch, or first order conservative methods decribed below. Masking is supported for 2D logically rectangular (i.e. with grid_rank=2) grids in the SCRIP format. This application can do regrid weight generation from a global or regional source grid to a global or regional destination grid. It assumes that the source and destination grids are on a sphere and that the coordinates given in the files are latitude and longitude values. The coordinates can either be in degrees or radians (this is indicated by the "units" attribute attached to the value). This file based regrid weight generation application is parallel. This application is used in the ESMF_RegridWeightGenCheck external demo, so that can serve as an example of its use.

This application requires the NetCDF libary to read the grid files and write out the weight files in NetCDF format. In addition, it also requires the LAPACK library to generate the patch regridding weights. To compile ESMF with the

NetCDF library and the LAPACK library, please refer to the Third Party Libraries Section in the ESMF User Guide for more information.

Internally this application uses the ESMF public API to generate the interpolation weights. If a the source or destination grid is logically rectangular, then `ESMF_GridCreate()` 27.3.8 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. The method `ESMF_MeshCreate()` 29.3.5 is used to create an ESMF_Mesh object, if the source or destination grid is a cubed sphere grid or an unstructured grid. When making this call, the flag `convert3D` is set to `TRUE` to convert the 2D coordinates into 3D Cartesian coordinates. `ESMF_FieldRegridStore()` is used to generate the weight table and indices table representing the interpolation matrix.

The regridding occurs in 3D to avoid problems with periodicity and with the pole singularity. This application supports four options for handling the pole region (i.e. the empty area above the top row of the source grid or below the bottom row of the source grid). The first option is to leave the pole region empty ("-p none"), in this case if a destination point lies above or below the top row of the source grid, it will fail to map, yielding an error (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 method (i.e. "-m conserve").

Masking is supported for grids generated from a SCRIP file where the grid_rank=2 (i.e. 2D logically rectangular grids). Masking is currently not supported for unstructured grids. If the variable "grid_imask" 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 a destination point can't be mapped because it falls outside the unmasked source grid, 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.

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

This application can also be used to generate patch interpolation weights. Patch interpolation is the ESMF version of a techique called "patch recovery" commonly used in finite element modeling [5] [10]. It typically results in better approximations to values and derivatives when compared to bilinear interpolation. Patch interpolation works by constructing multiple polynomial patches to represent the data in a source element. For 2D grids, these polynomials are currently 2nd degree 2D polynomials. The interpolated value at the destination point is the weighted average of the values of the patches at that point.

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

First-order conservative interpolation [19] is also available as a regridding method. This method will typically have a larger interpolation error than the previous two methods, but will do a much better job of preserving the value of the integral of data between the source and destination grid. In this method the value across each source cell is treated as a constant. The weights for a particular destination cell, are the area of intersection of each source cell with the destination cell divided by the area of the destination cell. Areas in this case are the great circle areas of the polygons which make up the cells (the cells around each center are defined by the corner coordinates in the grid file).

The interpolation weights generated by this application are output to a NetCDF file (specified by the "-w" or "–weight" keywords). The format of this file is the same as that generated by SCRIP. 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).

12.2 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 application is as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

-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.
```

Part III

Superstructure

13 Overview of Superstructure

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

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

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

13.1 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.

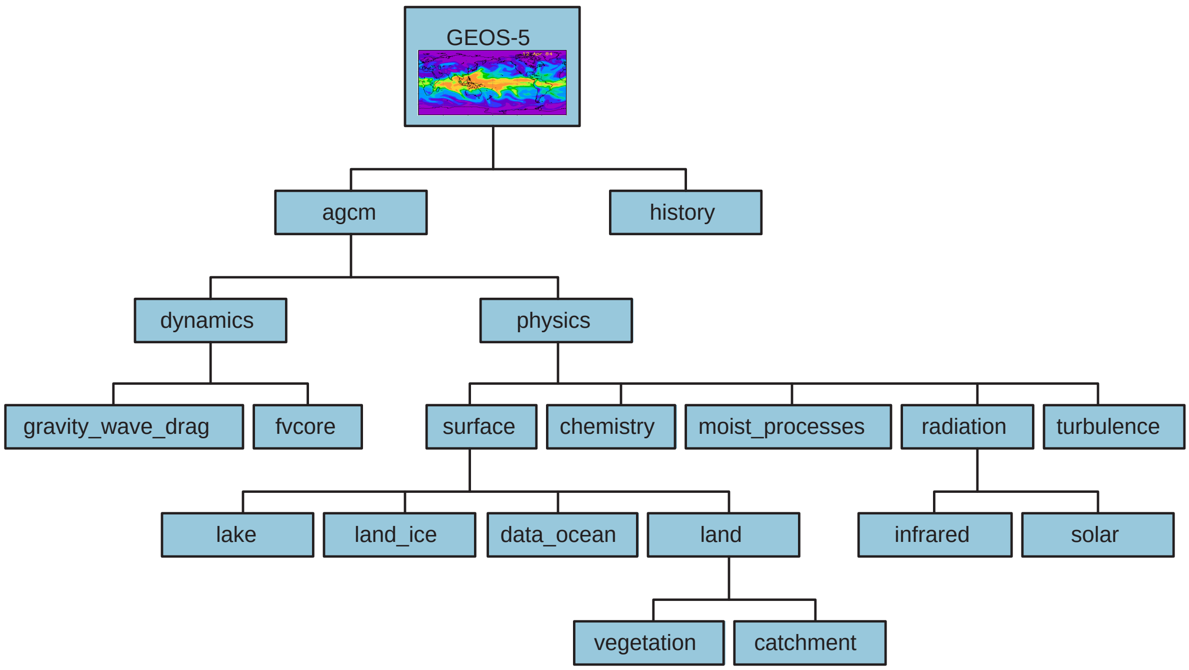
  The second part of an ESMF Component is user code, such as a model or data assimilation system. Users set entry points within their code so that it is callable by the framework. In practice, setting entry points means that within user code there are calls to ESMF methods that associate the name of a 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.

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.

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.



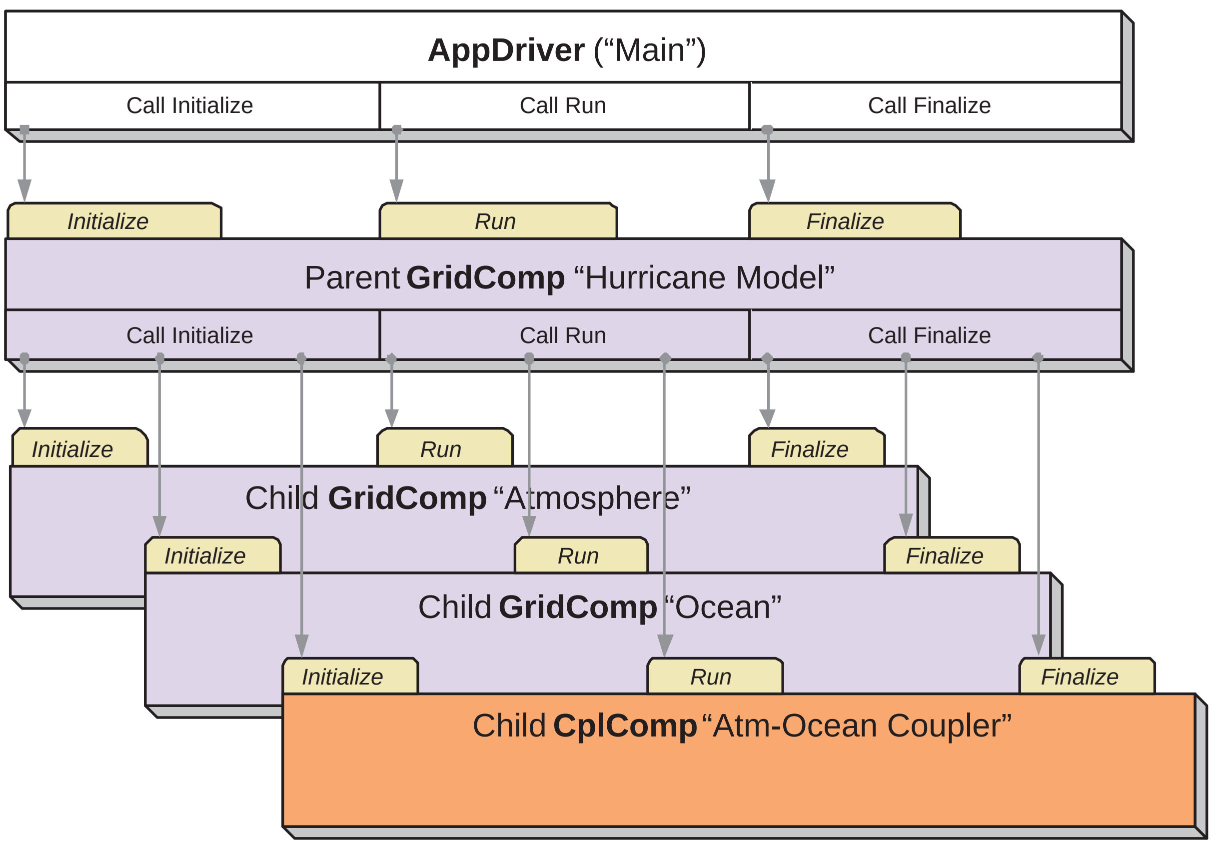
The application's "main" routine calls the parent Gridded Component's initialize, run, and finalize methods in order to execute the application. For each of these standard methods, the parent Gridded Component in turn calls the corresponding methods in the child Gridded Components and the Coupler Component. 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.

13.2 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.

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.

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.



13.3 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.

13.4 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.

13.5 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.

13.6 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.

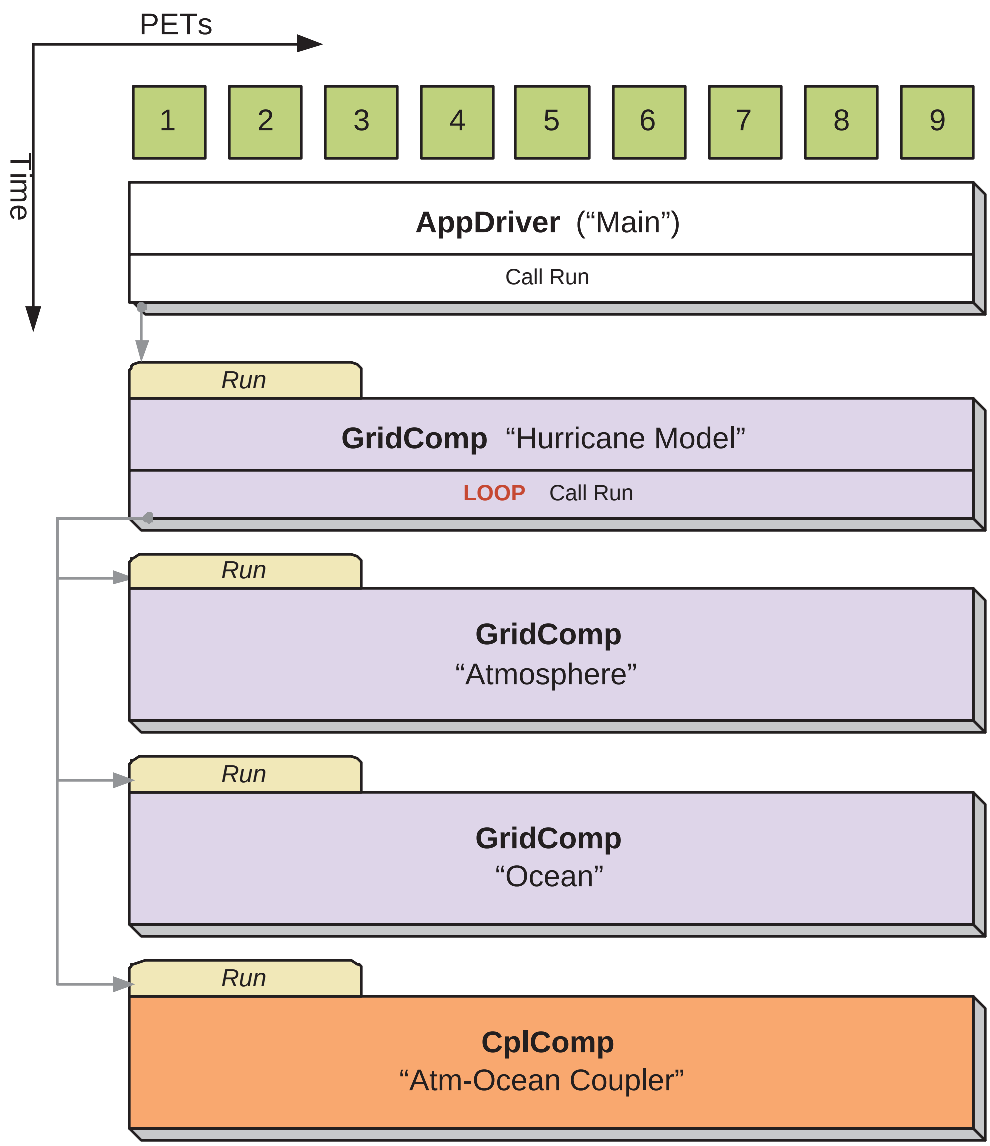


Figure 5: Schematic of the run method of a coupled application, with an "Atmosphere" and an "Ocean" Gridded Component running concurrently with an "Atm-Ocean Coupler." The top-level "Hurricane Model" Gridded Component contains the sequencing information and time advancement loop. The application driver,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

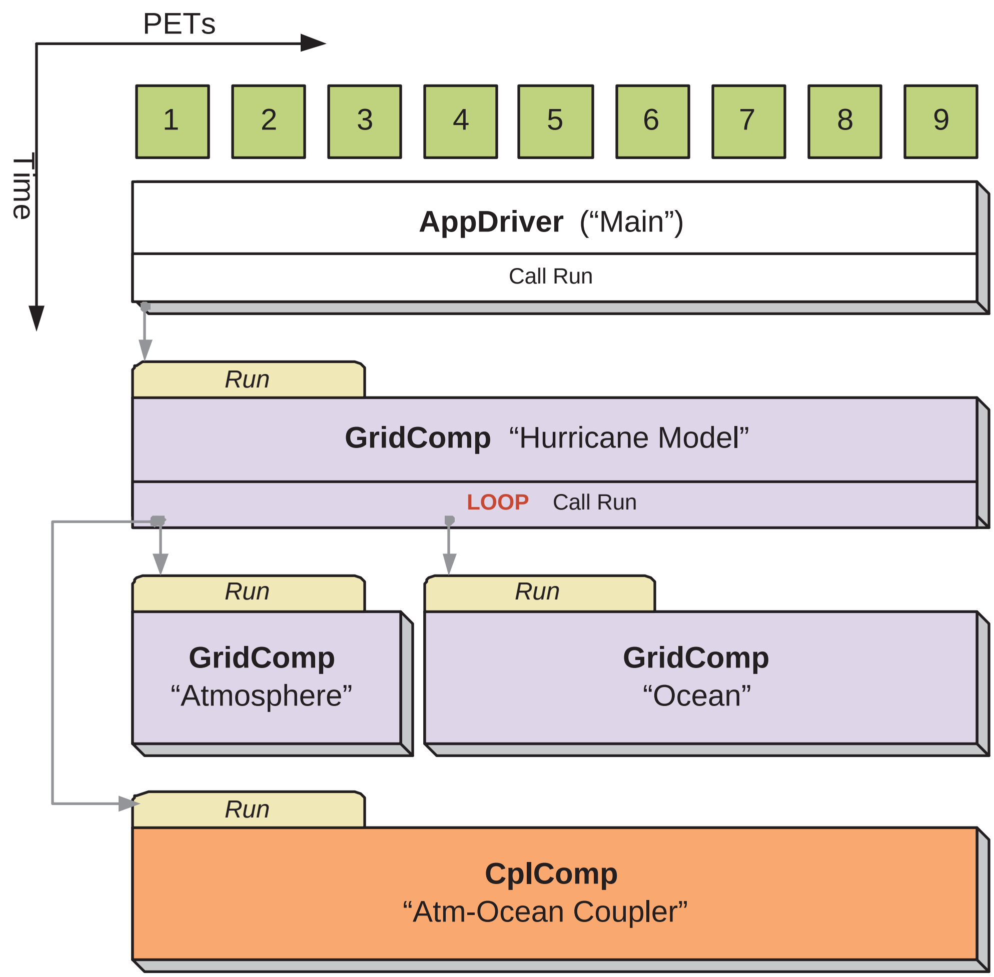
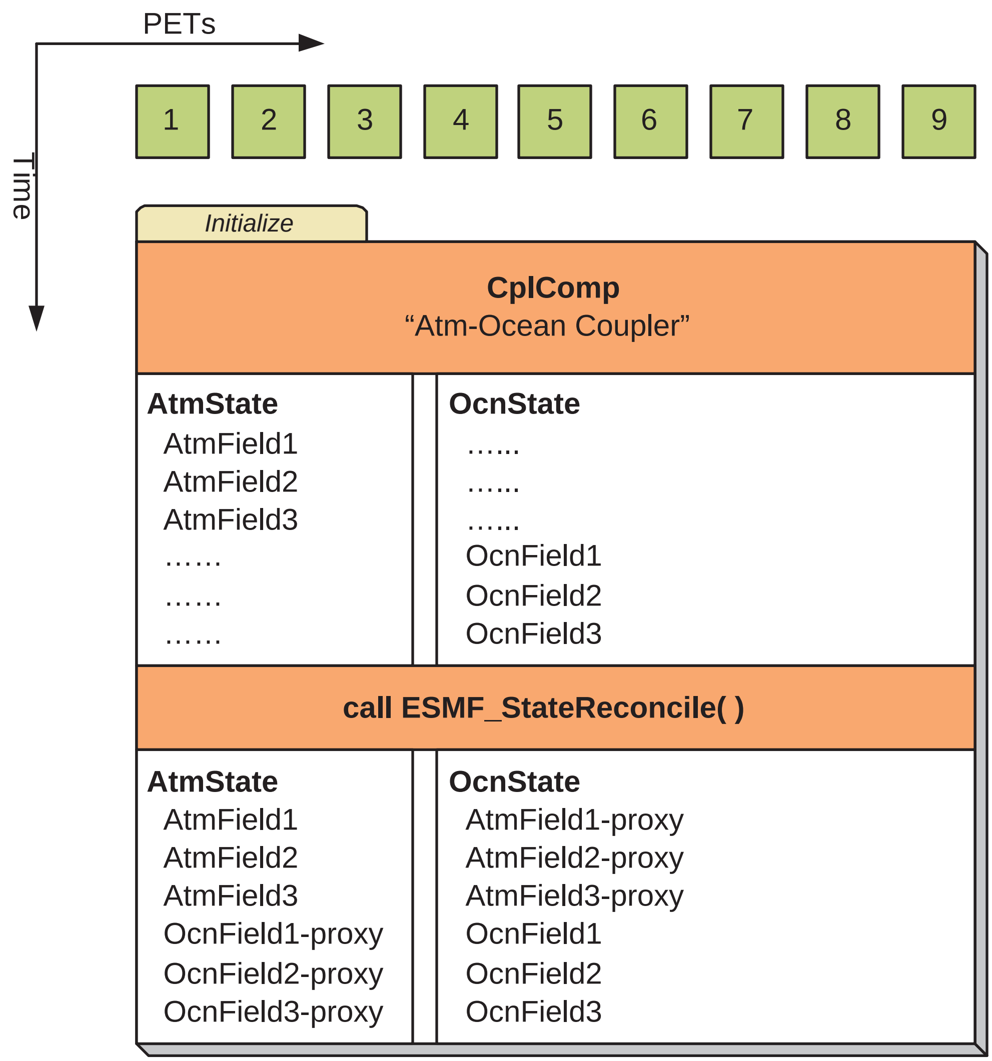
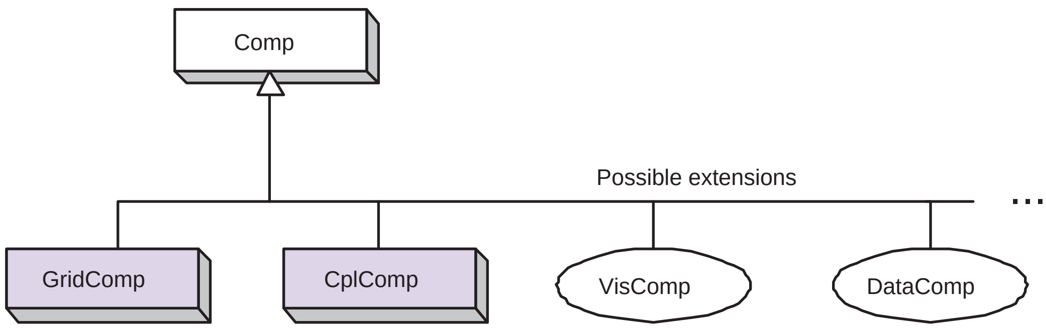


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



13.7 Object Model

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



14 Application Driver and Required ESMF Methods

14.1 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 Intialize, 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 communcations established. ESMF currently has some support for MPMD: different Components can run as separate executables, but the Coupler that transfers data between the Components must still run on the union of their PETs. This means that the Coupler Component must be linked into all of the executables.

14.2 Constants

14.2.1 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.

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.

14.3 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, 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
```

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

14.4.1 ESMF_Initialize - Initialize ESMF

INTERFACE:

```
    subroutine ESMF_Initialize(defaultConfigFileName, defaultCalKind, &
      defaultLogFileName, logkindflag, mpiCommunicator,  &
      ioUnitLBound, ioUnitUBound, vm, rc)
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
character(len=*),        intent(in),  optional :: defaultConfigFileName
type(ESMF_CalKind_Flag), intent(in),  optional :: defaultCalKind
character(len=*),        intent(in),  optional :: defaultLogFileName
type(ESMF_LogKind_Flag), intent(in),  optional :: logkindflag
integer,                 intent(in),  optional :: mpiCommunicator
integer,                 intent(in),  optional :: ioUnitLBound
integer,                 intent(in),  optional :: ioUnitUBound
type(ESMF_VM),           intent(out), optional :: vm
integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 IO 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).
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 46.2.1 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.
The arguments are:

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

[defaultCalKind] Sets the default calendar to be used by ESMF Time Manager. See section 37.2.1 for a list of valid options. If not specified, defaults to `ESMF_CALKIND_NOCALENDAR`.

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

[logkindflag] Sets the default Log Type to be used by ESMF Log Manager. See section 43.2.2 for a list of valid options. If not specified, defaults to `ESMF_LOGKIND_MULTI`.

[mpiCommunicator] MPI communicator defining the group of processes on which the ESMF application is running. 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.

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

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

14.4.2 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: Backward compatible starting with ESMF 5.2.0r.

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 14.2.1 for a complete list and description of valid flags.

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

14.4.3 User-code `SetServices` method

Many programs call some library routines. The library documentation must explain what the routine name is, what arguments are required and what are optional, and what the code does.
In contrast, all ESMF components must be written to *be called* by another part of the program; in effect, an ESMF component takes the place of a library. The interface is prescribed by the framework, and the component writer must provide specific subroutines which have standard argument lists and perform specific operations. 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 set services method. This subroutine must have an externally accessible name (be a public symbol), take a component as the first argument, and an integer return code as the second. 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 set services subroutine must provide is to specify the user-code entry points for the standard ESMF Component methods. To this end the user-written set services routine calls the
`ESMF_<Grid/Cpl>CompSetEntryPoint()` method to set each Component entry point.
See sections 15.2.1 and 16.2.1 for examples of how to write a user-code set services routine.

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

14.4.4 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 15.2.1 and 16.2.1 provide examples of how the entry points for Initialize, Run, and Finalize are set during the user-code set services 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 15.2.2, 15.2.3, 15.2.4, and 16.2.2, 16.2.3, 16.2.4 for examples of how to write entry points for the standard ESMF Component methods.

14.4.5 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 all of the parent PETs are provided to the child component.
Unless the optional `contextflag` argument is used during `ESMF_<Grid/Cpl>CompCreate()`, to indicate that the child component will execute in the same VM as the parent, the child component has the option to set certain aspects of how the provided resources are to be used when executing child component methods. The resources provided via the parent PETs are the associated processing elements (PEs) and virtual address spaces (VASs).
The optional user-written set vm routine is called from the parent through the `ESMF_<Grid/Cpl>CompSetVM()` library code, and 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 its set services routine can be called, and thus the optional `ESMF_<Grid/Cpl>CompSetVM()` call must be placed *before* `ESMF_<Grid/Cpl>CompSetServices()`.
If called by the parent, the user-code set vm routine has the option to specify how the PETs of the child component share the provided parent PEs. Further, PETs on the same single system image can be set to run multi-threaded, within a reduced number of VAS, allowing a component to leverage shared memory concepts.
Sections 15.2.5 and 16.2.5 provide examples for simple user-written set vm routines.

15 GridComp Class

15.1 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 17.1.

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.

15.2 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`.

15.2.1 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
```

15.2.2 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
```

15.2.3 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
```

15.2.4 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
```

15.2.5 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
```

15.2.6 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
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

  finalrc = ESMF_SUCCESS
!-------------------------------------------------------------------------

  call ESMF_Initialize(defaultlogfilename="InternalStateEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

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

  !  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
```

15.3 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.

15.4 Class API

15.4.1 ESMF_GridCompAssignment(=) - GridComp assignment

INTERFACE:

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

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

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.

15.4.2 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: Backward compatible starting with ESMF 5.2.0r.

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.

15.4.3 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: Backward compatible starting with ESMF 5.2.0r.

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.

15.4.4 ESMF_GridCompCreate - Create a GridComp

INTERFACE:

```
   recursive function ESMF_GridCompCreate(grid, &
     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_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: Backward compatible starting with ESMF 5.2.0r.

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] Default `ESMF_Grid` associated with this `gridcomp`. Note that it is perfectly ok to not pass a Grid in for this argument. This argument is simply a convenience for the user to allow them to associate a Grid with a component for their later use. The grid isn't actually used in the component code.

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

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

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

[petList] List of 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 9.9 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.

15.4.5 ESMF_GridCompDestroy - Release resources associated with a GridComp

INTERFACE:

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

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

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.

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

15.4.6 ESMF_GridCompFinalize - Call the GridComp's finalize routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompFinalize(gridcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, 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.

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

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

15.4.7 ESMF_GridCompGet - Get GridComp information

INTERFACE:

```
   subroutine ESMF_GridCompGet(gridcomp, gridIsPresent, grid, &
     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
     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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get information about an `ESMF_GridComp` object.
The arguments are:

gridcomp The `ESMF_GridComp` object being queried.

[gridIsPresent] `.true.` if `grid` was set in GridComp object, `.false.` otherwise.

[grid] Return the associated Grid. It is an error to query for the Grid if none is associated with the GridComp. If unsure, get `gridIsPresent` first to determine the status.

[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 9.9 for a complete list of valid flags.

[currentMethod] Return the current `ESMF_Method_Flag` of the `ESMF_GridComp` execution. See section 9.31 for a complete list of valid options.

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

[comptype] Return the Component type: `ESMF_COMPTYPE_GRID` or `ESMF_COMPTYPE_CPL`.

[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.

15.4.8 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: Backward compatible starting with ESMF 5.2.0r.

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: This method does not have an explicit Fortran interface. 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.

15.4.9 ESMF_GridCompInitialize - Call the GridComp's initialize routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompInitialize(gridcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

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

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

15.4.10 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: Backward compatible starting with ESMF 5.2.0r.

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.

15.4.11 ESMF_GridCompPrint - Print the contents of a GridComp

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: Backward compatible starting with ESMF 5.2.0r.

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.

15.4.12 ESMF_GridCompReadRestart - Call the GridComp's read restart routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompReadRestart(gridcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

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

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

15.4.13 ESMF_GridCompRun - Call the GridComp's run routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompRun(gridcomp, &
     importState, exportState,&
     clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

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

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

15.4.14 ESMF_GridCompSet - Set or reset information about the GridComp

INTERFACE:

```
   subroutine ESMF_GridCompSet(gridcomp, grid, 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_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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets or resets information about an `ESMF_GridComp`.
The arguments are:

gridcomp `ESMF_GridComp` to change.

[grid] Set the `ESMF_Grid` associated with the `ESMF_GridComp`.

[config] Set the configuration information for the `ESMF_GridComp` from this already created `ESMF_Config` object. If specified, takes priority over `configFile`.

[configFile] Set the configuration filename for this `ESMF_GridComp`. An `ESMF_Config` object will be created for this file and attached to the `ESMF_GridComp`. Superceeded by `config` if both are specified.

[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.

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

INTERFACE:

```
   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: Backward compatible starting with ESMF 5.2.0r.

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 9.31 for a complete list of valid method options.

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

[phase] The `phase` number for multi-phase methods. 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.

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.

15.4.16 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: Backward compatible starting with ESMF 5.2.0r.

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: This method does not have an explicit Fortran interface. 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.

15.4.17 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: Backward compatible starting with ESMF 5.2.0r.

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 Routine to be called.

[userRc] Return code set by userRoutine before returning.

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

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.
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.

15.4.18 ESMF_GridCompSetServices - Call user routine, located in shared object, to register GridComp methods

INTERFACE:

```
! Private name; call using ESMF_GridCompSetServices()
recursive subroutine ESMF_GridCompSetServicesShObj(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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Call into user provided routine which is responsible for setting Component's Initialize(), Run() and Finalize() services. The named `userRoutine` must exist in the shared object file specified in the `sharedObj` argument. 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.

[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.

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.

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.

15.4.19 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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for for setting Component's VM properties. The arguments are:

gridcomp Gridded Component.

userRoutine Routine to be called.

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

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

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. 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.

15.4.20 ESMF_GridCompSetVM - Call user routine, located in shared object, 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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for for setting Component's VM properties. The named `userRoutine` must exist in the shared object file specified in the `sharedObj` argument. All of the platform specific details about dynamic linking and loading apply.
The arguments are:

gridcomp Gridded Component.

userRoutine Routine to be called.

[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.

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.

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.

15.4.21 ESMF_GridCompSetVMMaxPEs - Associate PEs with PETs in GridComp VM

INTERFACE:

```
   subroutine ESMF_GridCompSetVMMaxPEs(gridcomp, &
     maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, 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(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.*

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

15.4.22 ESMF_GridCompSetVMMaxThreads - Set multi-threaded PETs in GridComp VM

INTERFACE:

```
  subroutine ESMF_GridCompSetVMMaxThreads(gridcomp, &
    maxPetCountPerVas, prefIntraProcess, prefIntraSsi, prefInterSsi, 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(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.*

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

15.4.23 ESMF_GridCompSetVMMinThreads - Set a reduced threading level in GridComp VM

INTERFACE:

```
   subroutine ESMF_GridCompSetVMMinThreads(gridcomp, &
     maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, 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(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.*

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

15.4.24 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: Backward compatible starting with ESMF 5.2.0r.

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.

15.4.25 ESMF_GridCompWait - Wait for a GridComp to return

INTERFACE:

```
   subroutine ESMF_GridCompWait(gridcomp, syncflag, 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(out), optional :: userRc
     integer,              intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

When executing asychronously, 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 9.44 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.

[userRc] Return code set by userRoutine before returning.

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

15.4.26 ESMF_GridCompWriteRestart - Call the GridComp's write restart routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompWriteRestart(gridcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

[userRc] Return code set by userRoutine before returning.

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

16 CplComp Class

16.1 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 15.1). 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.

16.2 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.

16.2.1 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.

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

```
! Example Coupler Component
module ESMF_CouplerEx

! 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
```

16.2.2 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
```

16.2.3 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
```

16.2.4 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
```

16.2.5 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
```

16.3 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.

16.4 Class API

16.4.1 ESMF_CplCompAssignment(=) - CplComp assignment

INTERFACE:

```
interface assignment(=)
cplcomp1 = cplcomp2
```

ARGUMENTS:

```
type(ESMF_CplComp) :: cplcomp1
type(ESMF_CplComp) :: cplcomp2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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.

16.4.2 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: Backward compatible starting with ESMF 5.2.0r.

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.

16.4.3 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: Backward compatible starting with ESMF 5.2.0r.

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.

16.4.4 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: Backward compatible starting with ESMF 5.2.0r.

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 configuration object from which the new component can read in namelist-type information to set parameters for this run. If both are specified, this object takes priority over configFile.

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

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

[petList] List of 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 9.9 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.

16.4.5 ESMF_CplCompDestroy - Release resources associated with a CplComp

INTERFACE:

```
   subroutine ESMF_CplCompDestroy(cplcomp, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

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.

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

16.4.6 ESMF_CplCompFinalize - Call the CplComp's finalize routine

INTERFACE:

```
   recursive subroutine ESMF_CplCompFinalize(cplcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

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

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

16.4.7 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: Backward compatible starting with ESMF 5.2.0r.

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 GridComp 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 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 CplComp. 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 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 9.9 for a complete list of valid flags.

[currentMethod] Return the current ESMF_Method_Flag of the ESMF_CplComp execution. See section 9.31 for a complete list of valid options.

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

[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 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.

16.4.8 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: Backward compatible starting with ESMF 5.2.0r.

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: This method does not have an explicit Fortran interface. 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.

16.4.9 ESMF_CplCompInitialize - Call the CplComp's initialize routine

INTERFACE:

```
   recursive subroutine ESMF_CplCompInitialize(cplcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

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

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

16.4.10 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: Backward compatible starting with ESMF 5.2.0r.

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.

16.4.11 ESMF_CplCompPrint - Print the contents of a CplComp

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: Backward compatible starting with ESMF 5.2.0r.

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.

16.4.12 ESMF_CplCompReadRestart – Call the CplComp's read restart routine

INTERFACE:

```
   recursive subroutine ESMF_CplCompReadRestart(cplcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

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

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

16.4.13 ESMF_CplCompRun - Call the CplComp's run routine

INTERFACE:

```
  recursive subroutine ESMF_CplCompRun(cplcomp, &
    importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
    integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

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

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

16.4.14 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: Backward compatible starting with ESMF 5.2.0r.

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] Set the configuration information for the ESMF_CplComp from this already created ESMF_Config object. If specified, takes priority over configFile.

[configFile] Set the configuration filename for this ESMF_CplComp. An ESMF_Config object will be created for this file and attached to the ESMF_CplComp. Superceeded by config if both are specified.

[clock] Set the private clock for this ESMF_CplComp.

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

16.4.15 ESMF_CplCompSetEntryPoint - Set user routine as entry point for standard Component method

INTERFACE:

```
   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: Backward compatible starting with ESMF 5.2.0r.

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 9.31 for a complete list of valid method options.

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

[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.

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.

16.4.16 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: Backward compatible starting with ESMF 5.2.0r.

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: This method does not have an explicit Fortran interface. 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.

16.4.17 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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

cplcomp Coupler Component.

userRoutine Routine to be called.

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

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

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.
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.

16.4.18 ESMF_CplCompSetServices - Call user routine, located in shared object, to register CplComp methods

INTERFACE:

```
! Private name; call using ESMF_CplCompSetServices()
recursive subroutine ESMF_CplCompSetServicesShObj(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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Call into user provided routine which is responsible for setting Component's Initialize(), Run() and Finalize() services. The named `userRoutine` must exist in the shared object file specified in the `sharedObj` argument. 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.

[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.

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.

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.

16.4.19 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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for for setting Component's VM properties. The arguments are:

cplcomp Coupler Component.

userRoutine Routine to be called.

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

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

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. 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.

16.4.20 ESMF_CplCompSetVM - Set CplComp VM properties in routine located in shared object

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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Optionally call into user provided `userRoutine` which is responsible for for setting Component's VM properties. The named `userRoutine` must exist in the shared object file specified in the `sharedObj` argument. All of the platform specific details about dynamic linking and loading apply.
The arguments are:

cplcomp Coupler Component.

userRoutine Routine to be called.

[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.

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.

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.

16.4.21 ESMF_CplCompSetVMMaxPEs - Associate PEs with PETs in CplComp VM

INTERFACE:

```
   subroutine ESMF_CplCompSetVMMaxPEs(cplcomp, &
     maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, 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(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.*

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

16.4.22 ESMF_CplCompSetVMMaxThreads - Set multi-threaded PETs in CplComp VM

INTERFACE:

```
   subroutine ESMF_CplCompSetVMMaxThreads(cplcomp, &
     maxPetCountPerVas, prefIntraProcess, prefIntraSsi, prefInterSsi, 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(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_GridCompSetVMMaxThreads() 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.*

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

16.4.23 ESMF_CplCompSetVMMinThreads - Set a reduced threading level in GridComp VM

INTERFACE:

```
   subroutine ESMF_CplCompSetVMMinThreads(cplcomp, &
     maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, 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(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_GridCompSetVMMinThreads()` 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.*

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

16.4.24 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: Backward compatible starting with ESMF 5.2.0r.

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.

16.4.25 ESMF_CplCompWait - Wait for a CplComp to return

INTERFACE:

```
   subroutine ESMF_CplCompWait(cplcomp, syncflag, 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(out), optional :: userRc
     integer,              intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

When executing asychronously, 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 9.44 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.

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

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

16.4.26 ESMF_CplCompWriteRestart – Call the CplComp's write restart routine

INTERFACE:

```
   recursive subroutine ESMF_CplCompWriteRestart(cplcomp, &
     importState, exportState, clock, syncflag, phase, 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(out),   optional :: userRc
     integer,              intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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 9.44 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 their each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accomodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For 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.

[userRc] Return code set by userRoutine before returning.

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

17 State Class

17.1 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.

17.2 Constants

17.2.1 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.

17.2.2 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`.

17.3 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. There is an option to turn off metadata reconciliation in the `ESMF_StateReconcile` call.

17.3.1 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`.

17.3.2 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)
```

17.3.3 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)
```

17.3.4 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)
```

17.3.5 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, valueList=neededFlag, rc=rc)
```

```
    if (rc == ESMF_SUCCESS .and. neededFlag(1)) 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
```

17.3.6 `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
!-----------------------------------------------------------------------------

    ! ESMF Framework module
    use ESMF
    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, comp_dummy, rc=rc)
call ESMF_GridCompSetServices(comp2, 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. There is the option of turning metadata reconciliation on or off with the optional parameter shown in the call below. The default behavior is for metadata reconciliation to be off. 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,  &
                             attreconflag=ESMF_ATTRECONCILE_OFF, rc=rc)

    print *, "State after calling StateReconcile()"
    call ESMF_StatePrint(state1, rc=rc)


end program ESMF_StateReconcileEx
```

17.3.7 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
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.

17.3.8 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`.

17.3.9 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.

17.4 Restrictions and Future Work

1. **No synchronization of object ids at object create time.** Object IDs are using during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time.

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

17.5 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 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 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:

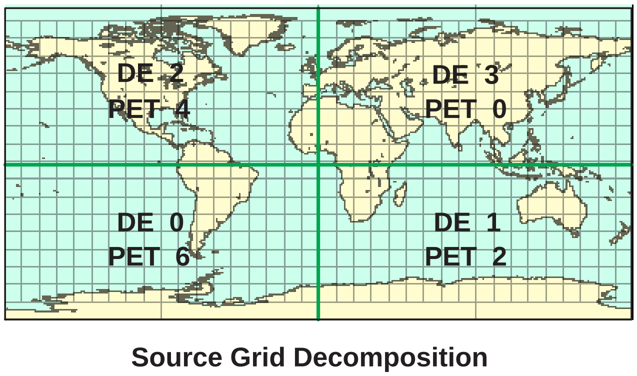


Figure 7: The mapping of PETs (processors) to DEs (data) in the source grid created by `user_model1.F90` in the FieldExcl system test.

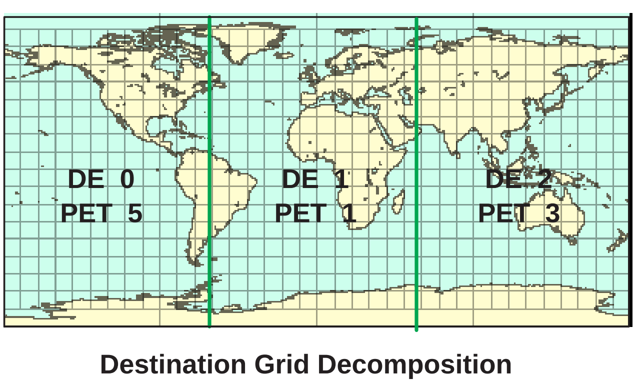


Figure 8: The mapping of PETs (processors) to DEs (data) in the destination grid created by `user_model2.F90` in the FieldExcl system test.

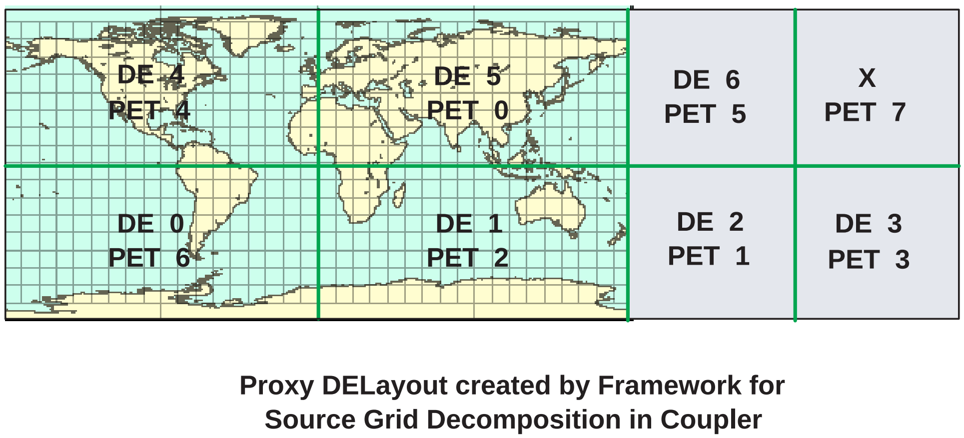


Figure 9: The mapping of PETs (processors) to DEs (data) in the source grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

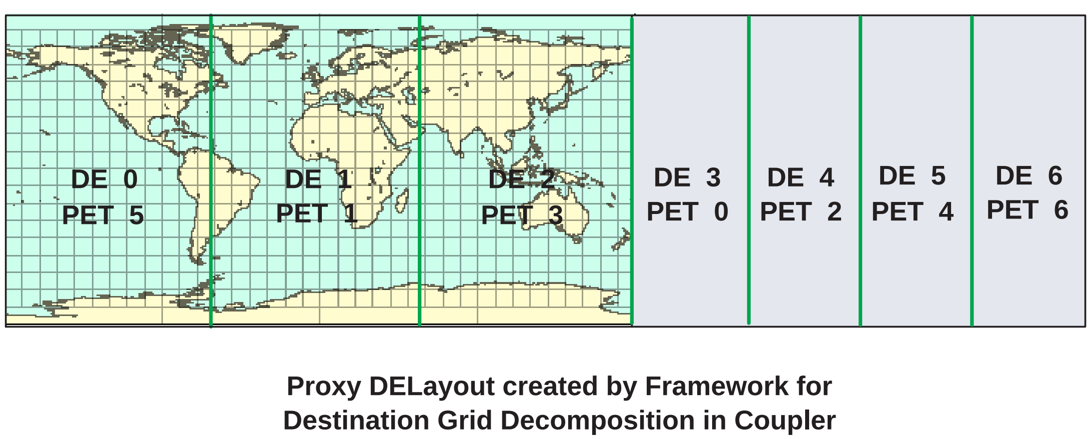
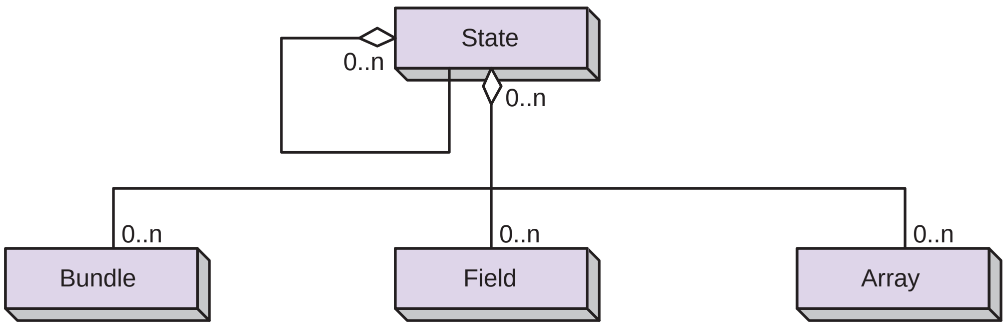


Figure 10: The mapping of PETs (processors) to DEs (data) in the destination grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

17.6 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.



17.7 Class API

17.7.1 ESMF_StateAssignment(=) - State assignment

INTERFACE:

```
interface assignment(=)
state1 = state2
```

ARGUMENTS:

```
type(ESMF_State) :: state1
type(ESMF_State) :: state2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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.

17.7.2 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: Backward compatible starting with ESMF 5.2.0r.

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.

17.7.3 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: Backward compatible starting with ESMF 5.2.0r.

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.

17.7.4 ESMF_StateAdd - Add a list of items to a State

INTERFACE:

```
   subroutine ESMF_StateAdd(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: Backward compatible starting with ESMF 5.2.0r.

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) :: stateList(:)

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.

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

17.7.5 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: Backward compatible starting with ESMF 5.2.0r.

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) :: stateList(:)

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.

17.7.6 ESMF_StateCreate - Create a new State

INTERFACE:

```
      function ESMF_StateCreate(stateintent, &
                   arrayList, arraybundleList, &
                   fieldList, fieldbundleList, &
                   nestedStateList, &
                   routehandleList, name, 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
      integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

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_Arrays.

[arraybundleList] A list (Fortran array) of ESMF_ArrayBundles.

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

[fieldbundleList] A list (Fortran array) of ESMF_FieldBundles.

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

[routehandleList] A list (Fortran array) of ESMF_RouteHandles.

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

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

17.7.7 ESMF_StateDestroy - Release resources for a State

INTERFACE:

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

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

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.

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

17.7.8 ESMF_StateGet - Get information about a State

INTERFACE:

```
      ! Private name; call using ESMF_StateGet()
      subroutine ESMF_StateGetInfo(state, &
            itemSearch, 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
      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: Backward compatible starting with ESMF 5.2.0r.

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.
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.

[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 17.2.1.

[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 17.2.2.

[name] Returns the name of this `ESMF_State`.

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

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.

17.7.9 ESMF_StateGet - Retrieve an item from a State

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: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns an <item> from an `ESMF_State` by name. If the `ESMF_State` contains the <item> directly, only `itemName` is required.
If the `state` contains nested `ESMF_State`s, 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.

17.7.10 ESMF_StateGet - Get information about an item in a State

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: Backward compatible starting with ESMF 5.2.0r.

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 existance 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 17.2.2. 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.

17.7.11 ESMF_StatePrint - Print the internal data for a State

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.

17.7.12 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 17.3.7 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:
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.

17.7.13 ESMF_StateReconcile – Reconcile State data across all PETs in a VM

INTERFACE:

```
subroutine ESMF_StateReconcile(state, vm, attreconflag, rc)
```

ARGUMENTS:

```
type(ESMF_State),             intent(inout)          :: state
type(ESMF_VM),                intent(in),  optional :: vm
type(ESMF_AttReconcileFlag), intent(in),  optional :: attreconflag
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 coupler's `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`. The option to reconcile the metadata associated with the objects contained in this `ESMF_State` also exists. The default behavior for this capability is to *not* reconcile metadata unless told otherwise.
The arguments are:

state `ESMF_State` to reconcile.

[vm] `ESMF_VM` for this `ESMF_Component`. By default, it set to the current vm.

[attreconflag] Flag to tell if Attribute reconciliation is to be done as well as data reconciliation. This flag is documented in section 9.4.

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

17.7.14 ESMF_StateRemove - Remove an item from a State

INTERFACE:

```
subroutine ESMF_StateRemove (state, itemName, rc)
```

ARGUMENTS:

```
    type(ESMF_State), intent(inout) :: state
    character(*), intent(in) :: itemName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Remove an existing reference to an item from a State.
The arguments are:

state The ESMF_State within which itemName will be replaced.

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 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.

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

17.7.15 ESMF_StateReplace - Replace a list of items within a State

INTERFACE:

```
subroutine ESMF_StateReplace(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: Backward compatible starting with ESMF 5.2.0r.

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) :: stateList(:)

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.

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

17.7.16 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: Backward compatible starting with ESMF 5.2.0r.

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.

17.7.17 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 17.3.7 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.

18 Attachable Methods

18.1 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.

18.2 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.

18.2.1 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

  call ESMF_MethodAdd(exportState, label="finalCalculation", &
    userRoutine=finalCalc, rc=rc)

  rc = 0
end subroutine !--------------------------------------------------------------
```

18.2.2 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

  ! access data objects in state and perform calculation

  print *, "dummy output from attached method "

  rc = 0
end subroutine !--------------------------------------------------------------
```

18.2.3 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

  call ESMF_MethodExecute(importState, label="finalCalculation", &
    userRc=userRc, rc=rc)

  rc = 0
end subroutine !--------------------------------------------------------------
```

18.3 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.

18.4 Class API

18.4.1 ESMF_MethodAdd - Attach user method to State

INTERFACE:

```
 ! Private name; call using ESMF_MethodAdd()
 subroutine ESMF_MethodStateAdd(state, label, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_State)                            :: state
character(len=*), intent(in)                :: label
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`.
The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

userRoutine The user-supplied subroutine to be associated with the `label`.

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

18.4.2 ESMF_MethodAdd - Attach user method, located in shared object, to State

INTERFACE:

```
  ! Private name; call using ESMF_MethodAdd()
  subroutine ESMF_MethodStateAddShObj(state, label, userRoutine, &
 sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_State)                            :: state
character(len=*), intent(in)                :: label
character(len=*), intent(in)                :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`.
The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

userRoutine Name of user-supplied subroutine to be associated with the `label`.

[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.

18.4.3 ESMF_MethodExecute - Execute user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
subroutine ESMF_MethodStateExecute(state, label, existflag, userRc, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
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` to attach to.

label Label of method.

[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.

18.4.4 ESMF_MethodRemove - Remove user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodStateRemove(state, label, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Remove attached method.
The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

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

18.4.5 ESMF_MethodAdd - Attach user method to GridComp

INTERFACE:

```
   ! Private name; call using ESMF_MethodAdd()
   subroutine ESMF_MethodGridCompAdd(gcomp, label, userRoutine, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                      :: gcomp
     character(len=*), intent(in)             :: label
     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`.
The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

userRoutine The user-supplied subroutine to be associated with the `label`.

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

18.4.6 ESMF_MethodAdd - Attach user method, located in shared object, to GridComp

INTERFACE:

```
   ! Private name; call using ESMF_MethodAdd()
   subroutine ESMF_MethodGridCompAddShObj(gcomp, label, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                      :: gcomp
     character(len=*), intent(in)             :: label
     character(len=*), intent(in)             :: userRoutine
     character(len=*), intent(in),  optional :: sharedObj
     integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`.
The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

userRoutine Name of user-supplied subroutine to be associated with the `label`.

[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.

18.4.7 ESMF_MethodAdd - Attach user method to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodCplCompAdd(cplcomp, label, userRoutine, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp)                     :: cplcomp
  character(len=*), intent(in)           :: label
  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`.
The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

userRoutine The user-supplied subroutine to be associated with the `label`.

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

18.4.8 ESMF_MethodAdd - Attach user method, located in shared object, to CplComp

INTERFACE:

```
 ! Private name; call using ESMF_MethodAdd()
 subroutine ESMF_MethodCplCompAddShObj(cplcomp, label, userRoutine, &
sharedObj, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Attach `userRoutine`.
The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

userRoutine Name of user-supplied subroutine to be associated with the `label`.

[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.

18.4.9 ESMF_MethodExecute - Execute user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
subroutine ESMF_MethodGridCompExecute(gcomp, label, existflag, userRc, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                        :: gcomp
character(len=*), intent(in)               :: label
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` to attach to.

label Label of method.

[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.

18.4.10 ESMF_MethodExecute - Execute user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
subroutine ESMF_MethodCplCompExecute(cplcomp, label, existflag, userRc, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                         :: cplcomp
character(len=*), intent(in)               :: label
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` to attach to.

label Label of method.

[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.

18.4.11 ESMF_MethodRemove - Remove user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodGridCompRemove(gcomp, label, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Remove attached method.
The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

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

18.4.12 ESMF_MethodRemove - Remove user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodCplCompRemove(cplcomp, label, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Remove attached method.
The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

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

19 Web Services

19.1 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.

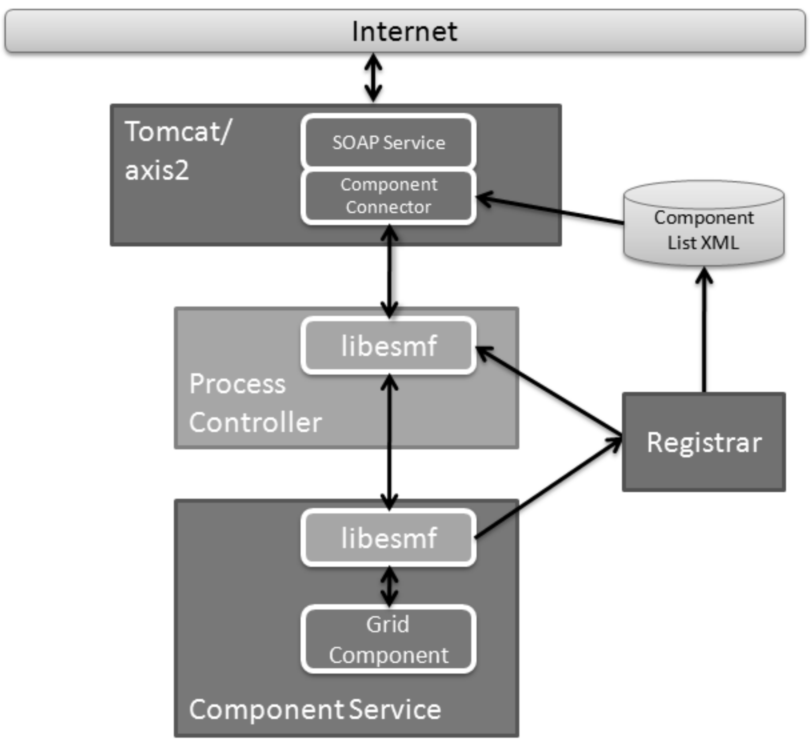
At the heart of this architecture is the Component Service; this is the application that does the model work. The ESMF Web Services part provides a way to make the model accessible via a network API (Application Programming Interface). ESMF provides the tools to turn a model component into a service 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.

19.1.1 Creating a Service around a Component

19.1.2 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

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.



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.

19.1.3 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

19.1.4 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.

19.1.5 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.

19.1.6 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.

19.2 Use and Examples

The following examples demonstrate how to use WebServices. ....

19.2.1 Making a Component available through WebServices

In this example a standard ESMF Component is made available through the WebServices 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
  ! ESMF Framework module
  use ESMF
  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
```

The port number specifies the id of the port on the local machine on which a listening socket will be created. 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
```

19.3 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.

19.4 Class API

19.4.1 ESMF_WebServicesLoop

INTERFACE:

```
subroutine ESMF_WebServicesLoop(comp, portNum, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                                  :: comp
integer,                 intent(inout), optional :: portNum
integer,                 intent(out),   optional :: rc
```

DESCRIPTION:

Encapsulates all of the functionality necessary to setup a component as a component service. If this is 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. If this is 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_GridComp` object that represents the Grid Component for which routine is run.

[portNum] Number of the port on which the component service is listening.

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

Part IV

Infrastructure: Fields and Grids

20 Overview of Infrastructure Data Handling

The ESMF infrastructure data classes are part of the framework's hierarchy of structures for handling Earth system model data and metadata on parallel platforms. The hierarchy is in complexity; the simplest data class in the infrastructure represents a distributed data array and the most complex data class represents a bundle of physical fields that are discretized on the same grid. Data class methods are called both from user-written code and from other classes internal to the framework.

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

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

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

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

Key Features

Hierarchy of data structures designed specifically for the Earth system domain and high performance, parallel computing.

Multi-use ESMF structures simplify user code overall.

Data objects support incremental construction and deferred allocation.

Native Fortran arrays can be associated with or retrieved from ESMF data objects, for ease of adoption, convenience, and performance.

20.1 Infrastructure Data Classes

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

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

  Data elements in Arrays are partitioned into categories defined by the role the data element plays in distributed halo operations. Haloing - sometimes called ghosting - is the practice of copying portions of array data to multiple memory locations to ensure that data dependencies can be satisfied quickly when performing a calculation. ESMF Arrays contain an **exclusive** domain, which contains data elements updated exclusively and definitively by a given DE; a **computational** domain, which contains all data elements with values that are updated by the DE in computations; and a **total** domain, which includes both the computational domain and data elements from other DEs which may be read but are not updated in computations.

- **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 re-ordering of data in memory.

20.2 Design and Implementation Notes

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

21 FieldBundle Class

21.1 Description

A FieldBundle functions mainly as a convenient container for storing similar Fields. It represents “bundles” of Fields that are discretized on the same Grid, Mesh, LocStream, or XGrid and distributed in the same manner. The FieldBundle is an important data structure because it can be added to a State, which is used for sending and receiving data between Components.
In the common case where FieldBundle is built on top of a Grid, Fields within a FieldBundle may be located at different locations relative to the vertices of their common Grid. The Fields in a FieldBundle may be of different dimensions, as long as the Grid dimensions that are distributed are the same. For example, a surface Field on a distributed lat/lon Grid and a 3D Field with an added vertical dimension on the same distributed lat/lon Grid can be included in the same FieldBundle.
FieldBundles can be created and destroyed, can have Attributes added or retrieved, and can have Fields added, removed, replaced, or retrieved. Methods include queries that return information about the FieldBundle itself and about the Fields that it contains. The Fortran data pointer of a Field within a FieldBundle can be obtained by first retrieving the Field with a call to `ESMF_FieldBundleGet()`, and then using `ESMF_FieldGet()` to get the data.
In the future FieldBundles will serve as a mechanism for performance optimization. ESMF will take advantage of the similarities of the Fields within a FieldBundle to optimize collective communication, IO, and regridding. See Section 21.3 for a description of features that are scheduled for future work.

21.2 Use and Examples

Examples of creating, destroying and accessing FieldBundles and their constituent Fields are provided in this section, along with some notes on FieldBundle methods.

21.2.1 Create a FieldBundle

After creating multiple Fields by calling `ESMF_FieldCreate()`, a FieldBundle can be created by passing a list of the Fields into the method `ESMF_FieldBundleCreate()`. The FieldBundle will contain references to the Fields. An empty FieldBundle can also be created and Fields added one at a time or in groups.

21.2.2 Access FieldBundle data

To access data in a FieldBundle the user can provide a Field name and retrieve the Field’s Fortran data pointer. Alternatively, the user can retrieve the data in the form of an ESMF Field and use the Field-level interfaces.

21.2.3 Destroy a FieldBundle

The user must call `ESMF_FieldBundleDestroy()` before deleting any of the Fields it contains. Because Fields can be shared by multiple FieldBundles and States, they are not deleted by this call.
DESCRIPTION:

See the following code fragments for examples of how to create new FieldBundles.

```
! Example program showing various ways to create a FieldBundle object.

program ESMF_FieldBundleCreateEx

! ESMF Framework module
use ESMF

implicit none

! Local variables
```

```
     integer :: i, rc, fieldcount
     type(ESMF_Grid) :: grid
     type(ESMF_ArraySpec) :: arrayspec
     character (len = ESMF_MAXSTR) :: bname1, fname1, fname2
     type(ESMF_Field) :: field(10), returnedfield1, returnedfield2
     type(ESMF_Field) :: simplefield
     type(ESMF_FieldBundle) :: bundle1, bundle2, bundle3

!-------------------------------------------------------------------------
!   !  Create several Fields and add them to a new FieldBundle.

    grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/100,200/), &
                                 regDecomp=(/2,2/), name="atmgrid", rc=rc)


    call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R8, rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    field(1) = ESMF_FieldCreate(grid, arrayspec, &
                                staggerloc=ESMF_STAGGERLOC_CENTER, &
                                name="pressure", rc=rc)


    field(2) = ESMF_FieldCreate(grid, arrayspec, &
                                staggerloc=ESMF_STAGGERLOC_CENTER, &
                                name="temperature", rc=rc)


    field(3) = ESMF_FieldCreate(grid, arrayspec, &
                                staggerloc=ESMF_STAGGERLOC_CENTER, &
                                name="heat flux", rc=rc)


    bundle1 = ESMF_FieldBundleCreate(fieldList=field(1:3), &
name="atmosphere data", rc=rc)

    print *, "FieldBundle example 1 returned"


!-------------------------------------------------------------------------
!   !  Create an empty FieldBundle and then add a single field to it.


    simplefield = ESMF_FieldCreate(grid, arrayspec, &
                  staggerloc=ESMF_STAGGERLOC_CENTER, name="rh", rc=rc)


    bundle2 = ESMF_FieldBundleCreate(name="time step 1", rc=rc)


    call ESMF_FieldBundleAdd(bundle2, (/simplefield/), rc=rc)


    call ESMF_FieldBundleGet(bundle2, fieldCount=fieldcount, rc=rc)

    print *, "FieldBundle example 2 returned, fieldcount =", fieldcount
```

```
!------------------------------------------------------------------------
!   !  Create an empty FieldBundle and then add multiple fields to it.


    bundle3 = ESMF_FieldBundleCreate(name="southern hemisphere", rc=rc)


    call ESMF_FieldBundleAdd(bundle3, field(1:3), rc=rc)


    call ESMF_FieldBundleGet(bundle3, fieldCount=fieldcount, rc=rc)

    print *, "FieldBundle example 3 returned, fieldcount =", fieldcount

!------------------------------------------------------------------------
!   !  Get a Field back from a FieldBundle, first by name and then by index.
!   !  Also get the FieldBundle name.

    call ESMF_FieldBundleGet(bundle1, "pressure", field=returnedfield1, rc=rc)


    call ESMF_FieldGet(returnedfield1, name=fname1, rc=rc)


    call ESMF_FieldBundleGet(bundle1, 2, returnedfield2, rc=rc)


    call ESMF_FieldGet(returnedfield2, name=fname2, rc=rc)


    call ESMF_FieldBundleGet(bundle1, name=bname1, rc=rc)

    print *, "FieldBundle example 4 returned, field names = ", &
                   trim(fname1), ", ", trim(fname2)
    print *, "FieldBundle name = ", trim(bname1)

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

     call ESMF_FieldBundleDestroy(bundle1, rc=rc)


     call ESMF_FieldBundleDestroy(bundle2, rc=rc)


     call ESMF_FieldBundleDestroy(bundle3, rc=rc)


     do i=1, 3
         call ESMF_FieldDestroy(field(i),rc=rc)


     enddo

     call ESMF_FieldDestroy(simplefield, rc=rc)


     end program ESMF_FieldBundleCreateEx
```

21.2.4 Redistribute data from a source FieldBundle to a destination FieldBundle

A user can use ESMF_FieldBundleRedist interface to redistribute data from source FieldBundle to destination FieldBundle. This interface is overloaded by type and kind; In the version of ESMF_FieldBundleRedist without factor argument, a default value of factor 1 is used.
In this example, we first create two FieldBundles, a source FieldBundle and a destination FieldBundle. Then we use ESMF_FieldBundleRedist to redistribute data from source FieldBundle to destination FieldBundle.

```
! retrieve VM and its context info such as PET number
call ESMF_VMGetCurrent(vm, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_VMGet(vm, localPet=lpe, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create distgrid and grid for field and fieldbundle creation
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
    regDecomp=(/2,2/), rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 3, ESMF_TYPEKIND_I4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create src and dst FieldBundles pair
srcFieldBundle = ESMF_FieldBundleCreate(rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

dstFieldBundle = ESMF_FieldBundleCreate(rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create src and dst Fields and add the Fields into FieldBundles
do i = 1, 3
    srcField(i) = ESMF_FieldCreate(grid, arrayspec, &
        ungriddedLBound=(/1/), ungriddedUBound=(/4/), &
        totalLWidth=(/1,1/), totalUWidth=(/1,2/), &
        rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_FieldGet(srcField(i), localDe=0, farrayPtr=srcfptr, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    srcfptr = lpe

    call ESMF_FieldBundleAdd(srcFieldBundle, (/srcField(i)/), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    dstField(i) = ESMF_FieldCreate(grid, arrayspec, &
        ungriddedLBound=(/1/), ungriddedUBound=(/4/), &
        totalLWidth=(/1,1/), totalUWidth=(/1,2/), &
        rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_FieldGet(dstField(i), localDe=0, farrayPtr=dstfptr, rc=rc)
```

```
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    dstfptr = 0

    call ESMF_FieldBundleAdd(dstFieldBundle, (/dstField(i)/), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! perform redist
call ESMF_FieldBundleRedistStore(srcFieldBundle, dstFieldBundle, &
     routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldBundleRedist(srcFieldBundle, dstFieldBundle, &
     routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! verify redist
do l = 1, 3
    call ESMF_FieldGet(dstField(l), localDe=0, farrayPtr=fptr, &
      exclusiveLBound=exLB, exclusiveUBound=exUB, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Verify that the redistributed data in dstField is correct.
    ! Before the redist op, the dst Field contains all 0.
    ! The redist op reset the values to the PE value,
    ! verify this is the case.
    ! MUST use exclusive bounds because Redist operates
    ! within excl. region.
    do k = exLB(3), exUB(3)
        do j = exLB(2), exUB(2)
            do i = exLB(1), exUB(1)
               if(fptr(i,j,k) .ne. lpe) finalrc = ESMF_FAILURE
            enddo
        enddo
    enddo
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! release route handle
call ESMF_FieldRedistRelease(routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldBundleDestroy(srcFieldBundle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_FieldBundleDestroy(dstFieldBundle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
do i = 1, 3
    call ESMF_FieldDestroy(srcField(i), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    call ESMF_FieldDestroy(dstField(i), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo
call ESMF_GridDestroy(grid, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

21.2.5 Perform sparse matrix multiplication from a source FieldBundle to a destination FieldBundle

A user can use `ESMF_FieldBundleSMM` interface to perform SMM from source FieldBundle to destination FieldBundle. This interface is overloaded by type and kind;
In this example, we first create two FieldBundles, a source FieldBundle and a destination FieldBundle. Then we use `ESMF_FieldBundleSMM` to perform sparse matrix multiplication from source FieldBundle to destination FieldBundle.
The operation performed in this example is better illustrated in section 22.3.36.
Section 24.2.17 provides a detailed discussion of the sparse matrix mulitiplication operation implemented in ESMF.

```
call ESMF_VMGetCurrent(vm, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_VMGet(vm, localPet=lpe, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create distgrid and grid
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
    regDecomp=(/4/), &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

grid = ESMF_GridCreate(distgrid=distgrid, &
    gridEdgeLWidth=(/0/), gridEdgeUWidth=(/0/), &
    name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_I4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create field bundles and fields
srcFieldBundle = ESMF_FieldBundleCreate(rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

dstFieldBundle = ESMF_FieldBundleCreate(rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

do i = 1, 3
    srcField(i) = ESMF_FieldCreate(grid, arrayspec, &
        totalLWidth=(/1/), totalUWidth=(/2/), &
        rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_FieldGet(srcField(i), localDe=0, farrayPtr=srcfptr, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    srcfptr = 1

    call ESMF_FieldBundleAdd(srcFieldBundle, (/srcField(i)/), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

```
    dstField(i) = ESMF_FieldCreate(grid, arrayspec, &
        totalLWidth=(/1/), totalUWidth=(/2/), &
        rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_FieldGet(dstField(i), localDe=0, farrayPtr=dstfptr, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    dstfptr = 0

    call ESMF_FieldBundleAdd(dstFieldBundle, (/dstField(i)/), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! initialize factorList and factorIndexList
allocate(factorList(4))
allocate(factorIndexList(2,4))
factorList = (/1,2,3,4/)
factorIndexList(1,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)
factorIndexList(2,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)
call ESMF_FieldBundleSMMStore(srcFieldBundle, dstFieldBundle, &
    routehandle, factorList, factorIndexList, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! perform smm
call ESMF_FieldBundleSMM(srcFieldBundle, dstFieldBundle, routehandle, &
      rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! verify smm
do l = 1, 3
    call ESMF_FieldGet(dstField(l), localDe=0, farrayPtr=fptr, &
        exclusiveLBound=exlb, exclusiveUBound=exub, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Verify that the smm data in dstField(l) is correct.
    ! Before the smm op, the dst Field contains all 0.
    ! The smm op reset the values to the index value, verify
    ! this is the case.
    !write(*, '(9I3)') l, lpe, fptr
    do i = exlb(1), exub(1)
        if(fptr(i) .ne. i) finalrc = ESMF_FAILURE
    enddo
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! release SMM route handle
call ESMF_FieldBundleSMMRelease(routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! release all acquired resources
call ESMF_FieldBundleDestroy(srcFieldBundle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_FieldBundleDestroy(dstFieldBundle, rc=rc)
```

```
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
do l = 1, 3
    call ESMF_FieldDestroy(srcField(l), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    call ESMF_FieldDestroy(dstField(l), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo
call ESMF_GridDestroy(grid, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_DistGridDestroy(distgrid, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
deallocate(factorList, factorIndexList)
```

21.2.6 Perform FieldBundle halo update

`ESMF_FieldBundleHalo` interface can be used to perform halo update of all the Fields contained in the `ESMF_FieldBundle`.

In this example, we will set up a FieldBundle for a 2D viscous and compressible flow problem. We will illustrate the FieldBundle halo update operation but we will not solve the non-linear PDEs here. The emphasis here is to demonstrate how to set up halo regions, how a numerical scheme updates the exclusive regions, and how halo update communicates data in the halo regions. Here are the governing equations:

$u_t + uu_x + vu_y + \frac{1}{\rho}p_x = 0$ (conservation of momentum in x-direction)

$v_t + uv_x + vv_y + \frac{1}{\rho}p_y = 0$ (conservation of momentum in y-direction)

$\rho_t + \rho u_x + \rho v_y = 0$ (conservation of mass)

$\frac{\rho}{\rho^\gamma} + u(\frac{p}{\rho^\gamma})_x + v(\frac{p}{\rho^\gamma})_y = 0$ (conservation of energy)

The four unknowns are pressure p, density ρ, velocity (u, v). The grids are set up using Arakawa D stagger (p on corner, ρ at center, u and v on edges). p, ρ, u, and v are bounded by necessary boundary conditions and initial conditions.

Section 24.2.14 provides a detailed discussion of the halo operation implemented in ESMF.

```
! create distgrid and grid according to the following decomposition
! and stagger pattern, r is density.
!
! p--------u-------+p+-------u--------p
! !                 |                 |
! !                 |                 |
! !                 |                 |
! v        r        v        r        v
! !      PET 0      |      PET 1      |
! !                 |                 |
! !                 |                 |
! p--------u-------+p+-------u--------p
! !                 |                 |
! !                 |                 |
! !                 |                 |
! v        r        v        r        v
! !      PET 2      |      PET 3      |
! !                 |                 |
! !                 |                 |
! p--------u-------+p+-------u--------p
!
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/256,256/), &
```

```
    regDecomp=(/2,2/), &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create field bundles and fields
fieldBundle = ESMF_FieldBundleCreate(rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! set up exclusive/total region for the fields
!
! halo: L/U, nDim, nField, nPet
! halo configuration for pressure, and similarly for density, u, and v
halo(1,1,1,1) = 0
halo(2,1,1,1) = 0
halo(1,2,1,1) = 0
halo(2,2,1,1) = 0
halo(1,1,1,2) = 1   ! halo in x direction on left hand side of pet 1
halo(2,1,1,2) = 0
halo(1,2,1,2) = 0
halo(2,2,1,2) = 0
halo(1,1,1,3) = 0
halo(2,1,1,3) = 1   ! halo in y direction on upper side of pet 2
halo(1,2,1,3) = 0
halo(2,2,1,3) = 0
halo(1,1,1,4) = 1   ! halo in x direction on left hand side of pet 3
halo(2,1,1,4) = 1   ! halo in y direction on upper side of pet 3
halo(1,2,1,4) = 0
halo(2,2,1,4) = 0


! names and staggers of the 4 unknown fields
names(1) = "pressure"
names(2) = "density"
names(3) = "u"
names(4) = "v"
staggers(1) = ESMF_STAGGERLOC_CORNER
staggers(2) = ESMF_STAGGERLOC_CENTER
staggers(3) = ESMF_STAGGERLOC_EDGE2
staggers(4) = ESMF_STAGGERLOC_EDGE1

! create a FieldBundle
lpe = lpe + 1
do i = 1, 4
    field(i) = ESMF_FieldCreate(grid, arrayspec, &
            totalLWidth=(/halo(1,1,i,lpe), halo(1,2,i,lpe)/), &
            totalUWidth=(/halo(2,1,i,lpe), halo(2,2,i,lpe)/), &
            staggerloc=staggers(i), name=names(i), &
            rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    call ESMF_FieldBundleAdd(fieldBundle, (/field(i)/), rc=rc)
```

```
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! compute the routehandle
call ESMF_FieldBundleHaloStore(fieldBundle, routehandle=routehandle, &
                               rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

do iter = 1, 10
    do i = 1, 4
        call ESMF_FieldGet(field(i), farrayPtr=fptr, &
            exclusiveLBound=excllb, exclusiveUBound=exclub, rc=rc)
        if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
        sizes = exclub - excllb
        ! fill the total region with 0.
        fptr = 0.
        ! only update the exclusive region on local PET
        do j = excllb(1), exclub(1)
          do k = excllb(2), exclub(2)
            fptr(j,k) = iter * cos(2.*PI*j/sizes(1))*sin(2.*PI*k/sizes(2))
          enddo
        enddo
    enddo
    ! call halo execution to update the data in the halo region,
    ! it can be verified that the halo regions change from 0.
    ! to non zero values.
    call ESMF_FieldBundleHalo(fieldbundle, routehandle=routehandle, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo
! release halo route handle
call ESMF_FieldBundleHaloRelease(routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

21.3 Restrictions and Future Work

1. **No mathematical operators.** The FieldBundle class does not support differential or other mathematical operators. We do not anticipate providing this functionality in the near future.
2. **Limited validation and print options.** We are planning to increase the number of validity checks available for FieldBundles as soon as possible. We also will be working on print options.
3. **Packed data not supported.** One of the options that we are currently working on for FieldBundles is packing. Packing means that the data from all the Fields that comprise the FieldBundle are manipulated collectively. This operation can be done without destroying the original Field data. Packing is being designed to facilitate optimized regridding, data communication, and IO operations. This will reduce the latency overhead of the communication.
4. **Interleaving Fields within a FieldBundle.** Data locality is important for performance on some computing platforms. An interleave option will allow the user to create a packed FieldBundle in which Fields are either concatenated in memory or in which Field elements are interleaved.

21.4 Design and Implementation Notes

1. **Fields in a FieldBundle reference the same Grid, Mesh, LocStream, or XGrid.** In order to reduce memory requirements and ensure consistency, the Fields within a FieldBundle all reference the same Grid, Mesh,

LocStream, or XGrid object. This restriction may be relaxed in the future.

21.5 Class API: Basic FieldBundle Methods

21.5.1 ESMF_FieldBundleAssignment(=) - FieldBundle assignment

INTERFACE:

```
interface assignment(=)
fieldbundle1 = fieldbundle2
```

ARGUMENTS:

```
type(ESMF_FieldBundle) :: fieldbundle1
type(ESMF_FieldBundle) :: fieldbundle2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign fieldbundle1 as an alias to the same ESMF fieldbundle object in memory as fieldbundle2. If fieldbundle2 is invalid, then fieldbundle1 will be equally invalid after the assignment.
The arguments are:

fieldbundle1 The `ESMF_FieldBundle` object on the left hand side of the assignment.

fieldbundle2 The `ESMF_FieldBundle` object on the right hand side of the assignment.

21.5.2 ESMF_FieldBundleOperator(==) - FieldBundle equality operator

INTERFACE:

```
interface operator(==)
  if (fieldbundle1 == fieldbundle2) then ... endif
            OR
  result = (fieldbundle1 == fieldbundle2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: fieldbundle1
type(ESMF_FieldBundle), intent(in) :: fieldbundle2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether fieldbundle1 and fieldbundle2 are valid aliases to the same ESMF fieldbundle object in memory. For a more general comparison of two ESMF FieldBundles, going beyond the simple alias test, the ESMF_FieldBundleMatch() function (not yet implemented) must be used.
The arguments are:

fieldbundle1 The `ESMF_FieldBundle` object on the left hand side of the equality operation.

fieldbundle2 The `ESMF_FieldBundle` object on the right hand side of the equality operation.

21.5.3 ESMF_FieldBundleOperator(/=) - FieldBundle not equal operator

INTERFACE:

```
   interface operator(/=)
     if (fieldbundle1 /= fieldbundle2) then ... endif
               OR
     result = (fieldbundle1 /= fieldbundle2)
```

RETURN VALUE:

```
     logical :: result
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(in) :: fieldbundle1
     type(ESMF_FieldBundle), intent(in) :: fieldbundle2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether fieldbundle1 and fieldbundle2 are *not* valid aliases to the same ESMF fieldbundle object in memory. For a more general comparison of two ESMF FieldBundles, going beyond the simple alias test, the ESMF_FieldBundleMatch() function (not yet implemented) must be used.
The arguments are:

fieldbundle1 The `ESMF_FieldBundle` object on the left hand side of the non-equality operation.

fieldbundle2 The `ESMF_FieldBundle` object on the right hand side of the non-equality operation.

21.5.4 ESMF_FieldBundleAdd - Add Fields to a FieldBundle

INTERFACE:

```
     ! Private name; call using ESMF_FieldBundleAdd()
     subroutine ESMF_FieldBundleAddList(fieldbundle, fieldList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout)         :: fieldbundle
     type(ESMF_Field),       intent(in)            :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Add Field(s) to a FieldBundle. It is an error if `fieldList` contains Fields that match by name Fields already contained in `fieldbundle` when multiflag is set to `.false.` and relaxedflag is set to `.false.`.

fieldbundle `ESMF_FieldBundle` to be added to.

fieldList List of `ESMF_Field` objects to be added.

[multiflag] A setting of `.true.` allows multiple items with the same name to be added to `ESMF_FieldBundle`. For `.false.` added items must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add" under `multiflag=.false.` mode, where it is *not* an error if `fieldList` contains items with names that are also found in `ESMF_FieldBundle`. The `ESMF_FieldBundle` 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.

21.5.5 ESMF_FieldBundleAddReplace - Conditionally add or replace Fields in a FieldBundle

INTERFACE:

```
     subroutine ESMF_FieldBundleAddReplace(fieldbundle, fieldList, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout)         :: fieldbundle
     type(ESMF_Field),       intent(in)            :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Fields in `fieldList` that do not match any Fields by name in `fieldbundle` are added to the FieldBundle. Fields in `fieldList` that match any Fields by name in `fieldbundle` replace those Fields.

fieldbundle `ESMF_FieldBundle` to be manipulated.

fieldList List of `ESMF_Field` objects to be added or used as replacement.

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

21.5.6 ESMF_FieldBundleCreate - Create a FieldBundle from a list of Fields

INTERFACE:

```
   function ESMF_FieldBundleCreate(fieldList, &
       multiflag, relaxedflag, name, rc)
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Field), intent(in),  optional :: fieldList(:)
     logical,          intent(in),  optional :: multiflag
     logical,          intent(in),  optional :: relaxedflag
     character (len=*),intent(in),  optional :: name
     integer,          intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_FieldBundle) :: ESMF_FieldBundleCreate
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_FieldBundle` object from a list of existing Fields.
The creation of a FieldBundle leaves the bundled Fields unchanged, they remain valid individual objects. a FieldBundle is a light weight container of Field references. The actual data remains in place, there are no data movements or duplications associated with the creation of an FieldBundle.

[fieldList] List of `ESMF_Field` objects to be bundled.

[multiflag] A setting of `.true.` allows multiple items with the same name to be added to `fieldbundle`. For `.false.` added items must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add" under `multiflag=.false.` mode, where it is *not* an error if `fieldList` contains items with names that are also found in `fieldbundle`. The `fieldbundle` is left unchanged for these items. For `.false.` this is treated as an error condition. The default setting is `.false.`.

[name] Name of the created `ESMF_FieldBundle`. A default name is generated if not specified.

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

21.5.7 ESMF_FieldBundleDestroy - Release resources associated with a FieldBundle

INTERFACE:

```
   subroutine ESMF_FieldBundleDestroy(fieldbundle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroy an `ESMF_FieldBundle` object. The member Fields are not touched by this operation and remain valid objects that need to be destroyed individually if necessary.
The arguments are:

fieldbundle `ESMF_FieldBundle` object to be destroyed.

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

21.5.8 ESMF_FieldBundleGet - Get information about a Field by name

INTERFACE:

```
! Private name; call using ESMF_FieldBundleGet()
subroutine ESMF_FieldBundleGetItem(fieldbundle, fieldName, &
  field, fieldCount, isPresent, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in)            :: fieldbundle
    character(len=*),       intent(in)            :: fieldName
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Field),       intent(out), optional :: field
    integer,                intent(out), optional :: fieldCount
    logical,                intent(out), optional :: isPresent
    integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get information about items that match `fieldName` in FieldBundle.

fieldbundle `ESMF_FieldBundle` to be queried.

fieldName Specified name.

[field] Upon return holds the requested field item. It is an error if this argument was specified and there is not exactly one field item in `ESMF_FieldBundle` that matches `fieldName`.

[fieldCount] Number of Fields with `fieldName` in `ESMF_FieldBundle`.

[isPresent] Upon return indicates whether field(s) with `fieldName` exist in `ESMF_FieldBundle`.

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

21.5.9 ESMF_FieldBundleGet - Get a list of Fields by name

INTERFACE:

```
! Private name; call using ESMF_FieldBundleGet()
subroutine ESMF_FieldBundleGetList(fieldbundle, fieldName, fieldList, &
  rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in)            :: fieldbundle
    character(len=*),       intent(in)            :: fieldName
    type(ESMF_Field),       intent(out)           :: fieldList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get the list of Fields from fieldbundle that match fieldName.

fieldbundle `ESMF_FieldBundle` to be queried.

fieldName Specified name.

[fieldList] List of Fields in `ESMF_FieldBundle` that match `fieldName`. The argument must be allocated to be at least of size `fieldCount` returned for this `fieldName`.

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

21.5.10 ESMF_FieldBundleGet - Get information of the FieldBundle

INTERFACE:

```
! Private name; call using ESMF_FieldBundleGet()
subroutine ESMF_FieldBundleGetListAll(fieldbundle, &
  geomtype, grid, locstream, mesh, xgrid, &
  fieldCount, fieldList, fieldNameList, name, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle),  intent(in)            :: fieldbundle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_GeomType_Flag),intent(out), optional :: geomtype
    type(ESMF_Grid),         intent(out), optional :: grid
    type(ESMF_LocStream),    intent(out), optional :: locstream
    type(ESMF_Mesh),         intent(out), optional :: mesh
    type(ESMF_XGrid),        intent(out), optional :: xgrid
    integer,                 intent(out), optional :: fieldCount
    type(ESMF_Field),        intent(out), optional :: fieldList(:)
    character(len=*),        intent(out), optional :: fieldNameList(:)
    character(len=*),        intent(out), optional :: name
    integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get the list of all Fields and field names bundled in a FieldBundle.

fieldbundle `ESMF_FieldBundle` to be queried.

[geomtype] Flag that indicates what type of geometry this FieldBundle object holds. Can be `ESMF_GEOMTYPE_GRID`, `ESMF_GEOMTYPE_MESH`, `ESMF_GEOMTYPE_LOCSTREAM`, `ESMF_GEOMTYPE_XGRID`

[grid] The Grid object that this FieldBundle object holds.

[locstream] The LocStream object that this FieldBundle object holds.

[mesh] The Mesh object that this FieldBundle object holds.

[xgrid] The XGrid object that this FieldBundle object holds.

[fieldCount] Upon return holds the number of Fields bundled in the fieldbundle.

[fieldList] Upon return holds a list of Fields bundled in `ESMF_FieldBundle`. The argument must be allocated to be at least of size `fieldCount`.

[fieldNameList] Upon return holds a list of the names of the field bundled in `ESMF_FieldBundle`. The argument must be allocated to be at least of size `fieldCount`.

[name] Name of the fieldbundle object.

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

21.5.11 ESMF_FieldBundleHalo - Execute a FieldBundle halo operation

INTERFACE:

```
   subroutine ESMF_FieldBundleHalo(fieldbundle, routehandle, &
     checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(inout)           :: fieldbundle
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),   optional  :: checkflag
         integer,                intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed FieldBundle halo operation for the Fields in FieldBundle. See `ESMF_FieldBundleStore()` on how to compute routehandle.

fieldbundle `ESMF_FieldBundle` with source data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

21.5.12 ESMF_FieldBundleHaloRelease - Release resources associated with a FieldBundle halo operation

INTERFACE:

```
   subroutine ESMF_FieldBundleHaloRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a FieldBundle halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

21.5.13 ESMF_FieldBundleHaloStore - Precompute a FieldBundle halo operation

INTERFACE:

```
     subroutine ESMF_FieldBundleHaloStore(fieldbundle, routehandle, &
       rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout)           :: fieldbundle
     type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a FieldBundle halo operation over the data in `fieldbundle`. By definition, all elements in the total Field regions that lie outside the exclusive regions will be considered potential destination elements for halo. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleHalo()` on any FieldBundle that is weakly congruent and typekind conform to `fieldbundle`. Congruency for FieldBundles is given by the congruency of its constituents. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
This call is *collective* across the current VM.

fieldbundle `ESMF_FieldBundle` containing data to be haloed. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

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

21.5.14 ESMF_FieldBundlePrint - Print FieldBundle internals

INTERFACE:

```
   subroutine ESMF_FieldBundlePrint(fieldbundle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Print internal information of the specified `fieldbundle` object.

The arguments are:

fieldbundle `ESMF_FieldBundle` object.

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

21.5.15 ESMF_FieldBundleRead - Read Fields to a FieldBundle from file(s)

INTERFACE:

```
   subroutine ESMF_FieldBundleRead(fieldbundle, file, &
     singleFile, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout)          :: fieldbundle
     character(*),           intent(in)             :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional  :: singleFile
     type(ESMF_IOFmtFlag),   intent(in),  optional  :: iofmt
     integer,                intent(out), optional  :: rc
```

DESCRIPTION:

Read field data to a FieldBundle object from file(s). For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

fieldbundle An `ESMF_FieldBundle` object.

file The name of the file from which fieldbundle data is read.

[singleFile] A logical flag, the default is .true., i.e., all Fields in the bundle are stored in one single file. If .false., each field is stored in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to `ESMF_IOFMT_NETCDF`.

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

21.5.16 ESMF_FieldBundleRedist - Execute a FieldBundle redistribution

INTERFACE:

```
   subroutine ESMF_FieldBundleRedist(srcFieldBundle, dstFieldBundle, &
     routehandle, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(in),    optional  :: srcFieldBundle
         type(ESMF_FieldBundle), intent(inout), optional  :: dstFieldBundle
         type(ESMF_RouteHandle), intent(inout)            :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),    optional  :: checkflag
         integer,                intent(out),   optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed FieldBundle redistribution from `srcFieldBundle` to `dstFieldBundle`. Both `srcFieldBundle` and `dstFieldBundle` must be weakly congruent and typekind conform with the respective FieldBundles used during `ESMF_FieldBundleRedistStore()`. Congruent FieldBundles possess matching DistGrids and the shape of the local array tiles matches between the FieldBundles for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
It is erroneous to specify the identical FieldBundle object for `srcFieldBundle` and `dstFieldBundle` arguments.
See `ESMF_FieldBundleRedistStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.
For examples and associated documentations using this method see Section 21.2.4.

[srcFieldBundle] `ESMF_FieldBundle` with source data.

[dstFieldBundle] `ESMF_FieldBundle` with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

21.5.17 ESMF_FieldBundleRedistRelease - Release resources associated with a FieldBundle redistribution

INTERFACE:

```
   subroutine ESMF_FieldBundleRedistRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a FieldBundle redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

21.5.18 ESMF_FieldBundleRedistStore - Precompute a FieldBundle redistribution with local factor argument

INTERFACE:

```
   ! Private name; call using ESMF_FieldBundleRedistStore()
   subroutine ESMF_FieldBundleRedistStore<type><kind>(srcFieldBundle, &
     dstFieldBundle, routehandle, factor, &
     srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(in)             :: srcFieldBundle
     type(ESMF_FieldBundle), intent(inout)          :: dstFieldBundle
     type(ESMF_RouteHandle), intent(inout)          :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)           :: factor
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),   optional :: srcToDstTransposeMap(:)
     integer,                intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a FieldBundle redistribution operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify a `factor` argument must use the <type><kind> overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1.
Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*. Redistribution corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.
Source and destination FieldBundles may be of different <type><kind>. Further source and destination FieldBundles may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleRedist()` on any pair of FieldBundles that are congruent and typekind conform with the srcFieldBundle, dstFieldBundle pair. Congruent FieldBundles possess matching DistGrids and the shape of the local array tiles matches between the FieldBundles for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.
For examples and associated documentations using this method see Section 21.2.4.
The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multiply source data.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcFieldBundle`. Each entry maps the corresponding `srcFieldBundle` dimension against the specified `dstFieldBundle` dimension. Mixing of distributed and undistributed dimensions is supported.

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

21.5.19 ESMF_FieldBundleRedistStore - Precompute a FieldBundle redistribution with local factor argument

INTERFACE:

```
  ! Private name; call using ESMF_FieldBundleRedistStore()
  subroutine ESMF_FieldBundleRedistStoreNF(srcFieldBundle, dstFieldBundle, &
         routehandle, factor, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(in)            :: srcFieldBundle
     type(ESMF_FieldBundle), intent(inout)         :: dstFieldBundle
     type(ESMF_RouteHandle), intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),   optional :: srcToDstTransposeMap(:)
     integer,                intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a FieldBundle redistribution operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*. Redistribution corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.
Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleRedist()` on any pair of Fields that are congruent and typekind conform with the srcFieldBundle, dstFieldBundle pair. Congruent Fields

possess matching DistGrids and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.
For examples and associated documentations using this method see Section 21.2.4.
The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcFieldBundle`. Each entry maps the corresponding `srcFieldBundle` dimension against the specified `dstFieldBundle` dimension. Mixing of distributed and undistributed dimensions is supported.

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

21.5.20 ESMF_FieldBundleRegrid - Execute a FieldBundle regrid operation

INTERFACE:

```
   subroutine ESMF_FieldBundleRegrid(srcFieldBundle, dstFieldBundle, &
          routehandle, zeroregion, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(in),    optional  :: srcFieldBundle
         type(ESMF_FieldBundle), intent(inout), optional  :: dstFieldBundle
         type(ESMF_RouteHandle), intent(inout)            :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         type(ESMF_Region_Flag), intent(in),    optional  :: zeroregion
         logical,                intent(in),    optional  :: checkflag
         integer,                intent(out),   optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed FieldBundle regrid from `srcFieldBundle` to `dstFieldBundle`. Both `srcFieldBundle` and `dstFieldBundle` must be congruent and typekind conform with the respective FieldBundles used during `ESMF_FieldBundleRegridStore()`. Congruent FieldBundles possess matching DistGrids and the shape of the local array tiles matches between the FieldBundles for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
It is erroneous to specify the identical FieldBundle object for `srcFieldBundle` and `dstFieldBundle` arguments.
See `ESMF_FieldBundleRegridStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.

[srcFieldBundle] `ESMF_FieldBundle` with source data.

[dstFieldBundle] `ESMF_FieldBundle` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in `dstFieldBundle` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstFieldBundle` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination FieldBundle that will be updated by the sparse matrix multiplication. See section 9.36 for a complete list of valid settings.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

21.5.21 ESMF_FieldBundleRegridRelease - Release resources associated with a FieldBundle regrid operation

INTERFACE:

```
   subroutine ESMF_FieldBundleRegridRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a FieldBundle regrid operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

21.5.22 ESMF_FieldBundleRegridStore - Precompute a FieldBundle regrid operation

INTERFACE:

```
     subroutine ESMF_FieldBundleRegridStore(srcFieldBundle, dstFieldBundle, &
                                            srcMaskValues, dstMaskValues, &
                                            regridmethod, polemethod, &
                                            regridPoleNPnts,  &
                                            unmappedaction, routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle),        intent(in)             :: srcFieldBundle
type(ESMF_FieldBundle),        intent(inout)          :: dstFieldBundle
integer(ESMF_KIND_I4),         intent(in),   optional :: srcMaskValues(:)
integer(ESMF_KIND_I4),         intent(in),   optional :: dstMaskValues(:)
type(ESMF_RegridMethod_Flag),  intent(in),   optional :: regridmethod
type(ESMF_PoleMethod_Flag),    intent(in),   optional :: polemethod
integer,                       intent(in),   optional :: regridPoleNPnts
type(ESMF_UnmappedAction_Flag),intent(in),   optional :: unmappedaction
type(ESMF_RouteHandle),        intent(inout),optional :: routehandle
integer,                       intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a FieldBundle regrid operation over the data in `srcFieldBundle` and `dstFieldBundle` pair.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleRegrid()` on any FieldBundle pairs that are weakly congruent and typekind conform to the FieldBundle pair used here. Congruency for FieldBundles is given by the congruency of its constituents. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Note `ESMF_FieldBundleRegridStore()` assumes the coordinates used in the Grids upon which the FieldBundles are built are in degrees.
This call is *collective* across the current VM.

srcFieldbundle Source `ESMF_FieldBundle` containing data to be regridded.

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

dstFieldbundle Destination `ESMF_FieldBundle`.

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

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

[regridmethod] The type of interpolation. Please see Section 22.2.3 for a list of valid options. If not specified, defaults to `ESMF_REGRIDMETHOD_BILINEAR`.

[polemethod] Which type of artificial pole to construct on the source Grid for regridding. Please see Section 22.2.2 for a list of valid options. If not specified, defaults to `ESMF_POLEMETHOD_ALLAVG`.

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

[routehandle] Handle to the precomputed Route.

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

21.5.23 ESMF_FieldBundleRemove - Remove Fields from FieldBundle

INTERFACE:

```
     subroutine ESMF_FieldBundleRemove(fieldbundle, fieldNameList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout)          :: fieldbundle
     character(len=*),       intent(in)             :: fieldNameList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Remove field(s) by name from FieldBundle. In the relaxed setting it is *not* an error if `fieldNameList` contains names that are not found in `fieldbundle`.

fieldbundle `ESMF_FieldBundle` from which to remove items.

fieldNameList List of items to remove.

[multiflag] A setting of `.true.` allows multiple Fields with the same name to be removed from `fieldbundle`. For `.false.`, items to be removed must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "remove" where it is *not* an error if `fieldNameList` contains item names that are not found in `fieldbundle`. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "remove" also covers the case where there are multiple items in `fieldbundle` that match a single entry in `fieldNameList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

21.5.24 ESMF_FieldBundleReplace - Replace Fields in FieldBundle

INTERFACE:

```
     subroutine ESMF_FieldBundleReplace(fieldbundle, fieldList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout)          :: fieldbundle
     type(ESMF_Field),       intent(in)             :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Replace field(s) by name in FieldBundle. In the relaxed setting it is not an error if `fieldList` contains Fields that do not match by name any item in `fieldbundle`. These Fields are simply ignored in this case.

fieldbundle `ESMF_FieldBundle` in which to replace items.

fieldList List of items to replace.

[multiflag] A setting of `.true.` allows multiple items with the same name to be replaced in `fieldbundle`. For `.false.`, items to be replaced must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "replace" where it is *not* an error if `fieldList` contains items with names that are not found in `fieldbundle`. These items in `fieldList` are ignored in the relaxed mode. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "replace" also covers the case where there are multiple items in `fieldbundle` that match a single entry by name in `fieldList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

21.5.25 ESMF_FieldBundleSMM - Execute a FieldBundle sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_FieldBundleSMM(srcFieldBundle, dstFieldBundle, &
         routehandle, zeroregion, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(in),    optional  :: srcFieldBundle
         type(ESMF_FieldBundle), intent(inout), optional  :: dstFieldBundle
         type(ESMF_RouteHandle), intent(inout)            :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         type(ESMF_Region_Flag), intent(in),    optional  :: zeroregion
         logical,                intent(in),    optional  :: checkflag
         integer,                intent(out),   optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed FieldBundle sparse matrix multiplication from `srcFieldBundle` to `dstFieldBundle`. Both `srcFieldBundle` and `dstFieldBundle` must be congruent and typekind conform with the respective FieldBundles used during `ESMF_FieldBundleSMMStore()`. Congruent FieldBundles possess matching DistGrids and the shape of the local array tiles matches between the FieldBundles for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
It is erroneous to specify the identical FieldBundle object for `srcFieldBundle` and `dstFieldBundle` arguments.
See `ESMF_FieldBundleSMMStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.
For examples and associated documentations using this method see Section 21.2.5.

[srcFieldBundle] `ESMF_FieldBundle` with source data.

[dstFieldBundle] `ESMF_FieldBundle` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in `dstFieldBundle` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstFieldBundle` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to

`ESMF_REGION_SELECT` will only zero out those elements in the destination FieldBundle that will be updated by the sparse matrix multiplication. See section 9.36 for a complete list of valid settings.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

21.5.26 ESMF_FieldBundleSMMRelease - Release resources associated with a FieldBundle sparse matrix multiplication

INTERFACE:

```
subroutine ESMF_FieldBundleSMMRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a FieldBundle sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

21.5.27 ESMF_FieldBundleSMMStore - Precompute a FieldBundle sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_FieldBundleSMMStore()
subroutine ESMF_FieldBundleSMMStore<type><kind>(srcFieldBundle, &
  dstFieldBundle,  routehandle, factorList, factorIndexList, &
  rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle),   intent(in)            :: srcFieldBundle
     type(ESMF_FieldBundle),   intent(inout)         :: dstFieldBundle
     type(ESMF_RouteHandle),   intent(inout)         :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)            :: factorList(:)
     integer,                  intent(in),           :: factorIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a FieldBundle sparse matrix multiplication operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.
Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleSMM()` on any pair of FieldBundles that are congruent and typekind conform with the srcFieldBundle, dstFieldBundle pair. Congruent FieldBundles possess matching DistGrids and the shape of the local array tiles matches between the FieldBundles for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.
For examples and associated documentations using this method see Section 21.2.5.
The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the `srcFieldBundle` while `factorIndexList(2,:)` specifies the sequence index of the destination element in `dstFieldBundle`. For this format to be a valid option source and destination FieldBundles must have matching number of tensor elements (the product of the sizes of all Field tensor dimensions). Under this condition an identiy matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the

`factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the `srcFieldBundle`. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the `dstFieldBundle`.

See section 24.2.17 for details on the definition of *sequence indices* and *tensor sequence indices*.

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

21.5.28 ESMF_FieldBundleSMMStore - Precompute a FieldBundle sparse matrix multiplication without local factors

INTERFACE:

```
! Private name; call using ESMF_FieldBundleSMMStore()
subroutine ESMF_FieldBundleSMMStoreNF(srcFieldBundle, dstFieldBundle, &
       routehandle, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle),   intent(in)            :: srcFieldBundle
    type(ESMF_FieldBundle),   intent(inout)         :: dstFieldBundle
    type(ESMF_RouteHandle),   intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a FieldBundle sparse matrix multiplication operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.
Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleSMM()` on any pair of FieldBundles that are congruent and typekind conform with the srcFieldBundle, dstFieldBundle pair. Congruent FieldBundles possess matching DistGrids and the shape of the local array tiles matches between the FieldBundles for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.
For examples and associated documentations using this method see Section 21.2.5.
The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

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

21.5.29 ESMF_FieldBundleValidate - Validate fieldbundle internals

INTERFACE:

```
   subroutine ESMF_FieldBundleValidate(fieldbundle, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(in)              :: fieldbundle
     integer,                intent(out),  optional  :: rc
```

DESCRIPTION:

Validates that the `fieldbundle` is internally consistent. The method returns an error code if problems are found.
The arguments are:

fieldbundle Specified `ESMF_FieldBundle` object.

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

21.5.30 ESMF_FieldBundleWrite - Write the Fields into a file

INTERFACE:

```
   subroutine ESMF_FieldBundleWrite(fieldbundle, file, &
     singleFile, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(in)              :: fieldbundle
     character(*),           intent(in)              :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional  :: singleFile
     integer,                intent(in),   optional  :: timeslice
     type(ESMF_IOFmtFlag),   intent(in),   optional  :: iofmt
     integer,                intent(out),  optional  :: rc
```

DESCRIPTION:

Write the Fields into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

fieldbundle An `ESMF_FieldBundle` object.

file The name of the output file to which field bundle data is written.

[singleFile] A logical flag, the default is .true., i.e., all fields in the bundle are written in one single file. If .false., each field will be written in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[timeslice] Some IO formats (e.g. NetCDF) support the output of data in form of time slices. The `timeslice` argument provides access to this capability. Usage of this feature requires that the first slice is written with a positive `timeslice` value, and that subsequent slices are written with a `timeslice` argument that increments by one each time. By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file.

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to `ESMF_IOFMT_NETCDF`.

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

22 Field Class

22.1 Description

An ESMF Field represents a physical field, such as temperature. The motivation for including Fields in ESMF is that bundles of Fields are the entities that are normally exchanged when coupling Components.
The ESMF Field class contains distributed and discretized field data, a reference to its associated grid, and metadata. The Field class stores the grid *staggering* for that physical field. This is the relationship of how the data array of a field maps onto a grid (e.g. one item per cell located at the cell center, one item per cell located at the NW corner, one item per cell vertex, etc.). This means that different Fields which are on the same underlying ESMF Grid but have different staggerings can share the same Grid object without needing to replicate it multiple times.
Fields can be added to States for use in inter-Component data communications. Fields can also be added to FieldBundles, which are groups of Fields on the same underlying Grid. One motivation for packing Fields into FieldBundles is convenience; another is the ability to perform optimized collective data transfers.
Field communication capabilities include: data redistribution, regridding, scatter, gather, sparse-matrix multiplication, and halo update. These are discussed in more detail in the documentation for the specific method calls. ESMF does not currently support vector fields, so the components of a vector field must be stored as separate Field objects.

22.2 Constants

22.2.1 ESMF_FIELDSTATUS

DESCRIPTION:

An ESMF_Field can be in different status after initialization. Field status can be queried using `ESMF_FieldGet()` method.
The type of this flag is:
`type(ESMF_FieldStatus_Flag)`
The valid values are:

ESMF_FIELDSTATUS_EMPTY Field is empty without geombase or data storage. Such a Field can be added to a `ESMF_State` and participate `ESMF_StateReconcile()`.

ESMF_FIELDSTATUS_GRIDSET Field is partially created. It has a geombase object internally created and the geombase object associates with either a ESMF_Grid, or a ESMF_Mesh, or an ESMF_XGrid, or a ESMF_LocStream. It's an error to set another geombase object in such a Field. It can also be added to a ESMF_State and participate ESMF_StateReconcile().

ESMF_FIELDSTATUS_COMPLETE Field is completely created with geombase and data storage internally allocated.

22.2.2 ESMF_POLEMETHOD

DESCRIPTION:

When interpolating between two Grids which have been mapped to a sphere these can be used to specify the type of artificial pole to create on the source Grid during interpolation. Creating the pole allows destination points above the top row or below the bottom row of the source Grid to still be mapped.
The type of this flag is:
type(ESMF_PoleMethod_Flag)
The valid values are:

ESMF_POLEMETHOD_NONE No pole. Destination points which lie above the top or below the bottom row of the source Grid won't be mapped.

ESMF_POLEMETHOD_ALLAVG 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 source values surrounding the pole.

ESMF_POLEMETHOD_NPNTAVG 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 is set by using the regridPoleNPnts parameter and ranges from 1 to the number of nodes around the pole. This option is useful for interpolating values which may be zeroed out by averaging around the entire pole (e.g. vector components).

ESMF_POLEMETHOD_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 holes, some destination points may still not be able to be mapped to the source Grid.

22.2.3 ESMF_REGRIDMETHOD

DESCRIPTION:
Specify which interpolation method to use during regridding.
The type of this flag is:
type(ESMF_RegridMethod_Flag)
The valid values are:

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

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

ESMF_REGRIDMETHOD_CONSERVE First order conservative interpolation. Value of a destination cell is the weighted sum of the values of the source cells that it overlaps. The weights are determined by the amount the source cell overlaps the destination cell. Will typically give less accurate approximations to values than the other interpolation methods, however, will do a much better job preserving the integral of the value between the source and destination. Needs corner coordinate values to be provided in the Grid. Currently only works for Fields created on the Grid center stagger (or the Mesh element location).

22.3 Use and Examples

A Field serves as an annotator of data, since it carries a description of the grid it is associated with and metadata such as name and units. Fields can be used in this capacity alone, as convenient, descriptive containers into which arrays can be placed and retrieved. However, for most codes the primary use of Fields is in the context of import and export States, which are the objects that carry coupling information between Components. Fields enable data to be self-describing, and a State holding ESMF Fields contains data in a standard format that can be queried and manipulated.
The sections below go into more detail about Field usage.

22.3.1 Field create and destroy

Fields can be created and destroyed at any time during application execution. However, these Field methods require some time to complete. We do not recommend that the user create or destroy Fields inside performance-critical computational loops.
All versions of the `ESMF_FieldCreate()` routines require a Grid object as input, or require a Grid be added before most operations involving Fields can be performed. The Grid contains the information needed to know which Decomposition Elements (DEs) are participating in the processing of this Field, and which subsets of the data are local to a particular DE.
The details of how the create process happens depends on which of the variants of the `ESMF_FieldCreate()` call is used. Some of the variants are discussed below.
There are versions of the `ESMF_FieldCreate()` interface which create the Field based on the input Grid. The ESMF can allocate the proper amount of space but not assign initial values. The user code can then get the pointer to the uninitialized buffer and set the initial data values.
Other versions of the `ESMF_FieldCreate()` interface allow user code to attach arrays that have already been allocated by the user. Empty Fields can also be created in which case the data can be added at some later time.
For versions of Create which do not specify data values, user code can create an ArraySpec object, which contains information about the typekind and rank of the data values in the array. Then at Field create time, the appropriate amount of memory is allocated to contain the data which is local to each DE.
When finished with a `ESMF_Field`, the `ESMF_FieldDestroy` method removes it. However, the objects inside the `ESMF_Field` created externally should be destroyed separately, since objects can be added to more than one `ESMF_Field`. For example, the same `ESMF_Grid` can be referenced by multiple `ESMF_Field`s. In this case the internal Grid is not deleted by the `ESMF_FieldDestroy` call.

22.3.2 Get Fortran data pointer, bounds, and counts information from a Field

A user can get bounds and counts information from an `ESMF_Field` through the `ESMF_FieldGet()` interface. Also available through this interface is the intrinsic Fortran data pointer contained in the internal `ESMF_Array` object of an `ESMF_Field`. The bounds and counts information are DE specific for the associated Fortran data pointer.
For a better discussion of the terminologies, bounds and widths in ESMF e.g. exclusive, computational, total bounds for the lower and upper corner of data region, etc.., user can refer to the explanation of these concepts for Grid and Array in their respective sections in the *Reference Manual*, e.g. Section 24.2.6 on Array and Section 27.3.16 on Grid.
In this example, we first create a 3D Field based on a 3D Grid and Array. Then we use the `ESMF_FieldGet()` interface to retrieve the data pointer, potentially updating or verifying its values. We also retrieve the bounds and counts information of the 3D Field to assist in data element iteration.

```
xdim = 180
ydim = 90
zdim = 50
```

```
    ! create a 3D data Field from a Grid and Array.
    ! first create a Grid
    grid3d = ESMF_GridCreateNoPeriDim(minIndex=(/1,1,1/), &
            maxIndex=(/xdim,ydim,zdim/), &
            regDecomp=(/2,2,1/), name="grid", rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_GridGet(grid=grid3d, staggerloc=ESMF_STAGGERLOC_CENTER, &
           distgrid=distgrid3d, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_GridGetFieldBounds(grid=grid3d, localDe=0, &
        staggerloc=ESMF_STAGGERLOC_CENTER, totalCount=fa_shape, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    allocate(farray(fa_shape(1), fa_shape(2), fa_shape(3)) )

    ! create an Array
    array3d = ESMF_ArrayCreate(distgrid3d, farray, &
indexflag=ESMF_INDEX_DELOCAL, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! create a Field
    field = ESMF_FieldCreate(grid=grid3d, array=array3d, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! retrieve the Fortran data pointer from the Field
    call ESMF_FieldGet(field=field, localDe=0, farrayPtr=farray1, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! retrieve the Fortran data pointer from the Field and bounds
    call ESMF_FieldGet(field=field, localDe=0, farrayPtr=farray1, &
        computationalLBound=compLBnd, computationalUBound=compUBnd, &
        exclusiveLBound=exclLBnd, exclusiveUBound=exclUBnd, &
        totalLBound=totalLBnd, totalUBound=totalUBnd, &
        computationalCount=comp_count, &
        exclusiveCount=excl_count, &
        totalCount=total_count, &
        rc=rc)

    ! iterate through the total bounds of the field data pointer
    do k = totalLBnd(3), totalUBnd(3)
        do j = totalLBnd(2), totalUBnd(2)
            do i = totalLBnd(1), totalUBnd(1)
                farray1(i, j, k) = sin(2*i/total_count(1)*PI) + &
                    sin(4*j/total_count(2)*PI) + &
                    sin(8*k/total_count(2)*PI)
            enddo
        enddo
    enddo
```

22.3.3 Get Grid, Array, and other information from a Field

A user can get the internal ESMF_Grid and ESMF_Array from a ESMF_Field. Note that the user should not issue any destroy command on the retrieved grid or array object since they are referenced from within the ESMF_Field. The retrieved objects should be used in a read-only fashion to query additional information not directly available through the ESMF_FieldGet() interface.

```
call ESMF_FieldGet(field, grid=grid, array=array, &
    typekind=typekind, dimCount=dimCount, staggerloc=staggerloc, &
    gridToFieldMap=gridToFieldMap, &
    ungriddedLBound=ungriddedLBound, ungriddedUBound=ungriddedUBound, &
    totalLWidth=totalLWidth, totalUWidth=totalUWidth, &
    name=name, &
    rc=rc)
```

22.3.4 Create a Field with a Grid, typekind, and rank

A user can create an ESMF_Field from an ESMF_Grid and typekind/rank. This create method associates the two objects.
We first create a Grid with a regular distribution that is 10x20 index in 2x2 DEs. This version of Field create simply associates the data with the Grid. The data is referenced explicitly on a regular 2x2 uniform grid. Finally we create a Field from the Grid, typekind, rank, and a user specified StaggerLoc.
This example also illustrates a typical use of this Field creation method. By creating a Field from a Grid and typekind/rank, the user allows the ESMF library to create a internal Array in the Field. Then the user can use ESMF_FieldGet() to retrieve the Fortran data array and necessary bounds information to assign initial values to it.

```
! create a grid
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create a Field from the Grid and arrayspec
field1 = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_R4, &
    indexflag=ESMF_INDEX_DELOCAL, &
    staggerloc=ESMF_STAGGERLOC_CENTER, name="pressure", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(field1, localDe=0, farrayPtr=farray2dd, &
    totalLBound=ftlb, totalUBound=ftub, totalCount=ftc, rc=rc)

do i = ftlb(1), ftub(1)
    do j = ftlb(2), ftub(2)
        farray2dd(i, j) = sin(i/ftc(1)*PI) * cos(j/ftc(2)*PI)
    enddo
enddo

if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.5 Create a Field with a Grid and Arrayspec

A user can create an ESMF_Field from an ESMF_Grid and a ESMF_Arrayspec with corresponding rank and type. This create method associates the two objects.

We first create a Grid with a regular distribution that is 10x20 index in 2x2 DEs. This version of Field create simply associates the data with the Grid. The data is referenced explicitly on a regular 2x2 uniform grid. Then we create an ArraySpec. Finally we create a Field from the Grid, ArraySpec, and a user specified StaggerLoc.
This example also illustrates a typical use of this Field creation method. By creating a Field from a Grid and an ArraySpec, the user allows the ESMF library to create a internal Array in the Field. Then the user can use `ESMF_FieldGet()` to retrieve the Fortran data array and necessary bounds information to assign initial values to it.

```
! create a grid
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! setup arrayspec
call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create a Field from the Grid and arrayspec
field1 = ESMF_FieldCreate(grid, arrayspec, &
     indexflag=ESMF_INDEX_DELOCAL, &
     staggerloc=ESMF_STAGGERLOC_CENTER, name="pressure", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(field1, localDe=0, farrayPtr=farray2dd, &
    totalLBound=ftlb, totalUBound=ftub, totalCount=ftc, rc=rc)

do i = ftlb(1), ftub(1)
    do j = ftlb(2), ftub(2)
        farray2dd(i, j) = sin(i/ftc(1)*PI) * cos(j/ftc(2)*PI)
    enddo
enddo

if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

A user can also create an ArraySpec that has a different rank from the Grid, For example, the following code shows creation of of 3D Field from a 2D Grid using a 3D ArraySpec.
This example also demonstrates the technique to create a typical 3D data Field that has 2 gridded dimensions and 1 ungridded dimension.
First we create a 2D grid with an index space of 180x360 equivalent to 180x360 Grid cells (note that for a distributed memory computer, this means each grid cell will be on a separate PE!). In the FieldCreate call, we use gridToFieldMap to indicate the mapping between Grid dimension and Field dimension. For the ungridded dimension (typically the altitude), we use ungriddedLBound and ungriddedUBound to describe its bounds. Internally the ungridded dimension has a stride of 1, so the number of elements of the ungridded dimension is ungriddedUBound - ungriddedLBound + 1. Note that gridToFieldMap in this specific example is (/1,2/) which is the default value so the user can neglect this argument for the FieldCreate call.

```
grid2d = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), &
      maxIndex=(/180,360/), regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 3, ESMF_TYPEKIND_R4, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

field1 = ESMF_FieldCreate(grid2d, arrayspec, &
     indexflag=ESMF_INDEX_DELOCAL, &
```

```
        staggerloc=ESMF_STAGGERLOC_CENTER, &
        gridToFieldMap=(/1,2/), &
        ungriddedLBound=(/1/), ungriddedUBound=(/50/), &
        name="pressure", rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.6 Create a Field with a Grid and Array

A user can create an `ESMF_Field` from an `ESMF_Grid` and a `ESMF_Array`. The Grid was created in the previous example.
This example creates a 2D `ESMF_Field` from a 2D `ESMF_Grid` and a 2D `ESMF_Array`.

```
    ! Get necessary information from the Grid
    call ESMF_GridGet(grid, staggerloc=ESMF_STAGGERLOC_CENTER, &
        distgrid=distgrid, rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Create a 2D ESMF_TYPEKIND_R4 arrayspec
    call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Create a ESMF_Array from the arrayspec and distgrid
    array2d = ESMF_ArrayCreate(arrayspec=arrayspec, &
            distgrid=distgrid, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Create a ESMF_Field from the grid and array
    field4 = ESMF_FieldCreate(grid, array2d, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.7 Create an empty Field and complete it with FieldEmptySet and FieldEmptyComplete

A user can create an `ESMF_Field` in three steps: first create an empty `ESMF_Field`; then set a `ESMF_Grid` on the empty `ESMF_Field`; and finally complete the `ESMF_Field` by calling `ESMF_FieldEmptyComplete`.

```
    ! create an empty Field
    field3 = ESMF_FieldEmptyCreate(name="precip", rc=rc)


    ! use FieldGet to retrieve the Field Status
    call ESMF_FieldGet(field3, status=fstatus, rc=rc)
```

Once the Field is created, we can verify that the status of the Field is `ESMF_FIELDSTATUS_EMPTY`.

```
    ! Test the status of the Field
    if(fstatus /= ESMF_FIELDSTATUS_EMPTY) finalrc = ESMF_FAILURE
```

Next we set a Grid on the empty Field. We use the 2D grid created in a previous example simply to demonstrate the method. The Field data points will be on east edge of the Grid cells with the specified `ESMF_STAGGERLOC_EDGE1`.

```
    ! Set a grid on the Field
    call ESMF_FieldEmptySet(field3, grid2d, &
             staggerloc=ESMF_STAGGERLOC_EDGE1, rc=rc)
```

```
! use FieldGet to retrieve the Field Status again
call ESMF_FieldGet(field3, status=fstatus, rc=rc)


! Test the status of the Field
if(fstatus /= ESMF_FIELDSTATUS_GRIDSET) finalrc = ESMF_FAILURE
```

The partially created Field is completed by specifying the typekind of its data storage. This method is overloaded with one of the following parameters, arrayspec, typekind, Fortran array, or Fortran array pointer. Additional optional arguments can be used to specify ungridded dimensions and halo regions similar to the other Field creation methods.

```
! Complete the Field by specifying the data typekind
! to be allocated internally.
call ESMF_FieldEmptyComplete(field3, typekind=ESMF_TYPEKIND_R8, &
  ungriddedLBound=(/1/), ungriddedUBound=(/5/), rc=rc)


! use FieldGet to retrieve the Field Status again
call ESMF_FieldGet(field3, status=fstatus, rc=rc)


! Test the status of the Field
if(fstatus /= ESMF_FIELDSTATUS_COMPLETE) finalrc = ESMF_FAILURE
```

22.3.8 Create an empty Field and complete it with FieldEmptyComplete

A user can create an empty `ESMF_Field`. Then the user can finalize the empty `ESMF_Field` from a `ESMF_Grid` and a intrinsic Fortran data array. This interface is overloaded for typekind and rank of the Fortran data array.
In this example, both grid and Fortran array pointer are 2 dimensional and each dimension index maps in order, i.e. 1st dimension of grid maps to 1st dimension of Fortran array pointer, 2nd dimension of grid maps to 2nd dimension of Fortran array pointer, so on and so forth.
In order to create or complete a Field from a Grid and a Fortran array pointer, certain rules of the Fortran array bounds must be obeyed. We will discuss these rules as we progress in Field creation examples. We will make frequent reference to the terminologies for bounds and widths in ESMF. For a better discussion of these terminologies and concepts behind them, e.g. exclusive, computational, total bounds for the lower and upper corner of data region, etc.., users can refer to the explanation of these concepts for Grid and Array in their respective sections in the *Reference Manual*, e.g. Section 24.2.6 on Array and Section 27.3.16 on Grid. The examples here are designed to help a user to get up to speed with creating Fields for typical use.
This example introduces a helper method, the `ESMF_GridGetFieldBounds` interface that facilitates the computation of Fortran data array bounds and shape to assist `ESMF_FieldEmptyComplete` finalizing a Field from a instrinsic Fortran data array and a Grid.

```
! create an empty Field
field3 = ESMF_FieldEmptyCreate(name="precip", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! use FieldGet to retrieve total counts
call ESMF_GridGetFieldBounds(grid2d, localDe=0, &
    staggerloc=ESMF_STAGGERLOC_CENTER, totalCount=ftc, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! allocate the 2d Fortran array based on retrieved total counts
allocate(farray2d(ftc(1), ftc(2)))

! finalize the Field
call ESMF_FieldEmptyComplete(field3, grid2d, farray2d, rc=rc)
```

22.3.9 Create a 7D Field with a 5D Grid and 2D ungridded bounds from a Fortran data array

In this example, we will show how to create a 7D Field from a 5D `ESMF_Grid` and 2D ungridded bounds with arbitrary halo widths and gridToFieldMap.
We first create a 5D DistGrid and a 5D Grid based on the DistGrid; then `ESMF_GridGetFieldBounds` computes the shape of a 7D array in fsize. We can then create a 7D Field from the 5D Grid and the 7D Fortran data array with other assimilating parameters.

```
    ! create a 5d distgrid
    distgrid5d = ESMF_DistGridCreate(minIndex=(/1,1,1,1,1/), &
        maxIndex=(/10,4,10,4,6/), regDecomp=(/2,1,2,1,1/), rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Create a 5d Grid
    grid5d = ESMF_GridCreate(distgrid=distgrid5d, name="grid", rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! use FieldGet to retrieve total counts
    call ESMF_GridGetFieldBounds(grid5d, localDe=0, ungriddedLBound=(/1,2/), &
        ungriddedUBound=(/4,5/), &
        totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
        gridToFieldMap=(/3,2,5,4,1/), &
        totalCount=fsize, &
        rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! allocate the 7d Fortran array based on retrieved total counts
    allocate(farray7d(fsize(1), fsize(2), fsize(3), fsize(4), fsize(5), &
fsize(6), fsize(7)))

    ! create the Field
    field7d = ESMF_FieldCreate(grid5d, farray7d, ESMF_INDEX_DELOCAL, &
        ungriddedLBound=(/1,2/), ungriddedUBound=(/4,5/), &
        totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
        gridToFieldMap=(/3,2,5,4,1/), &
        rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

A user can allocate the Fortran array in a different manner using the lower and upper bounds returned from FieldGet through the optional totalLBound and totalUBound arguments. In the following example, we create another 7D Field by retrieving the bounds and allocate the Fortran array with this approach. In this scheme, indexing the Fortran array is sometimes more convenient than using the shape directly.

```
    call ESMF_GridGetFieldBounds(grid5d, localDe=0, ungriddedLBound=(/1,2/), &
        ungriddedUBound=(/4,5/), &
        totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
        gridToFieldMap=(/3,2,5,4,1/), &
        totalLBound=flbound, totalUBound=fubound, &
        rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    allocate(farray7d2(flbound(1):fubound(1), flbound(2):fubound(2), &
       flbound(3):fubound(3), flbound(4):fubound(4), &
       flbound(5):fubound(5), flbound(6):fubound(6), &
                       flbound(7):fubound(7)) )
```

```
field7d2 = ESMF_FieldCreate(grid5d, farray7d2, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/1,2/), ungriddedUBound=(/4,5/), &
    totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
    gridToFieldMap=(/3,2,5,4,1/), &
    rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.10 Create a 2D Field with a 2D Grid and a Fortran data array

A user can create an `ESMF_Field` directly from an `ESMF_Grid` and an intrinsic Fortran data array. This interface is overloaded for typekind and rank of the Fortran data array.
In the following example, each dimension size of the Fortran array is equal to the exclusive bounds of its corresponding Grid dimension queried from the Grid through `ESMF_GridGet()` public interface.
Formally let fa_shape(i) be the shape of i-th dimension of user supplied Fortran array, then rule 1 states:

```
(1) fa_shape(i) = exclusiveCount(i)
              i = 1...GridDimCount
```

fa_shape(i) defines the shape of i-th dimension of the Fortran array. ExclusiveCount are the number of data elements of i-th dimension in the exclusive region queried from `ESMF_GridGet` interface. *Rule 1 assumes that the Grid and the Fortran intrinsic array have same number of dimensions; and optional arguments of FieldCreate from Fortran array are left unspecified using default setup*. These assumptions are true for most typical use of FieldCreate from Fortran data array. This is the easiest way to create a Field from a Grid and Fortran intrinsic data array.
Fortran array dimension sizes (called shape in most Fortran language books) are equivalent to the bounds and counts used in this manual. The following equation holds:

```
fa_shape(i) = shape(i) = counts(i) = upper_bound(i) - lower_bound(i) + 1
```

These typically mean the same concept unless specifically explained to mean something else. For example, ESMF uses DimCount very often to mean number of dimensions instead of its meaning implied in the above equation. We'll clarify the meaning of a word when ambiguity could occur.
Rule 1 is most useful for a user working with Field creation from a Grid and a Fortran data array in most scenarios. It extends to higher dimension count, 3D, 4D, etc... Typically, as the code example demonstrates, a user first creates a Grid , then uses `ESMF_GridGet()` to retrieve the exclusive counts. Next the user calculates the shape of each Fortran array dimension according to rule 1. The Fortran data array is allocated and initialized based on the computed shape. A Field can either be created in one shot created empty and finished using `ESMF_FieldEmptyComplete`. There are important details that can be skipped but are good to know for `ESMF_FieldEmptyComplete` and `ESMF_FieldCreate` from a Fortran data array. 1) these methods require *each PET contains exactly one DE*. This implies that a code using FieldCreate from a data array or FieldEmptyComplete must have the same number of DEs and PETs, formally $n_{DE} = n_{PET}$. Violation of this condition will cause run time failures. 2) the bounds and counts retrieved from GridGet are DE specific or equivalently PET specific, which means that *the Fortran array shape could be different from one PET to another*.

```
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_GridGet(grid, localDE=0, staggerloc=ESMF_STAGGERLOC_CENTER, &
    exclusiveCount=gec, rc=rc)
```

```
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

allocate(farray(gec(1), gec(2)) )

field = ESMF_FieldCreate(grid, farray, ESMF_INDEX_DELOCAL, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.11 Create a 2D Field with a 2D Grid and a Fortran data pointer

The setup of this example is similar to the previous section except that the Field is created from a data pointer instead of a data array. We highlight the ability to deallocate the internal fortran data pointer queried from the Field. This gives a user more flexibility with memory management.

```
allocate(farrayPtr(gec(1), gec(2)) )

field = ESMF_FieldCreate(grid, farrayPtr, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_FieldGet(field, farrayPtr=farrayPtr2, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
! deallocate the retrieved fortran array pointer
deallocate(farrayPtr2)
```

22.3.12 Create a 3D Field with a 2D Grid and a 3D Fortran data array

This example demonstrates a typical use of `ESMF_Field` combining a 2D grid and a 3D Fortran native data array. One immediate problem follows: how does one define the bounds of the ungridded dimension? This is solved by the optional arguments `ungriddedLBound` and `ungriddedUBound` of the `ESMF_FieldCreate` interface. By definition, `ungriddedLBound` and `ungriddedUBound` are both 1 dimensional integer Fortran arrays.
Formally, let fa_shape(j=1...FieldDimCount-GridDimCount) be the shape of the ungridded dimensions of a Field relative to the Grid used in Field creation. The Field dimension count is equal to the number of dimensions of the Fortran array, which equals the number of dimensions of the resultant Field. GridDimCount is the number of dimensions of the Grid.
fa_shape(j) is computed as:

```
fa_shape(j) = ungriddedUBound(j) - ungriddedLBound(j) + 1
```

fa_shape is easy to compute when the gridded and ungridded dimensions do not mix. However, it's conceivable that at higher dimension count, gridded and ungridded dimensions can interleave. To aid the computation of ungridded dimension shape we formally introduce the mapping concept.
Let $map_{A,B}(i = 1...n_A) = i_B$, and $i_B \in [\phi, 1...n_B]$. n_A is the number of elements in set A, n_B is the number of elements in set B. $map_{A,B}(i)$ defines a mapping from i-th element of set A to i_B-th element in set B. $i_B = \phi$ indicates there does not exist a mapping from i-th element of set A to set B.
Suppose we have a mapping from dimension index of ungriddedLBound (or ungriddedUBound) to Fortran array dimension index, called ugb2fa. By definition, n_A equals to the dimension count of ungriddedLBound (or ungriddedUBound), n_B equals to the dimension count of the Fortran array. We can now formulate the computation of ungridded dimension shape as rule 2:

```
(2) fa_shape(ugb2fa(j)) = ungriddedUBound(j) - ungriddedLBound(j) + 1
                      j = 1..FortranArrayDimCount - GridDimCount
```

The mapping can be computed in linear time proportional to the Fortran array dimension count (or rank) using the following algorithm in pseudocode:

```
map_index = 1
do i = 1, farray_rank
    if i-th dimension of farray is ungridded
        ugb2fa(map_index) = i
        map_index = map_index + 1
    endif
enddo
```

Here we use rank and dimension count interchangably. These 2 terminologies are typically equivalent. But there are subtle differences under certain conditions. Rank is the total number of dimensions of a tensor object. Dimension count allows a finer description of the heterogeneous dimensions in that object. For example, A Field of rank 5 can have 3 gridded dimensions and 2 ungridded dimensions. Rank is precisely the summation of dimension count of all types of dimensions.

For example, if a 5D array is used with a 3D Grid, there are 2 ungridded dimensions: ungriddedLBound=(/1,2/) and ungriddedUBound=(/5,7/). Suppose the distribution of dimensions look like (O, X, O, X, O), O means gridded, X means ungridded. Then the mapping from ungridded bounds to Fortran array is ugb2fa=(/2, 4/). The shape of 2nd and 4th dimension of Fortran array should equal (5, 8).

Back to our 3D Field created from a 2D Grid and 3D Fortran array example, suppose the 3rd Field dimension is ungridded, ungriddedLBound=(/3/), ungriddedUBound=(/9/). First we use rule 1 to compute shapes of the gridded Fortran array dimension, then we use rule 2 to compute shapes of the ungridded Fortran array dimension. In this example, we used the exclusive bounds obtained in the previous example.

```
fa_shape(1) = gec(1) ! rule 1
fa_shape(2) = gec(2)
fa_shape(3) = 7 ! rule 2 9-3+1
allocate(farray3d(fa_shape(1), fa_shape(2), fa_shape(3)))
field = ESMF_FieldCreate(grid, farray3d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.13 Create a 3D Field with a 2D Grid and a 3D Fortran data array with gridToFieldMap argument

Building upon the previous example, we will create a 3D Field from a 2D grid and 3D array but with a slight twist. In this example, we introduce the gridToFieldMap argument that allows a user to map Grid dimension index to Field dimension index.

In this example, both dimensions of the Grid are distributed and the mapping from DistGrid to Grid is (/1,2/). We will introduce rule 3 assuming distgridToGridMap=(/1,2,3...gridDimCount/), and distgridDimCount equals to gridDimCount. This is a reasonable assumption in typical Field use.

We apply the mapping gridToFieldMap on rule 1 to create rule 3:

```
(3) fa_shape(gridToFieldMap(i)) = exclusiveCount(i)
                              i = 1,..GridDimCount.
```

Back to our example, suppose the 2nd Field dimension is ungridded, ungriddedLBound=(/3/), ungriddedUBound=(/9/). gridToFieldMap=(/3,1/), meaning the 1st Grid dimension maps to 3rd Field dimension, and 2nd Grid dimension maps to 1st Field dimension.

First we use rule 3 to compute shapes of the gridded Fortran array dimension, then we use rule 2 to compute shapes of the ungridded Fortran array dimension. In this example, we use the exclusive bounds obtained in the previous example.

```
gridToFieldMap2d(1) = 3
gridToFieldMap2d(2) = 1
do i = 1, 2
    fa_shape(gridToFieldMap2d(i)) = gec(i)
end do
fa_shape(2) = 7
allocate(farray3d(fa_shape(1), fa_shape(2), fa_shape(3)))
field = ESMF_FieldCreate(grid, farray3d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    gridToFieldMap=gridToFieldMap2d, &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.14 Create a 3D Field with a 2D Grid and a 3D Fortran data array with halos

This example is similar to example 22.3.13, in addition we will show a user can associate different halo width to a Fortran array to create a Field through the totalLWidth and totalUWidth optional arguments. A diagram of the dimension configuration from Grid, halos, and Fortran data array is shown here.

The `ESMF_FieldCreate()` interface supports creating a Field from a Grid and a Fortran array padded with halos on the distributed dimensions of the Fortran array. Using this technique one can avoid passing non-contiguous Fortran array slice to FieldCreate. It guarantees the same exclusive region, and by using halos, it also defines a bigger total region to contain the entire contiguous memory block of the Fortran array.
The elements of totalLWidth and totalUWidth are applied in the order distributed dimensions appear in the Fortran array. By definition, totalLWidth and totalUWidth are 1 dimensional arrays of non-negative integer values. The size of haloWidth arrays is equal to the number of distributed dimensions of the Fortran array, which is also equal to the number of distributed dimensions of the Grid used in the Field creation.
Because the order of totalWidth (representing both totalLWidth and totalUWidth) element is applied to the order distributed dimensions appear in the Fortran array dimensions, it's quite simple to compute the shape of distributed dimensions of the Fortran array. They are done in a similar manner when applying ungriddedLBound and ungriddedUBound to ungridded dimensions of the Fortran array defined by rule 2.
Assume we have the mapping from the dimension index of totalWidth to the dimension index of Fortran array, called mhw2fa; and we also have the mapping from dimension index of Fortran array to dimension index of the Grid, called fa2g. The shape of distributed dimensions of a Fortran array can be computed by rule 4:

```
(4) fa_shape(mhw2fa(k)) = exclusiveCount(fa2g(mhw2fa(k)) +
                          totalUWidth(k) + totalLWidth(k)
                      k = 1...size(totalWidth)
```

This rule may seem confusing but algorithmically the computation can be done by the following pseudocode:

```
fa_index = 1
do i = 1, farray_rank
   if i-th dimension of Fortran array is distributed
       fa_shape(i) = exclusiveCount(fa2g(i)) +
                     totalUWidth(fa_index) + totalLWidth(fa_index)
```

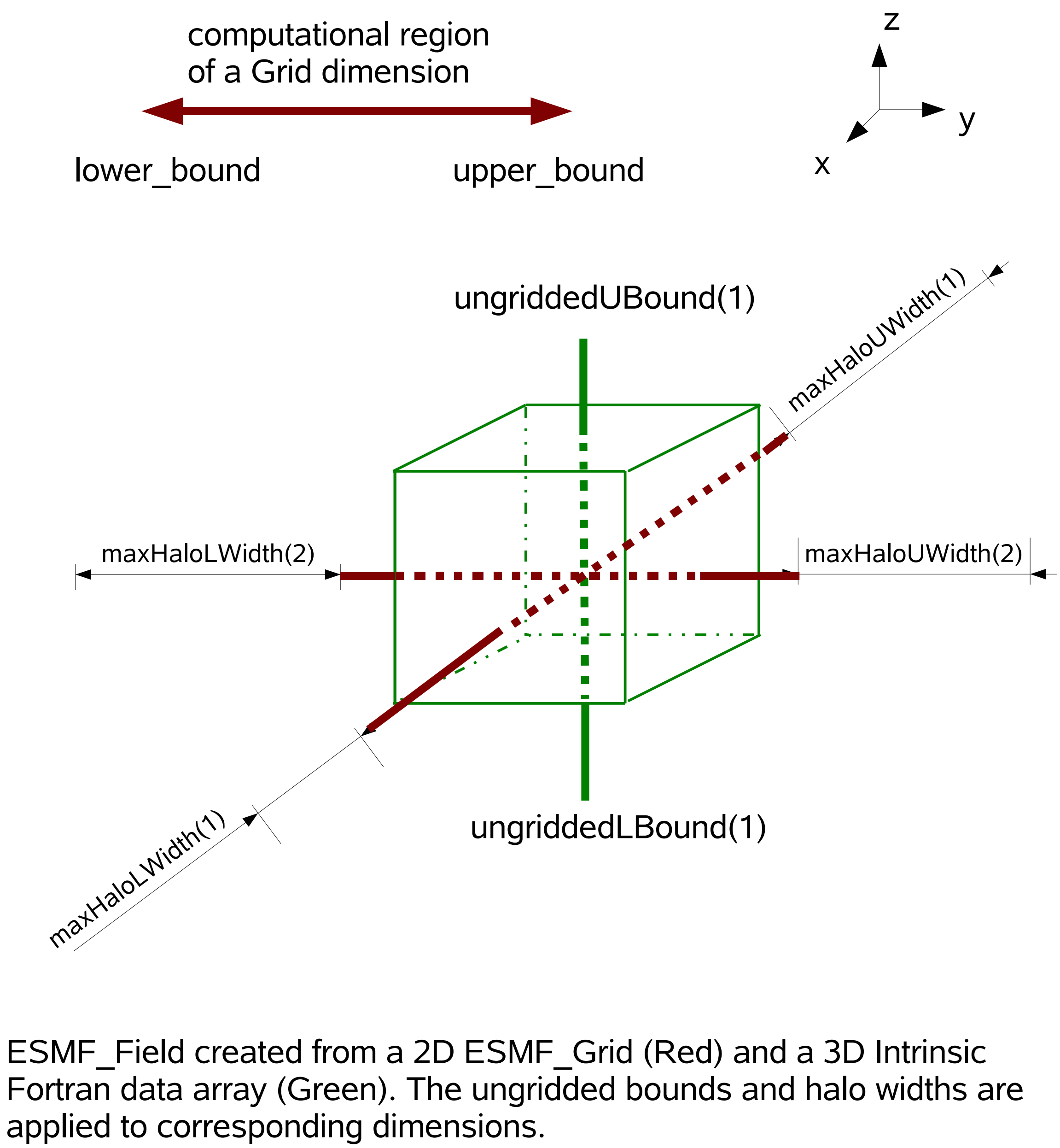


Figure 12: Field dimension configuration from Grid, halos, and Fortran data array.

```
            fa_index = fa_index + 1
        endif
    enddo
```

The only complication then is to figure out the mapping from Fortran array dimension index to Grid dimension index. This process can be done by computing the reverse mapping from Field to Grid.
Typically, we don't have to consider these complications if the following conditions are met: 1) All Grid dimensions are distributed. 2) DistGrid in the Grid has a dimension index mapping to the Grid in the form of natural order (/1,2,3,.../). This natural order mapping is the default mapping between various objects throughout ESMF. 3) Grid to Field mapping is in the form of natural order, i.e. default mapping. These seem like a lot of conditions but they are the default case in the interaction among DistGrid, Grid, and Field. When these conditions are met, which is typically true, the shape of distributed dimensions of Fortran array follows rule 5 in a simple form:

```
(5) fa_shape(k) = exclusiveCount(k) +
                  totalUWidth(k) + totalLWidth(k)
              k = 1...size(totalWidth)
```

Let's examine an example on how to apply rule 5. Suppose we have a 5D array and a 3D Grid that has its first 3 dimensions mapped to the first 3 dimensions of the Fortran array. totalLWidth=(/1,2,3/), totalUWidth=(/7,9,10/), then by rule 5, the following pseudo code can be used to compute the shape of the first 3 dimensions of the Fortran array. The shape of the remaining two ungridded dimensions can be computed according to rule 2.

```
do k = 1, 3
    fa_shape(k) = exclusiveCount(k) +
                  totalUWidth(k) + totalLWidth(k))
enddo
```

Suppose now gridToFieldMap=(/2,3,4/) instead which says the first dimension of Grid maps to the 2nd dimension of Field (or Fortran array) and so on and so forth, we can obtain a more general form of rule 5 by introducing first_distdim_index shift when Grid to Field map (gridToFieldMap) is in the form of (/a,a+1,a+2.../).

```
(6) fa_shape(k+first_distdim_index-1) = exclusiveCount(k) +
                                        totalUWidth(k) + totalLWidth(k)
                                    k = 1...size(totalWidth)
```

It's obvious that first_distdim_index=a. If the first dimension of the Fortran array is distributed, then rule 6 degenerates into rule 5, which is the typical case.
Back to our example creating a 3D Field from a 2D Grid and a 3D intrinsic Fortran array, we will use the Grid created from previous example that satisfies condition 1 and 2. We'll also use a simple gridToFieldMap (1,2) which is the default mapping that satisfies condition 3. First we use rule 5 to compute the shape of distributed dimensions then we use rule 2 to compute the shape of the ungridded dimensions.

```
gridToFieldMap2d(1) = 1
gridToFieldMap2d(2) = 2
totalLWidth2d(1) = 3
totalLWidth2d(2) = 4
```

```
totalUWidth2d(1) = 3
totalUWidth2d(2) = 5
do k = 1, 2
    fa_shape(k) = gec(k) + totalLWidth2d(k) + totalUWidth2d(k)
end do
fa_shape(3) = 7          ! 9-3+1
allocate(farray3d(fa_shape(1), fa_shape(2), fa_shape(3)))
field = ESMF_FieldCreate(grid, farray3d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    totalLWidth=totalLWidth2d, totalUWidth=totalUWidth2d, &
    gridToFieldMap=gridToFieldMap2d, &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.15 Create a Field from a LocStream, typekind, and rank

In this example, an `ESMF_Field` is created from an `ESMF_LocStream` and typekind/rank. The location stream object is uniformly distributed in a 1 dimensional space on 4 DEs. The rank is 1 dimensional. Please refer to LocStream examples section for more information on LocStream creation.

```
locs = ESMF_LocStreamCreate(minIndex=1, maxIndex=16, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

field = ESMF_FieldCreate(locs, typekind=ESMF_TYPEKIND_I4, &
    rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.16 Create a Field from a LocStream and arrayspec

In this example, an `ESMF_Field` is created from an `ESMF_LocStream` and an `ESMF_Arrayspec`. The location stream object is uniformly distributed in a 1 dimensional space on 4 DEs. The arrayspec is 1 dimensional. Please refer to LocStream examples section for more information on LocStream creation.

```
locs = ESMF_LocStreamCreate(minIndex=1, maxIndex=16, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_I4, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

field = ESMF_FieldCreate(locs, arrayspec, &
    rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.17 Create a Field from a Mesh, typekind, and rank

In this example, an `ESMF_Field` is created from an `ESMF_Mesh` and typekind/rank. The mesh object is on a Euclidean surface that is partitioned to a 2x2 rectangular space with 4 elements and 9 nodes. The nodal space is represented by a distgrid with 9 indices. Field is created on locally owned nodes on each PET. Therefore, the created Field has 9 data points globally. The mesh object can be represented by the picture below. For more information on Mesh creation, please see Section 29.3.1.

```
             Mesh Ids

 2.0   7 ------- 8 -------- 9
       |         |          |
       |    3    |    4     |
       |         |          |
 1.0   4 ------- 5 -------- 6
       |         |          |
       |    1    |    2     |
       |         |          |
 0.0   1 ------- 2 -------- 3

      0.0       1.0        2.0

     Node Ids at corners
     Element Ids in centers


            Mesh Owners

 2.0   2 ------- 2 -------- 3
       |         |          |
       |    2    |    3     |
       |         |          |
 1.0   0 ------- 0 -------- 1
       |         |          |
       |    0    |    1     |
       |         |          |
 0.0   0 ------- 0 -------- 1

      0.0       1.0        2.0

     Node Owners at corners
     Element Owners in centers
```

```
! Create Mesh structure in 1 step
mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=nodeIds, nodeCoords=nodeCoords, &
       nodeOwners=nodeOwners, elementIds=elemIds,&
       elementTypes=elemTypes, elementConn=elemConn, &
       rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Field is created on the 1 dimensinonal nodal distgrid. On
! each PET, Field is created on the locally owned nodes.
field = ESMF_FieldCreate(mesh, typekind=ESMF_TYPEKIND_I4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.18 Create a Field from a Mesh and arrayspec

In this example, an `ESMF_Field` is created from an `ESMF_Mesh` and an `ESMF_Arrayspec`. The mesh object is on a Euclidean surface that is partitioned to a 2x2 rectangular space with 4 elements and 9 nodes. The nodal space is represented by a distgrid with 9 indices. Field is created on locally owned nodes on each PET. Therefore, the created Field has 9 data points globally. The mesh object can be represented by the picture below. For more information on

Mesh creation, please see Section 29.3.1.

```
! Create Mesh structure in 1 step
mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=nodeIds, nodeCoords=nodeCoords, &
       nodeOwners=nodeOwners, elementIds=elemIds,&
       elementTypes=elemTypes, elementConn=elemConn, &
       rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_I4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Field is created on the 1 dimensinonal nodal distgrid. On
! each PET, Field is created on the locally owned nodes.
field = ESMF_FieldCreate(mesh, arrayspec, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.19 Create a Field from a Mesh and an Array

In this example, an `ESMF_Field` is created from an `ESMF_Mesh` and an `ESMF_Array`. The mesh object is created in the previous example and the array object is retrieved from the field created in the previous example too.

```
call ESMF_MeshGet(mesh, nodalDistgrid=distgrid, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
array = ESMF_ArrayCreate(distgrid=distgrid, arrayspec=arrayspec, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
! query the array from the previous example
call ESMF_FieldGet(field, array=array, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
! create a Field from a mesh and an array
field1 = ESMF_FieldCreate(mesh, array, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.20 Create a Field from a Mesh and an ArraySpec with optional features

In this example, an `ESMF_Field` is created from an `ESMF_Mesh` and an `ESMF_ArraySpec`. The mesh object is created in the previous example. The Field is also created with optional arguments such as ungridded dimensions and dimension mapping.
In this example, the mesh is mapped to the 2nd dimension of the `ESMF_Field`, with its first dimension being the ungridded dimension with bounds 1,3.

```
call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_I4, rc=rc)
field = ESMF_FieldCreate(mesh, arrayspec=arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/3/), rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.21 Create a Field with replicated dimensions

In this example an `ESMF_Field` with replicated dimension is created from an `ESMF_Grid` and an `ESMF_Arrayspec`. A user can also use other `ESMF_FieldCreate()` methods to create replicated dimension Field, this example illustrates the key concepts and use of a replicated dimension Field.

Normally gridToFieldMap argument in `ESMF_FieldCreate()` should not contain 0 value entries. However, for Field with replicated dimension, a 0 entry in gridToFieldMap indicates the corresponding Grid dimension is replicated in the Field. In such a Field, the rank of the Field is no longer necessarily greater than its Grid rank. An example will make this clear. We will start by creating Distgrid and Grid.

```
! create 4D distgrid
distgrid = ESMF_DistGridCreate(minIndex=(/1,1,1,1/), &
    maxIndex=(/6,4,6,4/), regDecomp=(/2,1,2,1/), rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create 4D grid on top of the 4D distgrid
grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create 3D arrayspec
call ESMF_ArraySpecSet(arrayspec, 3, ESMF_TYPEKIND_R8, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

In this example, a user creates a 3D Field with replicated dimension replicated along the 2nd and 4th dimension of its underlying 4D Grid. In addition, the 2nd dimension of the Field is ungridded (why?). The 1st and 3rd dimensions of the Field have halos.

```
! create field, 2nd and 4th dimensions of the Grid are replicated
field = ESMF_FieldCreate(grid, arrayspec, indexflag=ESMF_INDEX_DELOCAL, &
    gridToFieldMap=(/1,0,2,0/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/4/), &
    totalLWidth=(/1,1/), totalUWidth=(/4,5/), &
    staggerloc=ESMF_STAGGERLOC_CORNER, &
    rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! get basic information from the field
call ESMF_FieldGet(field, grid=grid1, array=array, typekind=typekind, &
    dimCount=dimCount, staggerloc=lstaggerloc, &
    gridToFieldMap=lgridToFieldMap, ungriddedLBound=lungriddedLBound, &
    ungriddedUBound=lungriddedUBound, totalLWidth=ltotalLWidth, &
    totalUWidth=ltotalUWidth, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

! get bounds information from the field
call ESMF_FieldGet(field, localDe=0, farrayPtr=farray, &
    exclusiveLBound=felb, exclusiveUBound=feub, exclusiveCount=fec, &
    computationalLBound=fclb, computationalUBound=fcub, &
    computationalCount=fcc, totalLBound=ftlb, totalUBound=ftub, &
    totalCount=ftc, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

Next we verify that the field and array bounds agree with each other

```
call ESMF_ArrayGet(array, rank=arank, dimCount=adimCount, rc=rc)
if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

gridrank_repdim = 0
do i = 1, size(gridToFieldMap)
```

```
        if(gridToFieldMap(i) == 0) gridrank_repdim = gridrank_repdim + 1
    enddo
```

Number of undistributed dimension of the array *X* is computed from total rank of the array *A*, the dimension count of its underlying distgrid *B* and number of replicated dimension in the distgrid *C*. We have the following formula: X = A - (B - C)

```
    allocate(audlb(arank-adimCount+gridrank_repdim), &
audub(arank-adimCount+gridrank_repdim))
    call ESMF_ArrayGet(array, exclusiveLBound=aelb, exclusiveUBound=aeub, &
        computationalLBound=aclb, computationalUBound=acub, &
        totalLBound=atlb, totalUBound=atub, &
        undistLBound=audlb, undistUBound=audub, &
        rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! verify the ungridded bounds from field match
    ! undistributed bounds from its underlying array
    do i = 1, arank-adimCount
        if(lungriddedLBound(i) .ne. audlb(i) ) &
            rc = ESMF_FAILURE
    enddo
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    do i = 1, arank-adimCount
        if(lungriddedUBound(i) .ne. audub(i) ) &
            rc = ESMF_FAILURE
    enddo
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

We then verify the data in the replicated dimension Field can be updated and accessed.

```
    do ik = ftlb(3), ftub(3)
     do ij = ftlb(2), ftub(2)
      do ii = ftlb(1), ftub(1)
        farray(ii,ij,ik) = ii+ij*2+ik
      enddo
     enddo
    enddo
    ! access and verify
    call ESMF_FieldGet(field, localDe=0, farrayPtr=farray1, &
        rc=rc)
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE
    do ik = ftlb(3), ftub(3)
     do ij = ftlb(2), ftub(2)
      do ii = ftlb(1), ftub(1)
        n = ii+ij*2+ik
        if(farray1(ii,ij,ik) .ne. n ) rc = ESMF_FAILURE
      enddo
     enddo
    enddo
    if (rc.NE.ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! release resources
```

```
call ESMF_FieldDestroy(field)
call ESMF_GridDestroy(grid)
call ESMF_DistGridDestroy(distgrid)
```

22.3.22 Create a Field on an arbitrarily distributed Grid

With the introduction of Field on arbitrarily distributed Grid, Field has two kinds of dimension count: one associated geometrical (or physical) dimensionality, the other one associated with its memory index space representation. Field and Grid dimCount reflect the physical index space of the objects. A new type of dimCount rank should be added to both of these entities. rank gives the number of dimensions of the memory index space of the objects. This would be the dimension of the pointer pulled out of Field and the size of the bounds vector, for example.

For non-arbitrary Grids rank=dimCount, but for grids and fields with arbitrary dimensions rank = dimCount - (number of Arb dims) + 1 (Internally Field can use the Arb info from the grid to create the mapping from the Field Array to the DistGrid)

When creating a Field size(GridToFieldMap)=dimCount for both Arb and Non-arb grids This array specifies the mapping of Field to Grid identically for both Arb and Nonarb grids If a zero occurs in an entry corresponding to any arbitrary dimension, then a zero must occur in every entry corresponding to an arbitrary dimension (i.e. all arbitrary dimensions must either be all replicated or all not replicated, they can't be broken apart).

In this example an `ESMF_Field` is created from an arbitrarily distributed `ESMF_Grid` and an `ESMF_Arrayspec`. A user can also use other `ESMF_FieldCreate()` methods to create such a Field, this example illustrates the key concepts and use of Field on arbitrary distributed Grid.

The Grid is 3 dimensional in physics index space but the first two dimension are collapsed into a single memory index space. Thus the result Field is 3D in physics index space and 2D in memory index space. This is made obvious with the 2D arrayspec used to create this Field.

```
! create a 3D grid with the first 2 dimensions collapsed
! and arbitrarily distributed
grid3d = ESMF_GridCreateNoPeriDim(coordTypeKind=ESMF_TYPEKIND_R8, &
  minIndex=(/1,1,1/), maxIndex=(/xdim, ydim,zdim/), &
  arbIndexList=localArbIndex,arbIndexCount=localArbIndexCount, &
  name="arb3dgrid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create a 2D arrayspec
call ESMF_ArraySpecSet(arrayspec2D, rank=2, typekind=ESMF_TYPEKIND_R4, &
     rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create a 2D Field using the Grid and the arrayspec
field = ESMF_FieldCreate(grid3d, arrayspec2D, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(field, rank=rank, dimCount=dimCount, &
                   rc=rc)
if (myPet .eq. 0) print *, 'Field rank, dimCount', &
                            rank, dimCount
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! verify that the dimension counts are correct
if (rank .ne. 2) correct = .false.
if (dimCount .ne. 3) correct = .false.
```

22.3.23 Create a Field on an arbitrarily distributed Grid with replicated dimensions & ungridded bounds

The next example is slightly more complicated in that the Field also contains ungridded dimension and its gridded dimension is replicated on the arbitrarily distributed dimension of the Grid.
The same 3D Grid and 2D arrayspec in the previous example are used but a gridToFieldMap argument is supplied to the `ESMF_FieldCreate()` call. The first 2 entries of the map are 0, the last (3rd) entry is 1. The 3rd dimension of the Grid is mapped to the first dimension of the Field, this dimension is then replicated on the arbitrarily distributed dimensions of the Grid. In addition, the Field also has one ungridded dimension. Thus the final dimension count of the Field is 2 in both physics and memory index space.

```
field = ESMF_FieldCreate(grid3d, arrayspec2D,gridToFieldMap=(/0,0,1/), &
        ungriddedLBound=(/1/), ungriddedUBound=(/10/),rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(field, rank=rank, dimCount=dimCount, &
                   rc=rc)
if (myPet .eq. 0) print *, 'Field rank, dimCount', &
                            rank, dimCount
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

if (rank .ne. 2) correct = .false.
if (dimCount .ne. 2) correct = .false.
```

22.3.24 Field regridding

The Field regrid operation moves data between Fields which lie on different Grids. In order to do this the data in the source Field is interpolated to the destination Grid and then put into the destination Field. In ESMF the regrid operation is implemented as a sparse matrix multiply. The `ESMF_FieldRegridStore()` call generates the sparse matrix for the regrid operation. This matrix may be either retrieved in a factor and index raw form, or may be retrieved in the form of a routeHandle which contains an internal representation of the communication and mathermatical operations necessary to perform the regrid. Note `ESMF_FieldRegridStore()` assumes the latitude and longitude coordinates of the grid are in degrees. The routeHandle can then be used in an `ESMF_FieldRegrid()` call to perform the interpolation between the two Fields. The routeHandle depends only on the coordinates in the Grids where the Fields are built, so as long as the coordinates stay the same, the operation can be performed multiple times using the same routeHandle. This is true even if the Field data changes. The same routeHandle may also be used to interpolate between any source and destination Field which lie on the same stagger location and Grid as the original Fields. When it's no longer needed the routeHandle should be destroyed by using `ESMF_FieldRegridRelease()` to free the memory it's using.
There are two options for accessing ESMF regridding functionality: online and offline. Online regridding means that the weights are generated via subroutine calls during the execution of the users code. This is the method described in the following sections. Offline regridding means that the weights are generated by a seperate application from the user code. Please see Section 12 for a description of the offline regridding application and the options it supports.
ESMF currently supports regridding only on a subset of the full range of Grids and Meshes it supports.
In 2D ESMF supports regridding between any combination of the following:

- Structured Grids composed of a single logically rectangular patch
- Unstructured Meshes composed of any combination of triangles and quadralaterals (e.g. rectangles)

In 3D ESMF supports regridding between any combination of the following:

- Structured Grids composed of a single logically rectangular patch
- Unstructured Meshes composed of hexahedrons (e.g. cubes).

Note that regridding involving tetrahedra is currently NOT supported.
In terms of masking, ESMF regrid currently supports masking for Fields built on structured Grids. The user may mask out points in the source Field or destination Field or both. The user also has the option to return an error for unmapped destination points or to ignore them. At this point ESMF does not support extrapolation to destination points outside the unmasked source Field.
ESMF currently supports three options for interpolation: bilinear, patch, and conservative. 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.
Patch (or higher-order) interpolation is the ESMF version of a techique called "patch recovery" commonly used in finite element modeling [5] [10]. 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 squared 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.
First-order conservative interpolation [19] is also available as a regridding method. This method will typically have a larger interpolation error than the previous two methods, but will do a much better job of preserving the value of the integral of data between the source and destination grid. In this method the value across each source cell is treated as a constant. The weights for a particular destination cell, are the area of intersection of each source cell with the destination cell divided by the area of the destination cell. Areas in this case are the great circle areas of the polygons which make up the cells (the cells around each center are defined by the corner coordinates in the grid file). To use this method the user must have created 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`) must contain coordinates describing the outer perimeter of the Grid cells. Currently conservative interpolation is only supported for 2D Grids and Meshes.

		Online	Offline
2D Polygons	Triangles	√	√
	Quadrilaterals	√	√
3D Polygons	Hexahedrons	√	
Regridding	Bilinear	√	√
	Patch	√	√
	Conservative (1st order)	√	√
Masking	Destination	√	√
	Source	√	√
	Unmapped points	√	
Pole Options	Full circle average	√	√
	N-point average	√	√
	Teeth pole	√	√

Table 1: Comparison of the offline vs. online regridding capabilities of ESMF

The following sections give examples of using the regridding functionality.

22.3.25 Precompute a regridding operation between two Fields

To create the sparse matrix regrid operator we call the `ESMF_FieldRegridStore()` routine. In this example we choose the `ESMF_REGRIDMETHOD_BILINEAR` regridding method. Other methods are available and more we will be added in the future. This method creates two meshes, and a Rendezvous decomposition of these meshes is computed. An octree search is performed, followed by a determination of which source cell each destination gridpoint is in. Bilinear weights are then computed locally on each cell. This matrix of weights is, finally, sent back to the destination grid's row decomposition and declared as a sparse matrix. This matrix is embedded in the routeHandle object. Note the coordinates of the source and destination grids upon whilch the source and destination fields are defined should be in degrees.

```
call ESMF_FieldRegridStore(srcField=srcField, dstField=dstField, &
                routeHandle=routeHandle, &
                indices=indices, weights=weights, &
                regridmethod=ESMF_REGRIDMETHOD_BILINEAR, rc=localrc)
```

22.3.26 Apply a regridding operation between a pair of Fields

The `ESMF_FieldRegrid` subroutine calls `ESMF_ArraySparseMatMul` and performs a regrid from source to destination field.

```
call ESMF_FieldRegrid(srcField, dstField, routeHandle, rc=localrc)
```

22.3.27 Release the stored information for a regridding operation

```
call ESMF_FieldRegridRelease(routeHandle, rc=localrc)
```

22.3.28 Precompute a regridding operation using masks

As before, to create the sparse matrix regrid operator we call the `ESMF_FieldRegridStore()` routine. However, in this case we apply masking to the regrid operation. The mask value for each index location in the Grids may be set using the `ESMF_GridAddItem()` call (see Section 27.3.14 and Section 27.3.15). Mask values may be set independantly for the source and destination Grids. If no mask values have been set in a Grid, then it is assumed no masking should be used for that Grid. The `srcMaskValues` parameter allows the user to set the list of values which indicate that a source location should be masked out. The `dstMaskValues` parameter allows the user to set the list of values which indicate that a destination location should be masked out. The absence of one of these parameters indicates that no masking should be used for that Field (e.g no `srcMaskValue` parameter indicates that source masking shouldn't occur). The `unmappedaction` flag may be used with or without masking and indicates what should occur if destination points can not be mapped to a source cell. Here the `ESMF_UNMAPPEDACTION_IGNORE` value indicates that unmapped destination points are to be ignored and no sparse matrix entries should be generated for them.

```
call ESMF_FieldRegridStore(srcField=srcField, srcMaskValues=(/1/),       &
                           dstField=dstField, dstMaskValues=(/1/),       &
                           unmappedaction=ESMF_UNMAPPEDACTION_IGNORE, &
                           routeHandle=routeHandle,                      &
                           indices=indices, weights=weights,           &
                           regridmethod=ESMF_REGRIDMETHOD_BILINEAR,     &
                           rc=localrc)
```

The `ESMF_FieldRegrid` and `ESMF_FieldRegridRelease` calls may then be applied as in the previous example.

22.3.29 Regrid 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@list.woc.noaa.gov).

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 constructuted 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.

22.3.30 Field Regrid Example: Mesh to Mesh

This example demonstrates the regridding process between Fields created on Meshes. First the Meshes are created. This example omits the setup of the arrays describing the Mesh, but please see Section 29.3.1 for examples of this. After creation Fields are constructed on the Meshes, and then ESMF_FieldRegridStore() is called to construct a RouteHandle implementing the regrid operation. Finally, ESMF_FieldRegrid() is called with the Fields and the RouteHandle to do the interpolation between the source Field and destination Field. Note the coordinates of the source and destination Mesh should be in degrees.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create Source Mesh
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Create the Mesh structure.
! For brevity's sake, the code to fill the Mesh creation
! arrays is omitted from this example. However, here
! is a brief description of the arrays:
! srcNodeIds    - the global ids for the src nodes
! srcNodeCoords - the coordinates for the src nodes
! srcNodeOwners - which PET owns each src node
```

```
! srcElemIds    - the global ids of the src elements
! srcElemTypes  - the topological shape of each src element
! srcElemConn   - how to connect the nodes to form the elements
!                 in the source mesh
! Several examples of setting up these arrays can be seen in
! the Mesh Section "Mesh Creation".
srcMesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=srcNodeIds, nodeCoords=srcNodeCoords, &
       nodeOwners=srcNodeOwners, elementIds=srcElemIds,&
       elementTypes=srcElemTypes, elementConn=srcElemConn, rc=rc)



!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create and Fill Source Field
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Set description of source Field
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=rc)

! Create source Field
srcField = ESMF_FieldCreate(srcMesh, arrayspec, &
                      name="source", rc=rc)

! Get source Field data pointer to put data into
call ESMF_FieldGet(srcField, 0, fptr1D,  rc=rc)

! Get number of local nodes to allocate space
! to hold local node coordinates
call ESMF_MeshGet(srcMesh, &
       numOwnedNodes=numOwnedNodes, rc=rc)

! Allocate space to hold local node coordinates
! (spatial dimension of Mesh*number of local nodes)
allocate(ownedNodeCoords(2*numOwnedNodes))

! Get local node coordinates
call ESMF_MeshGet(srcMesh, &
       ownedNodeCoords=ownedNodeCoords, rc=rc)

! Set the source Field to the function 20.0+x+y
do i=1,numOwnedNodes
  ! Get coordinates
  x=ownedNodeCoords(2*i-1)
  y=ownedNodeCoords(2*i)

 ! Set source function
 fptr1D(i) = 20.0+x+y
enddo

! Deallocate local node coordinates
deallocate(ownedNodeCoords)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
! Create Destination Mesh
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Create the Mesh structure.
! For brevity's sake, the code to fill the Mesh creation
! arrays is omitted from this example. However, here
! is a brief description of the arrays:
! dstNodeIds    - the global ids for the dst nodes
! dstNodeCoords - the coordinates for the dst nodes
! dstNodeOwners - which PET owns each dst node
! dstElemIds    - the global ids of the dst elements
! dstElemTypes  - the topological shape of each dst element
! dstElemConn   - how to connect the nodes to form the elements
!                 in the destination mesh
! Several examples of setting up these arrays can be seen in
! the Mesh Section "Mesh Creation".
dstMesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=dstNodeIds, nodeCoords=dstNodeCoords, &
       nodeOwners=dstNodeOwners, elementIds=dstElemIds,&
       elementTypes=dstElemTypes, elementConn=dstElemConn, rc=rc)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create Destination Field
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Set description of source Field
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=rc)

! Create destination Field
dstField = ESMF_FieldCreate(dstMesh, arrayspec, &
                      name="destination", rc=rc)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Do Regrid
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Compute RouteHandle which contains the regrid operation
call ESMF_FieldRegridStore( &
        srcField, &
        dstField=dstField, &
        routeHandle=routeHandle, &
        regridmethod=ESMF_REGRIDMETHOD_BILINEAR, &
        rc=rc)

! Perform Regrid operation moving data from srcField to dstField
call ESMF_FieldRegrid(srcField, dstField, routeHandle, rc=rc)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! dstField now contains the interpolated data.
! If the Meshes don't change, then routeHandle
! may be used repeatedly to interpolate from
! srcField to dstField.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
! User code to use the routeHandle, Fields, and
! Meshes goes here before they are freed below.


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Free the objects created in the example.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Free the RouteHandle
call ESMF_FieldRegridRelease(routeHandle, rc=rc)

! Free the Fields
call ESMF_FieldDestroy(srcField, rc=rc)

call ESMF_FieldDestroy(dstField, rc=rc)

! Free the Meshes
call ESMF_MeshDestroy(dstMesh, rc=rc)

call ESMF_MeshDestroy(srcMesh, rc=rc)
```

22.3.31 Gather Field data onto root PET

User can use `ESMF_FieldGather` interface to gather Field data from multiple PETS onto a single root PET. This interface is overloaded by type, kind, and rank.
Note that the implementation of Scatter and Gather is not seqence index based. If the Field is built on arbitrarily distributed Grid, Mesh, LocStream or XGrid, Gather will not gather data to rootPet from source data points corresponding to the sequence index on the rootPet. Instead Gather will gather a contiguous memory range from source PET to rootPet. The size of the memory range is equal to the number of data elements on the source PET. Vice versa for the Scatter operation. In this case, the user should use `ESMF_FieldRedist` to achieve the same data operation result. For examples how to use `ESMF_FieldRedist` to perform Gather and Scatter, please refer to 22.3.35 and 22.3.34.
In this example, we first create a 2D Field, then use `ESMF_FieldGather` to collect all the data in this Field into a data pointer on PET 0.

```
  ! Get current VM and pet number
  call ESMF_VMGetCurrent(vm, rc=rc)
  if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

  call ESMF_VMGet(vm, localPet=lpe, rc=rc)
  if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

  ! Create a 2D Grid and use this grid to create a Field
  ! farray is the Fortran data array that contains data on each PET.
  grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), &
      name="grid", rc=rc)
  if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

  field = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, rc=localrc)
  if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

```
call ESMF_FieldGet(field, farrayPtr=fptr, rc=localrc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
!---------Initialize pet specific field data----------------
!    1        5         10
! 1  +--------+---------+
!    |        |         |
!    |   0    |    1    |
!    |        |         |
! 10 +--------+---------+
!    |        |         |
!    |   2    |    3    |
!    |        |         |
! 20 +--------+---------+
fptr = lpe

! allocate the Fortran data array on PET 0 to store gathered data
if(lpe .eq. 0) allocate(farrayDst(10,20))
call ESMF_FieldGather(field, farrayDst, rootPet=0, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! check that the values gathered on rootPet are correct
if(lpe .eq. 0) then
   do i = 1, 2
      do j = 1, 2
         if(farrayDst(i, j) .ne. (i-1)+(j-1)*2) localrc=ESMF_FAILURE
         if(farrayDst(i*5, j*10) .ne. (i-1)+(j-1)*2) localrc=ESMF_FAILURE
      enddo
   enddo
  if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
endif

! destroy all objects created in this example to prevent memory leak
call ESMF_FieldDestroy(field, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_GridDestroy(grid, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
if(lpe .eq. 0) deallocate(farrayDst)
```

22.3.32 Scatter Field data from root PET onto its set of joint PETs

User can use `ESMF_FieldScatter` interface to scatter Field data from root PET onto its set of joint PETs. This interface is overloaded by type, kind, and rank.
In this example, we first create a 2D Field, then use `ESMF_FieldScatter` to scatter the data from a data array located on PET 0 onto this Field.

```
! Create a 2D Grid and use this grid to create a Field
! farray is the Fortran data array that contains data on each PET.
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
    regDecomp=(/2,2/), &
    name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

field = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, rc=localrc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

```
! initialize values to be scattered
!    1        5         10
! 1  +--------+---------+
!    |        |         |
!    |   0    |    1    |
!    |        |         |
! 10 +--------+---------+
!    |        |         |
!    |   2    |    3    |
!    |        |         |
! 20 +--------+---------+
if(lpe .eq. 0) then
    allocate(farraySrc(10,20))
    farraySrc(1:5,1:10) = 0
    farraySrc(6:10,1:10) = 1
    farraySrc(1:5,11:20) = 2
    farraySrc(6:10,11:20) = 3
endif

! scatter the data onto individual PETs of the Field
call ESMF_FieldScatter(field, farraySrc, rootPet=0, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(field, localDe=0, farrayPtr=fptr, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! verify that the scattered data is properly distributed
do i = lbound(fptr, 1), ubound(fptr, 1)
    do j = lbound(fptr, 2), ubound(fptr, 2)
        if(fptr(i, j) .ne. lpe) localrc = ESMF_FAILURE
    enddo
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! destroy all objects created in this example to prevent memory leak
call ESMF_FieldDestroy(field, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_GridDestroy(grid, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
if(lpe .eq. 0) deallocate(farraySrc)
```

22.3.33 Redistribute data from source Field to destination Field

User can use ESMF_FieldRedist interface to redistribute data from source Field to destination Field. This interface is overloaded by type and kind; In the version of ESMF_FieldRedist without factor argument, a default value of 1 is used.
In this example, we first create two 1D Fields, a source Field and a destination Field. Then we use ESMF_FieldRedist to redistribute data from source Field to destination Field.

```
! Get current VM and pet number
call ESMF_VMGetCurrent(vm, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

```
call ESMF_VMGet(vm, localPet=localPet, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create grid
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
        regDecomp=(/4/), &
        rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

grid = ESMF_GridCreate(distgrid=distgrid, &
    name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create srcField
! +--------+--------+--------+--------+
!      0        1        2        3            ! value
! 1        4        8        12       16       ! bounds
srcField = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(srcField, farrayPtr=srcfptr, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

srcfptr(:) = localPet

! create dstField
! +--------+--------+--------+--------+
!      0        0        0        0            ! value
! 1        4        8        12       16       ! bounds
dstField = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_FieldGet(dstField, farrayPtr=dstfptr, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

dstfptr(:) = 0

! perform redist
! 1. setup routehandle from source Field to destination Field
call ESMF_FieldRedistStore(srcField, dstField, routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! 2. use precomputed routehandle to redistribute data
call ESMF_FieldRedist(srcfield, dstField, routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! verify redist
call ESMF_FieldGet(dstField, localDe=0, farrayPtr=fptr, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! Verify that the redistributed data in dstField is correct.
! Before the redist op, the dst Field contains all 0.
```

```
! The redist op reset the values to the PE value, verify this is the case.
do i = lbound(fptr, 1), ubound(fptr, 1)
    if(fptr(i) .ne. localPet) localrc = ESMF_FAILURE
enddo
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

Field redistribution can also be performed between weakly congruent Fields. In this case, source and destination Fields can have ungridded dimensions with size different from the Field pair used to compute the routehandle.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=2, rc=rc)
```

Create two fields with ungridded dimensions using the Grid created previously. The new Field pair has matching number of elements. The ungridded dimension is mapped to the first dimension of either Field.

```
srcFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)
```

```
dstFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)
```

Using the previously computed routehandle, weakly congruent Fields can be redistributed.

```
call ESMF_FieldRedist(srcfieldA, dstFieldA, routehandle, rc=rc)
```

```
call ESMF_FieldRedistRelease(routehandle, rc=rc)
```

22.3.34 FieldRedist as a form of scatter involving arbitrary distribution

User can use `ESMF_FieldRedist` interface to redistribute data from source Field to destination Field, where the destination Field is built on an arbitrarily distributed structure, e.g. `ESMF_Mesh`. The underlying mechanism is explained in section 24.2.18.
In this example, we will create 2 one dimensional Fields, the src Field has a regular decomposition and holds all its data on a single PET, in this case PET 0. The destination Field is built on a Mesh which is itself built on an arbitrarily distributed distgrid. Then we use `ESMF_FieldRedist` to redistribute data from source Field to destination Field, similar to a traditional scatter operation.
The src Field only has data on PET 0 where it is sequentially initialized, i.e. 1,2,3...This data will be redistributed (or scattered) from PET 0 to the destination Field arbitrarily distributed on all the PETs.

```
! a one dimensional grid whose elements are all located on PET 0
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/9/), &
    regDecomp=(/1/), &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
grid = ESMF_GridCreate(distgrid=distgrid, &
    indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

srcField = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! initialize the source data
```

```
if (localPet == 0) then
    call ESMF_FieldGet(srcField, farrayPtr=srcfptr, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    do i = 1, 9
        srcfptr(i) = i
    enddo
endif
```

For more information on Mesh creation, user can refer to Mesh examples section or Field creation on Mesh example for more details.

```
! Create Mesh structure
mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=nodeIds, nodeCoords=nodeCoords, &
       nodeOwners=nodeOwners, elementIds=elemIds,&
       elementTypes=elemTypes, elementConn=elemConn, &
       rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

Create the destination Field on the Mesh that is arbitrarily distributed on all the PETs.

```
dstField = ESMF_FieldCreate(mesh, typekind=ESMF_TYPEKIND_I4, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

Perform the redistribution from source Field to destination Field.

```
call ESMF_FieldRedistStore(srcField, dstField, &
        routehandle=routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
call ESMF_FieldRedist(srcField, dstField, routehandle=routehandle, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

We can now verify that the sequentially intialized source data is scattered on to the destination Field. The data has been scattered onto the destination Field with the following distribution.

```
4 elements on PET 0:  1 2 4 5
2 elements on PET 1:  3 6
2 elements on PET 2:  7 8
1 element  on PET 3:  9
```

Because the redistribution is index based, the elements also corresponds to the index space of Mesh in the destination Field.

```
call ESMF_FieldGet(dstField, farrayPtr=dstfptr, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

The scatter operation is successful. Since the routehandle computed with `ESMF_FieldRedistStore` can be reused, user can use the same routehandle to scatter multiple source Fields from a single PET to multiple destination Fields distributed on all PETs. The `gathering` operation is just the opposite of the demonstrated `scattering` operation, where a user would redist from a source Field distributed on multiple PETs to a destination Field that only has data storage on a single PET.
Now it's time to release all the resources.

```
call ESMF_FieldRedistRelease(routehandle=routehandle, rc=rc)
```

22.3.35 FieldRedist as a form of gather involving arbitrary distribution

Similarly, one can use the same approach to gather the data from an arbitrary distribution to a non-arbitrary distribution. This concept is demonstrated by using the previous Fields but the data operation is reversed. This time data is gathered from the Field built on the mesh to the Field that has only data allocation on rootPet.
First a FieldRedist routehandle is created from the Field built on Mesh to the Field that has only data allocation on rootPet.

```
call ESMF_FieldRedistStore(dstField, srcField, routehandle=routehandle, &
     rc=rc)
```

Perform FieldRedist, this will gather the data points from the Field built on mesh to the data pointer on the rootPet (default to 0) stored in the srcField.

```
call ESMF_FieldRedist(dstField, srcField, routehandle=routehandle, rc=rc)
```

Release the routehandle used for the gather operation.

```
call ESMF_FieldRedistRelease(routehandle=routehandle, rc=rc)
```

22.3.36 Sparse matrix multiplication from source Field to destination Field

A user can use `ESMF_FieldSMM()` interface to perform sparse matrix multiplication from source Field to destination Field. This interface is overloaded by type and kind;
In this example, we first create two 1D Fields, a source Field and a destination Field. Then we use `ESMF_FieldSMM` to perform sparse matrix multiplication from source Field to destination Field.
The source and destination Field data are arranged such that each of the 4 PETs has 4 data elements. Moreover, the source Field has all its data elements initialized to a linear function based on local PET number. Then collectively on each PET, a SMM according to the following formula is preformed:
$dstField(i) = i * srcField(i), i = 1...4$

Because source Field data are initialized to a linear function based on local PET number, the formula predicts that the result destination Field data on each PET is 1,2,3,4. This is verified in the example.
Section 24.2.17 provides a detailed discussion of the sparse matrix mulitiplication operation implemented in ESMF.

```
! Get current VM and pet number
call ESMF_VMGetCurrent(vm, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_VMGet(vm, localPet=lpe, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

! create distgrid and grid
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
    regDecomp=(/4/), &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

```
    grid = ESMF_GridCreate(distgrid=distgrid, &
        name="grid", rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    call ESMF_GridGetFieldBounds(grid, localDe=0, totalCount=fa_shape, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! create src\_farray, srcArray, and srcField
    ! +--------+--------+--------+--------+
    !      1        2        3        4            ! value
    ! 1        4        8        12       16       ! bounds
    allocate(src_farray(fa_shape(1)) )
    src_farray = lpe+1
    srcArray = ESMF_ArrayCreate(distgrid, src_farray, &
indexflag=ESMF_INDEX_DELOCAL, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    srcField = ESMF_FieldCreate(grid, srcArray, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! create dst_farray, dstArray, and dstField
    ! +--------+--------+--------+--------+
    !      0        0        0        0            ! value
    ! 1        4        8        12       16       ! bounds
    allocate(dst_farray(fa_shape(1)) )
    dst_farray = 0
    dstArray = ESMF_ArrayCreate(distgrid, dst_farray, &
indexflag=ESMF_INDEX_DELOCAL, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    dstField = ESMF_FieldCreate(grid, dstArray, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! perform sparse matrix multiplication
    ! 1. setup routehandle from source Field to destination Field
    ! initialize factorList and factorIndexList
    allocate(factorList(4))
    allocate(factorIndexList(2,4))
    factorList = (/1,2,3,4/)
    factorIndexList(1,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)
    factorIndexList(2,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)

    call ESMF_FieldSMMStore(srcField, dstField, routehandle, &
        factorList, factorIndexList, rc=localrc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! 2. use precomputed routehandle to perform SMM
    call ESMF_FieldSMM(srcfield, dstField, routehandle, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! verify sparse matrix multiplication
    call ESMF_FieldGet(dstField, localDe=0, farrayPtr=fptr, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! Verify that the result data in dstField is correct.
```

```
! Before the SMM op, the dst Field contains all 0.
! The SMM op reset the values to the index value, verify this is the case.
! +--------+--------+--------+--------+
!  1 2 3 4  2 4 6 8  3 6 9 12  4 8 12 16       ! value
! 1        4        8        12       16       ! bounds
do i = lbound(fptr, 1), ubound(fptr, 1)
    if(fptr(i) /= i*(lpe+1)) rc = ESMF_FAILURE
enddo
```

Field sparse matrix matmul can also be performed between weakly congruent Fields. In this case, source and destination Fields can have ungridded dimensions with size different from the Field pair used to compute the routehandle.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=2, rc=rc)
```

Create two fields with ungridded dimensions using the Grid created previously. The new Field pair has matching number of elements. The ungridded dimension is mapped to the first dimension of either Field.

```
srcFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)


dstFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)
```

Using the previously computed routehandle, weakly congruent Fields can perform sparse matrix matmul.

```
call ESMF_FieldSMM(srcfieldA, dstFieldA, routehandle, rc=rc)


! release route handle
call ESMF_FieldSMMRelease(routehandle, rc=rc)
```

In the following discussion, we demonstrate how to set up a SMM routehandle between a pair of Fields that are different in number of gridded dimensions and the size of those gridded dimensions. The source Field has a 1D decomposition with 16 total elements; the destination Field has a 2D decomposition with 12 total elements. For ease of understanding of the actual matrix calculation, a global indexing scheme is used.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
    indexflag=ESMF_INDEX_GLOBAL, &
    regDecomp=(/4/), &
    rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

grid = ESMF_GridCreate(distgrid=distgrid, &
    indexflag=ESMF_INDEX_GLOBAL, &
    name="grid", rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

call ESMF_GridGetFieldBounds(grid, localDe=0, totalLBound=tlb, &
                   totalUBound=tub, rc=rc)
if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

create 1D src_farray, srcArray, and srcField

```
+  PET0  +  PET1  +  PET2  +  PET3  +
+--------+--------+--------+--------+
     1        2        3        4            ! value
1        4        8        12       16       ! bounds of seq indices
```

```
 allocate(src_farray2(tlb(1):tub(1)) )
 src_farray2 = lpe+1
 srcArray = ESMF_ArrayCreate(distgrid, src_farray2, &
indexflag=ESMF_INDEX_GLOBAL, &
   rc=rc)
 if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
 !print *, lpe, '+', tlb, tub, '+', src_farray2

 srcField = ESMF_FieldCreate(grid, srcArray, rc=rc)
 if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

Create 2D dstField on the following distribution (numbers are the sequence indices):

```
+  PET0  +  PET1  +  PET2  +  PET3  +
+--------+--------+--------+--------+
|        |        |        |        |
|   1    |   4    |   7    |   10   |
|        |        |        |        |
+--------+--------+--------+--------+
|        |        |        |        |
|   2    |   5    |   8    |   11   |
|        |        |        |        |
+--------+--------+--------+--------+
|        |        |        |        |
|   3    |   6    |   9    |   12   |
|        |        |        |        |
+--------+--------+--------+--------+
```

```
 ! Create the destination Grid
 dstGrid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/3,4/), &
   indexflag = ESMF_INDEX_GLOBAL, &
   regDecomp = (/1,4/), &
   rc=rc)
 if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

 dstField = ESMF_FieldCreate(dstGrid, typekind=ESMF_TYPEKIND_R4, &
   indexflag=ESMF_INDEX_GLOBAL, &
   rc=rc)
 if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

Perform sparse matrix multiplication $dst_i = M_{i,j} * src_j$ First setup routehandle from source Field to destination Field with prescribed factorList and factorIndexList.
The sparse matrix is of size 12x16, however only the following entries are filled:

```
M(3,1) = 0.1
M(3,10) = 0.4
```

```
M(8,2) = 0.25
M(8,16) = 0.5
M(12,1) = 0.3
M(12,16) = 0.7
```

By the definition of matrix calculation, the 8th element on PET2 in the dstField equals to 0.25*srcField(2) + 0.5*srcField(16) = 0.25*1+0.5*4=2.25 For simplicity, we will load the factorList and factorIndexList on PET 0 and 1, the SMMStore engine will load balance the parameters on all 4 PETs internally for optimal performance.

```
    if(lpe == 0) then
      allocate(factorList(3), factorIndexList(2,3))
      factorList=(/0.1,0.4,0.25/)
      factorIndexList(1,:)=(/1,10,2/)
      factorIndexList(2,:)=(/3,3,8/)
      call ESMF_FieldSMMStore(srcField, dstField, routehandle=routehandle, &
          factorList=factorList, factorIndexList=factorIndexList, rc=localrc)
      if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    else if(lpe == 1) then
      allocate(factorList(3), factorIndexList(2,3))
      factorList=(/0.5,0.3,0.7/)
      factorIndexList(1,:)=(/16,1,16/)
      factorIndexList(2,:)=(/8,12,12/)
      call ESMF_FieldSMMStore(srcField, dstField, routehandle=routehandle, &
          factorList=factorList, factorIndexList=factorIndexList, rc=localrc)
      if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    else
      call ESMF_FieldSMMStore(srcField, dstField, routehandle=routehandle, &
          rc=localrc)
      if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    endif

    ! 2. use precomputed routehandle to perform SMM
    call ESMF_FieldSMM(srcfield, dstField, routehandle=routehandle, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
```

22.3.37 Field Halo solving a domain decomposed heat transfer problem

`ESMF_FieldHalo()` interface can be used to perform halo update of a Field. This eases communication programming from a user perspective. By definition, user program only needs to update locally owned exclusive region in each domain, then call FieldHalo to communicate the values in the halo region from/to neighboring domain elements. In this example, we solve a 1D heat transfer problem: $u_t = \alpha^2 u_{xx}$ with the initial condition $u(0, x) = 20$ and boundary conditions $u(t, 0) = 10, u(t, 1) = 40$. The temperature field u is represented by a `ESMF_Field`. A finite difference explicit time steping scheme is employed. During each time step, FieldHalo update is called to communicate values in the halo region to neighboring domain elements. The steady state (as $t \to \infty$) solution is a linear temperature profile along x. The numerical solution is an approximation of the steady state solution. It can be verified to represent a linear temperature profile.
Section 24.2.14 provides a discussion of the halo operation implemented in `ESMF_Array`.

```
! create 1D distgrid and grid decomposed according to the following diagram:
! +------------+   +----------------+   +---------------+   +--------------+
! |   DE 0  |  |   |  |   DE 1   |  |   |  |   DE 2  |  |   |  |   DE 3    |
! |  1 x 16 |  |   |  |  1 x 16  |  |   |  |  1 x 16 |  |   |  |  1 x 16   |
! |         | 1|<->|1 |          | 1|<->|1 |         | 1|<->|1 |           |
! |         |  |   |  |          |  |   |  |         |  |   |  |           |
```

```
! +------------+   +---------------+   +--------------+   +--------------+
    distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/npx/), &
        regDecomp=(/4/), rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    ! set up initial condition and boundary conditions of the
    ! temperature Field
    if(lpe == 0) then
        allocate(fptr(17), tmp_farray(17))
        fptr = 20.
        fptr(1) = 10.
        tmp_farray(1) = 10.
        startx = 2
        endx = 16

        field = ESMF_FieldCreate(grid, fptr, totalUWidth=(/1/), &
name="temperature", rc=rc)
        if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    else if(lpe == 3) then
        allocate(fptr(17), tmp_farray(17))
        fptr = 20.
        fptr(17) = 40.
        tmp_farray(17) = 40.
        startx = 2
        endx = 16

        field = ESMF_FieldCreate(grid, fptr, totalLWidth=(/1/), &
name="temperature", rc=rc)
        if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    else
        allocate(fptr(18), tmp_farray(18))
        fptr = 20.
        startx = 2
        endx = 17

        field = ESMF_FieldCreate(grid, fptr, &
            totalLWidth=(/1/), totalUWidth=(/1/), name="temperature", rc=rc)
        if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    endif

    ! compute the halo update routehandle of the decomposed temperature Field
    call ESMF_FieldHaloStore(field, routehandle=routehandle, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

    dt = 0.01
    dx = 1./npx
    alpha = 0.1

    ! Employ explicit time steping
    ! Solution converges after about 9000 steps based on apriori knowledge.
    ! The result is a linear temperature profile stored in field.
    do iter = 1, 9000
```

```
 ! only elements in the exclusive region are updated locally
 ! in each domain
 do i = startx, endx
   tmp_farray(i) = &
   fptr(i)+alpha*alpha*dt/dx/dx*(fptr(i+1)-2.*fptr(i)+fptr(i-1))
  enddo
  fptr = tmp_farray
 ! call halo update to communicate the values in the halo region to
 ! neighboring domains
 call ESMF_FieldHalo(field, routehandle=routehandle, rc=rc)
 if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
enddo

! release the halo routehandle
call ESMF_FieldHaloRelease(routehandle, rc=rc)
```

22.4 Restrictions and Future Work

1. **CAUTION:** It depends on the specific entry point of `ESMF_FieldCreate()` used during Field creation, which Fortran operations are supported on the Fortran array pointer `farrayPtr`, returned by `ESMF_FieldGet()`. Only if the `ESMF_FieldCreate()` *from pointer* variant was used, will the returned `farrayPtr` variable contain the original bounds information, and be suitable for the Fortran `deallocate()` call. This limitation is a direct consequence of the Fortran 95 standard relating to the passing of array arguments.

2. **No mathematical operators.** The Fields class does not currently support advanced operations on fields, such as differential or other mathematical operators.

3. **No vector Fields.** ESMF does not currently support storage of multiple vector Field components in the same Field component, although that support is planned. At this time users need to create a separate Field object to represent each vector component.

22.5 Design and Implementation Notes

1. Some methods which have a Field interface are actually implemented at the underlying Grid or Array level; they are inherited by the Field class. This allows the user API (Application Programming Interface) to present functions at the level which is most consistent to the application without restricting where inside the ESMF the actual implementation is done.

2. The Field class is implemented in Fortran, and as such is defined inside the framework by a Field derived type and a set of subprograms (functions and subroutines) which operate on that derived type. The Field class itself is very thin; it is a container class which groups a Grid and an Array object together.

3. Fields follow the framework-wide convention of the *unison* creation and operation rule: All PETs which are part of the currently executing VM must create the same Fields at the same point in their execution. Since an early user request was that global object creation not impose the overhead of a barrier or synchronization point, Field creation does no inter-PET communication. For this to work, each PET must query the total number of PETs in this VM, and which local PET number it is. It can then compute which DE(s) are part of the local decomposition, and any global information can be computed in unison by all PETs independently of the others. In this way the overhead of communication is avoided, at the cost of more difficulty in diagnosing program bugs which result from not all PETs executing the same create calls.

4. Related to the item above, the user request to not impose inter-PET communication at object creation time means that requirement FLD 1.5.1, that all Fields will have unique names, and if not specified, the framework will generate a unique name for it, is difficult or impossible to support. A part of this requirement has been implememted; a unique object counter is maintained in the Base object class, and if a name is not given at create time a name such as "Field003" is generated which is guarenteed to not be repeated by the framework. However,

it is impossible to error check that the user has not replicated a name, and it is possible under certain conditions that if not all PETs have created the same number of objects, that the counters on different PETs may not stay synchronized. This remains an open issue.

22.6 Class API

22.6.1 ESMF_FieldAssignment(=) - Field assignment

INTERFACE:

```
interface assignment(=)
field1 = field2
```

ARGUMENTS:

```
type(ESMF_Field) :: field1
type(ESMF_Field) :: field2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign field1 as an alias to the same ESMF Field object in memory as field2. If field2 is invalid, then field1 will be equally invalid after the assignment.
The arguments are:

field1 The `ESMF_Field` object on the left hand side of the assignment.

field2 The `ESMF_Field` object on the right hand side of the assignment.

22.6.2 ESMF_FieldOperator(==) - Field equality operator

INTERFACE:

```
interface operator(==)
if (field1 == field2) then ... endif
OR
result = (field1 == field2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field1
type(ESMF_Field), intent(in) :: field2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether field1 and field2 are valid aliases to the same ESMF Field object in memory. For a more general comparison of two ESMF Fields, going beyond the simple alias test, the ESMF_FieldMatch() function (not yet implemented) must be used.
The arguments are:

field1 The `ESMF_Field` object on the left hand side of the equality operation.

field2 The `ESMF_Field` object on the right hand side of the equality operation.

22.6.3 ESMF_FieldOperator(/=) - Field not equal operator

INTERFACE:

```
interface operator(/=)
if (field1 /= field2) then ... endif
OR
result = (field1 /= field2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field1
type(ESMF_Field), intent(in) :: field2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether field1 and field2 are *not* valid aliases to the same ESMF Field object in memory. For a more general comparison of two ESMF Fields, going beyond the simple alias test, the ESMF_FieldMatch() function (not yet implemented) must be used.
The arguments are:

field1 The `ESMF_Field` object on the left hand side of the non-equality operation.

field2 The `ESMF_Field` object on the right hand side of the non-equality operation.

22.6.4 ESMF_FieldCreate - Create a Field from Grid and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridTKR(grid, typekind, &
  indexflag, staggerloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridTKR
```

ARGUMENTS:

```
    type(ESMF_Grid), intent(in) :: grid
    type(ESMF_TypeKind_Flag), intent(in) :: typekind
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Index_Flag), intent(in), optional :: indexflag
    type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
    integer, intent(in), optional :: gridToFieldMap(:)
    integer, intent(in), optional :: ungriddedLBound(:)
    integer, intent(in), optional :: ungriddedUBound(:)
    integer, intent(in), optional :: totalLWidth(:)
    integer, intent(in), optional :: totalUWidth(:)
    character (len=*), intent(in), optional :: name
    integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 22.3.4.
The arguments are:

grid `ESMF_Grid` object.

typekind The typekind of the Field.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section 9.24 for a list of valid indexflag options.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[name] Field name.

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

22.6.5 ESMF_FieldCreate - Create a Field from Grid and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridArraySpec(grid, arrayspec, &
  indexflag, staggerloc, gridToFieldMap, ungriddedLBound, &
  ungriddedUBound, totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridArraySpec
```

ARGUMENTS:

```
     type(ESMF_Grid), intent(in) :: grid
     type(ESMF_ArraySpec), intent(in) :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Index_Flag), intent(in), optional :: indexflag
     type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
     integer, intent(in), optional :: gridToFieldMap(:)
     integer, intent(in), optional :: ungriddedLBound(:)
     integer, intent(in), optional :: ungriddedUBound(:)
     integer, intent(in), optional :: totalLWidth(:)
     integer, intent(in), optional :: totalUWidth(:)
     character (len=*), intent(in), optional :: name
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 22.3.5.
The arguments are:

grid `ESMF_Grid` object.

arrayspec Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section 9.24 for a list of valid indexflag options.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[name] Field name.

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

22.6.6 ESMF_FieldCreate - Create a Field from Grid and Array

INTERFACE:

```
   ! Private name; call using ESMF_FieldCreate()
   function ESMF_FieldCreateGridArray(grid, array, datacopyflag, &
     staggerloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, totalLWidth, &
     totalUWidth, name, rc)
```

RETURN VALUE:

```
     type(ESMF_Field) :: ESMF_FieldCreateGridArray
```

ARGUMENTS:

```
     type(ESMF_Grid), intent(in) :: grid
     type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
     type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
     integer, intent(in), optional :: gridToFieldMap(:)
     integer, intent(in), optional :: ungriddedLBound(:)
     integer, intent(in), optional :: ungriddedUBound(:)
     integer, intent(in), optional :: totalLWidth(:)
     integer, intent(in), optional :: totalUWidth(:)
     character (len = *), intent(in), optional :: name
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 22.3.6.
The arguments are:

grid `ESMF_Grid` object.

array `ESMF_Array` object.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[name] Field name.

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

22.6.7 ESMF_FieldCreate - Create a Field from Grid and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridData<rank><type><kind>(grid, &
farray, indexflag, datacopyflag, staggerloc, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, &
totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridData<rank><type><kind>
```

ARGUMENTS:

```
  type(ESMF_Grid), intent(in) :: grid
  <type> (ESMF_KIND_<kind>), intent(in) target :: farray(<rank>)
  type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  character (len=*), intent(in), optional :: name
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Field` from a fortran data array and `ESMF_Grid`. The fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed. For examples and associated documentations using this method see section 22.3.10, 22.3.12, 22.3.13, 22.3.14, and 22.3.9.
The arguments are:

grid `ESMF_Grid` object.

farray Native fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the `farray` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the farray rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farray dimensions less the total (distributed + undistributed) dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the farray. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the farray. That is, for each gridded dimension the farray size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the farray. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the farray. That is, for each gridded dimension the farray size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[name] Field name.

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

22.6.8 ESMF_FieldCreate - Create a Field from Grid and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridDataPtr<rank><type><kind>(grid, &
farrayPtr, datacopyflag, staggerloc, gridToFieldMap, &
totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridDataPtr<rank><type><kind>
```

ARGUMENTS:

```
   type(ESMF_Grid), intent(in) :: grid
   <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(in), optional :: totalLWidth(:)
   integer, intent(in), optional :: totalUWidth(:)
   character (len=*), intent(in), optional :: name
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an ESMF_Field from a fortran data pointer and ESMF_Grid. The fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
For examples and associated documentations using this method see section 22.3.11, 22.3.12, 22.3.13, 22.3.14, and 22.3.9.
The arguments are:

grid ESMF_Grid object.

farrayPtr Native fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the farrayPtr. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the farrayPtr. That is, for each gridded dimension the farrayPtr size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the farrayPtr. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the farrayPtr.

That is, for each gridded dimension the `farrayPtr` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[name] Field name.

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

22.6.9 ESMF_FieldCreate - Create a Field from LocStream and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSTKR(locstream, typekind, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSTKR
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
type(ESMF_TypeKind_Flag), intent(in) :: typekind
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 22.3.15.
The arguments are:

locstream `ESMF_LocStream` object.

typekind The typekind of the Field.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[name] Field name.

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

22.6.10 ESMF_FieldCreate - Create a Field from LocStream and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSArraySpec(locstream, arrayspec, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSArraySpec
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
type(ESMF_ArraySpec), intent(in) :: arrayspec
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 22.3.16.
The arguments are:

locstream ESMF_LocStream object.

arrayspec Data type and kind specification.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

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

22.6.11 ESMF_FieldCreate - Create a Field from LocStream and Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSArray(locstream, array, &
  datacopyflag, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSArray
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
type(ESMF_Array), intent(in) :: array
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len = *), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 22.3.6.
The arguments are:

locstream `ESMF_LocStream` object.

array `ESMF_Array` object.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

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

22.6.12 ESMF_FieldCreate - Create a Field from LocStream and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSData<rank><type><kind>(locstream, farray, &
indexflag, datacopyflag, gridToFieldMap, ungriddedLBound, &
ungriddedUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSData<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
<type> (ESMF_KIND_<kind>), intent(in), target:: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` from a fortran data array and `ESMF_LocStream`. The fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

locstream `ESMF_LocStream` object.

farray Native fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the farray or reference directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the locstream's dimCount. The list elements map each dimension of the locstream to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the locstream's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farray rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farray dimensions less the total (distributed + undistributed) dimensions in the locstream. Unlocstreamded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[name] Field name.

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

22.6.13 ESMF_FieldCreate - Create a Field from LocStream and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSDataPtr<rank><type><kind>(locstream, &
farrayPtr, datacopyflag, gridToFieldMap, &
name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
```

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

DESCRIPTION:

Create an ESMF_Field from a fortran data pointer and ESMF_LocStream. The fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

locstream ESMF_LocStream object.

farrayPtr Native fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the locstream's dimCount. The list elements map each dimension of the locstream to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the locstream's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the locstream. Unlocstreamded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[name] Field name.

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

22.6.14 ESMF_FieldCreate - Create a Field from Mesh and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshTKR(mesh, typekind, meshloc, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshTKR
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
type(ESMF_TypeKind_Flag), intent(in) :: typekind
type(ESMF_MeshLoc), intent(in), optional :: meshloc
```

```
    integer, intent(in), optional :: gridToFieldMap(:)
    integer, intent(in), optional :: ungriddedLBound(:)
    integer, intent(in), optional :: ungriddedUBound(:)
    character (len=*), intent(in), optional :: name
    integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 22.3.4.
The arguments are:

mesh ESMF_Mesh object.

typekind The typekind of the Field.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[name] Field name.

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

22.6.15 ESMF_FieldCreate - Create a Field from Mesh and ArraySpec

INTERFACE:

```
  ! Private name; call using ESMF_FieldCreate()
  function ESMF_FieldCreateMeshArraySpec(mesh, arrayspec, &
    meshloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
    name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshArraySpec
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
type(ESMF_ArraySpec), intent(in) :: arrayspec
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 22.3.5.
The arguments are:

mesh `ESMF_Mesh` object.

arrayspec Data type and kind specification.

[meshloc] Which part of the mesh to build the Field on. Can be set to either `ESMF_MESHLOC_NODE` or `ESMF_MESHLOC_ELEMENT`. If not set, defaults to `ESMF_MESHLOC_NODE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

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

22.6.16 ESMF_FieldCreate - Create a Field from Mesh and Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshArray(mesh, array, meshloc, &
  datacopyflag, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
  type(ESMF_Field) :: ESMF_FieldCreateMeshArray
```

ARGUMENTS:

```
  type(ESMF_Mesh), intent(in) :: mesh
  type(ESMF_Array), intent(in) :: array
  type(ESMF_MeshLoc), intent(in), optional :: meshloc
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  character (len = *), intent(in), optional :: name
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 22.3.6.
The arguments are:

mesh `ESMF_Mesh` object.

array `ESMF_Array` object.

[meshloc] Which part of the mesh to build the Field on. Can be set to either `ESMF_MESHLOC_NODE` or `ESMF_MESHLOC_ELEMENT`. If not set, defaults to `ESMF_MESHLOC_NODE`.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

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

22.6.17 ESMF_FieldCreate - Create a Field from Mesh and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshData<rank><type><kind>(mesh, &
farray, meshloc, indexflag, datacopyflag, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshData<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_MeshLoc), intent(in), optional :: meshloc
type(ESMF_Index_Flag), intent(in) :: indexflag
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field from a fortran data array and ESMF_Mesh. The fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

mesh ESMF_Mesh object.

farray Native fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farray.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the farray or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the mesh's dimCount. The list elements map each dimension of the mesh to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the mesh's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farray rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the `field` are the total `farray` dimensions less the total (distributed + undistributed) dimensions in the `mesh`. Unmeshded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the `ESMF_ArrayRedist()` operation. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[name] Field name.

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

22.6.18 ESMF_FieldCreate - Create a Field from Mesh and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshDataPtr<rank><type><kind>(mesh, &
farrayPtr, meshloc, datacopyflag, gridToFieldMap, &
name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
type(ESMF_MeshLoc), intent(in), optional :: meshloc
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` from a fortran data pointer and `ESMF_Mesh`. The fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

mesh ESMF_Mesh object.

farrayPtr Native fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the mesh's dimCount. The list elements map each dimension of the mesh to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the mesh's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the mesh. Unmeshded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[name] Field name.

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

22.6.19 ESMF_FieldCreate - Create a Field from XGrid and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGTKR(xgrid, xgridside, gridindex, typekind, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGTKR
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
type(ESMF_TypeKind_Flag), intent(in) :: typekind
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 22.3.15.
The arguments are:

xgrid ESMF_XGrid object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

typekind The typekind of the Field.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[name] Field name.

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

22.6.20 ESMF_FieldCreate - Create a Field from XGrid and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGArraySpec(xgrid, xgridside, gridindex, &
  arrayspec, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGArraySpec
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridSide
integer, intent(in), optional :: gridIndex
type(ESMF_ArraySpec), intent(in) :: arrayspec
integer, intent(in), optional :: gridToFieldMap(:)
```

```
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 22.3.16.
The arguments are:

xgrid `ESMF_XGrid` object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

arrayspec Data type and kind specification.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

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

22.6.21 ESMF_FieldCreate - Create a Field from XGrid and Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGArray(xgrid, xgridside, gridindex, array, &
  datacopyflag, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGArray
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
type(ESMF_Array), intent(in) :: array
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len = *), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 22.3.6.
The arguments are:

xgrid `ESMF_XGrid` object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

array `ESMF_Array` object.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

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

22.6.22 ESMF_FieldCreate - Create a Field from XGrid and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGData<rank><type><kind>(xgrid, &
xgridside, gridindex, farray, indexflag, datacopyflag, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, name,&
rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGData<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field from a fortran data array and ESMF_Xgrid. The fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

xgrid ESMF_XGrid object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

farray Native fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the farray or reference directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the `xgrid`'s dimCount. The list elements map each dimension of the `xgrid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `xgrid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `farray` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `farray` dimensions less the total (distributed + undistributed) dimensions in the `xgrid`. Unxgridded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the `ESMF_ArrayRedist()` operation. If the Field dimCount is less than the Xgrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Xgrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than xgrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than xgrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[name] Field name.

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

22.6.23 ESMF_FieldCreate - Create a Field from XGrid and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGDataPtr<rank><type><kind>(xgrid, xgridside, &
gridindex, farrayPtr, datacopyflag, gridToFieldMap, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field from a fortran data pointer and ESMF_Xgrid. The fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

xgrid ESMF_XGrid object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

farrayPtr Native fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the xgrid's dimCount. The list elements map each dimension of the xgrid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the xgrid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the xgrid. Unxgridded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Xgrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Xgrid dimension will be replicating the Field across the DEs along this direction.

[name] Field name.

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

22.6.24 ESMF_FieldDestroy - Release resources associated with a Field

INTERFACE:

```
   subroutine ESMF_FieldDestroy(field, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Releases resources associated with the ESMF_Field. If an ESMF_Grid is associated with field, it will not be released. If field is not released with this call, it will be released by the automatic garbage collection facility in the scope of the Component that created field.
The arguments are:

field ESMF_Field object.

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

22.6.25 ESMF_FieldEmptyComplete - Complete a Field from arrayspec

INTERFACE:

```
   ! Private name; call using ESMF_FieldEmptyComplete()
 subroutine ESMF_FieldEmptyCompAS(field, arrayspec, &
  indexflag, gridToFieldMap, &
  ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_ArraySpec), intent(in) :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_Index_Flag), intent(in), optional :: indexflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Complete an ESMF_Field and allocate space internally for an ESMF_Array based on arrayspec. The input ESMF_Field must have a status of ESMF_FIELDSTATUS_GRIDSET. After this call the completed ESMF_Field has a status of ESMF_FIELDSTATUS_COMPLETE.
The arguments are:

field The input ESMF_Field with a status of ESMF_FIELDSTATUS_GRIDSET.

arrayspec Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section 9.24 for a list of valid indexflag options.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

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

22.6.26 ESMF_FieldEmptyComplete - Complete a Field from typekind

INTERFACE:

```
   ! Private name; call using ESMF_FieldEmptyComplete()
 subroutine ESMF_FieldEmptyCompTK(field, typekind, &
  indexflag, gridToFieldMap, &
  ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_TypeKind_Flag), intent(in) :: typekind
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_Index_Flag), intent(in), optional :: indexflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Complete an ESMF_Field and allocate space internally for an ESMF_Array based on typekind. The input ESMF_Field must have a status of ESMF_FIELDSTATUS_GRIDSET. After this call the completed ESMF_Field has a status of ESMF_FIELDSTATUS_COMPLETE.
For an example and associated documentation using this method see section 22.3.7.
The arguments are:

field The input ESMF_Field with a status of ESMF_FIELDSTATUS_GRIDSET.

typekind Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section 9.24 for a list of valid indexflag options.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

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

22.6.27 ESMF_FieldEmptyComplete - Complete a Field from Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyComp<rank><type><kind>(field, &
farray, indexflag, datacopyflag, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   <type> (ESMF_KIND_<kind>),intent(in), target :: farray(<rank>)
   type(ESMF_Index_Flag), intent(in) :: indexflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(in), optional :: ungriddedLBound(:)
   integer, intent(in), optional :: ungriddedUBound(:)
   integer, intent(in), optional :: totalLWidth(:)
   integer, intent(in), optional :: totalUWidth(:)
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Complete an ESMF_Field and allocate space internally for an ESMF_Array based on typekind. The input ESMF_Field must have a status of ESMF_FIELDSTATUS_GRIDSET. After this call the completed ESMF_Field has a status of ESMF_FIELDSTATUS_COMPLETE.
The fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.
For an example and associated documentation using this method see section 22.3.8.
The arguments are:

field The input ESMF_Field with a status of ESMF_FIELDSTATUS_GRIDSET. The ESMF_Field will have the same dimension (dimCount) as the rank of the farray.

farray Native fortran data array to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the farray or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farray dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

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

22.6.28 ESMF_FieldEmptyComplete - Complete a Field from Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompPtr<rank><type><kind>(field, &
farrayPtr, datacopyflag, gridToFieldMap, &
totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Complete an `ESMF_Field` and allocate space internally for an `ESMF_Array` based on typekind. The input `ESMF_Field` must have a status of `ESMF_FIELDSTATUS_GRIDSET`. After this call the completed `ESMF_Field` has a status of `ESMF_FIELDSTATUS_COMPLETE`.

The fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

field The input `ESMF_Field` with a status of `ESMF_FIELDSTATUS_GRIDSET`. The `ESMF_Field` will have the same dimension (dimCount) as the rank of the `farrayPtr`.

farrayPtr Native fortran data pointer to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[datacopyflag] Indicates whether to copy the `farrayPtr` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farrayPtr` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

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

22.6.29 ESMF_FieldEmptyComplete - Complete a Field from Grid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompGrid<rank><type><kind>(field, grid, &
farray, indexflag, datacopyflag, staggerloc, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_Grid), intent(in) :: grid
  <type> (ESMF_KIND_<kind>), intent(in), target:: farray(<rank>)
  type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
```

```
type(ESMF_STAGGERLOC), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.
The fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farray`.

grid The `ESMF_Grid` object to complete the Field.

farray Native fortran data array to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the `farray` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farray` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

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

22.6.30 ESMF_FieldEmptyComplete - Complete a Field from Grid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompGridPtr<rank><type><kind>(field, grid, &
farrayPtr, datacopyflag, staggerloc, gridToFieldMap, &
totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_Grid), intent(in) :: grid
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  type(ESMF_STAGGERLOC), intent(in), optional :: staggerloc
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.
The fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The fortran data pointer inside `ESMF_Field` can be queried and deallocated when
The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farrayPtr`.

grid The `ESMF_Grid` object to complete the Field.

farrayPtr Native fortran data pointer to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[datacopyflag] Indicates whether to copy the `farrayPtr` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farrayPtr dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

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

22.6.31 ESMF_FieldEmptyComplete - Complete a Field from LocStream started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompLS<rank><type><kind>(field, locstream, &
farray, indexflag, datacopyflag, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_LocStream), intent(in) :: locstream
  <type> (ESMF_KIND_<kind>), intent(in), target:: farray(<rank>)
  type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an ESMF_Field allocated with the ESMF_FieldEmptyCreate() call.
The fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

field The ESMF_Field object to be completed and committed in this call. The field will have the same dimension (dimCount) as the rank of the farray.

locstream The ESMF_LocStream object to complete the Field.

farray Native fortran data array to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the farray or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the locstream's dimCount. The list elements map each dimension of the locstream to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the locstream's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farray dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

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

22.6.32 ESMF_FieldEmptyComplete - Complete a Field from LocStream started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompLSPtr<rank><type><kind>(field, locstream, &
farrayPtr, datacopyflag, gridToFieldMap, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_LocStream), intent(in) :: locstream
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.
The fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farrayPtr`.

locstream The `ESMF_LocStream` object to complete the Field.

farrayPtr Native fortran data pointer to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[datacopyflag] Indicates whether to copy the `farrayPtr` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `locstream`'s dimCount. The list elements map each dimension of the `locstream` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `locstream`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farrayPtr` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

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

22.6.33 ESMF_FieldEmptyComplete - Complete a Field from Mesh started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompMesh<rank><type><kind>(field, mesh, &
farray, meshloc, indexflag, datacopyflag, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_Mesh), intent(in) :: mesh
  <type> (ESMF_KIND_<kind>), intent(in), target:: farray(<rank>)
  type(ESMF_MeshLoc), intent(in), optional :: meshloc
  type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.
The fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farray`.

mesh The `ESMF_Mesh` object to complete the Field.

farray Native fortran data array to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farray`.

[location] Which part of the mesh to build the Field on. Can be set to either `ESMF_MESHLOC_NODE` or `ESMF_MESHLOC_ELEMENT`. If not set, defaults to `ESMF_MESHLOC_NODE`.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the `farray` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `mesh`'s dimCount. The list elements map each dimension of the `mesh` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `mesh`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farray` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than Mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than Mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

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

22.6.34 ESMF_FieldEmptyComplete - Complete a Field from Mesh started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompMeshPtr<rank><type><kind>(field, mesh, &
farrayPtr, meshloc, datacopyflag, gridToFieldMap, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Mesh), intent(in) :: mesh
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_MeshLoc), intent(in), optional :: meshloc
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.
The fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farrayPtr`.

mesh The `ESMF_Mesh` object to complete the Field.

farrayPtr Native fortran data pointer to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[location] Which part of the mesh to build the Field on. Can be set to either `ESMF_MESHLOC_NODE` or `ESMF_MESHLOC_ELEMENT`. If not set, defaults to `ESMF_MESHLOC_NODE`.

[datacopyflag] Indicates whether to copy the `farrayPtr` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `mesh`'s dimCount. The list elements map each dimension of the `mesh` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `mesh`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farrayPtr` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

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

22.6.35 ESMF_FieldEmptyComplete - Complete a Field from XGrid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompXG<rank><type><kind>(field, xgrid, &
xgridside, gridindex, &
farray, indexflag, datacopyflag, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_XGrid), intent(in) :: xgrid
   type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
   integer, intent(in), optional :: gridindex
   <type> (ESMF_KIND_<kind>), intent(in), target:: farray(<rank>)
   type(ESMF_Index_Flag), intent(in) :: indexflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(in), optional :: ungriddedLBound(:)
   integer, intent(in), optional :: ungriddedUBound(:)
   integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.
The fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.
The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farray`.

xgrid The `ESMF_XGrid` object to complete the Field.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

farray Native fortran data array to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the `farray` or reference it directly. For valid values see 9.12. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `xgrid`'s dimCount. The list elements map each dimension of the `xgrid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `xgrid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farray` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than XGrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than XGrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

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

22.6.36 ESMF_FieldEmptyComplete - Complete a Field from XGrid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompXGPtr<rank><type><kind>(field, xgrid, &
xgridside, gridindex, &
farrayPtr, indexflag, datacopyflag, gridToFieldMap, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_XGrid), intent(in) :: xgrid
   type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
   integer, intent(in), optional :: gridindex
   <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an ESMF_Field allocated with the ESMF_FieldEmptyCreate() call.
The fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the fortran data pointer in this case. This gives user more flexibility over memory management.
The arguments are:

field The ESMF_Field object to be completed and committed in this call. The field will have the same dimension (dimCount) as the rank of the farrayPtr.

xgrid The ESMF_XGrid object to complete the Field.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

farrayPtr Native fortran data pointer to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Indicates whether to copy the farrayPtr or reference it directly. For valid values see 9.12. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the xgrid's dimCount. The list elements map each dimension of the xgrid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the xgrid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farrayPtr dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

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

22.6.37 ESMF_FieldEmptyCreate - Create an empty Field

INTERFACE:

```
   function ESMF_FieldEmptyCreate(name, rc)
```

RETURN VALUE:

```
     type(ESMF_Field) :: ESMF_FieldEmptyCreate
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This version of ESMF_FieldCreate builds an empty ESMF_Field and depends on later calls to add an ESMF_Grid and ESMF_Array to it. The empty ESMF_Field can be completed in one more step or two more steps by the ESMF_FieldEmptySet and ESMF_FieldEmptyComplete methods. Attributes can be added to an empty Field object. For an example and associated documentation using this method see section 22.3.8 and 22.3.7. The arguments are:

[name] Field name.

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

22.6.38 ESMF_FieldEmptySet - Set a Grid in an empty Field

INTERFACE:

```
   ! Private name; call using ESMF_FieldEmptySet()
   subroutine ESMF_FieldEmptySetGrid(field, grid, StaggerLoc, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_Grid), intent(in) :: grid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_STAGGERLOC), intent(in), optional :: StaggerLoc
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Set a grid and an optional staggerloc (default to center stagger ESMF_STAGGERLOC_CENTER) in an empty ESMF_Field. The ESMF_Field must be empty for this to succeed. After this operation, the ESMF_Field contains the ESMF_Grid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.
For an example and associated documentation using this method see section 22.3.7.
The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_Grid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

grid ESMF_Grid to be set in the ESMF_Field.

[StaggerLoc] Stagger location of data in grid cells. For valid predefined values see section 27.2.7. To create a custom stagger location see section 27.3.22. The default value is ESMF_STAGGERLOC_CENTER.

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

22.6.39 ESMF_FieldEmptySet - Set a Mesh in an empty Field

INTERFACE:

```
! Private name; call using ESMF_FieldEmptySet()
subroutine ESMF_FieldEmptySetMesh(field, mesh, meshloc, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_Mesh), intent(in) :: mesh
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_MeshLoc), intent(in), optional :: meshloc
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Set a mesh and an optional meshloc (default to center stagger ESMF_MESHLOC_NODE) in an empty ESMF_Field. The ESMF_Field must be empty for this to succeed. After this operation, the ESMF_Field contains the ESMF_Mesh internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.
The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_Mesh internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

mesh ESMF_Mesh to be set in the ESMF_Field.

[location] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

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

22.6.40 ESMF_FieldEmptySet - Set a LocStream in an empty Field

INTERFACE:

```
! Private name; call using ESMF_FieldEmptySet()
subroutine ESMF_FieldEmptySetLocStream(field, locstream, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_LocStream), intent(in) :: locstream
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer, intent(out), optional :: rc
```

DESCRIPTION:

Set a ESMF_LocStream in an empty ESMF_Field. The ESMF_Field must be empty for this to succeed. After this operation, the ESMF_Field contains the ESMF_LocStream internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.
The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_LocStream internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

locstream ESMF_LocStream to be set in the ESMF_Field.

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

22.6.41 ESMF_FieldEmptySet - Set an XGrid in an empty Field

INTERFACE:

```
   ! Private name; call using ESMF_FieldEmptySet()
   subroutine ESMF_FieldEmptySetXGrid(field, xgrid, xgridside, gridindex, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_XGrid), intent(in) :: xgrid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
   integer, intent(in), optional :: gridindex
   integer, intent(out), optional :: rc
```

DESCRIPTION:

Set a xgrid and optional xgridside (default to balanced side ESMF_XGRIDSIDE_Balanced) and gridindex (default to 1) in an empty ESMF_Field. The ESMF_Field must be empty for this to succeed. After this operation, the ESMF_Field contains the ESMF_XGrid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.
The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_XGrid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

xgrid ESMF_XGrid to be set in the ESMF_Field.

[xgridside] Side of XGrid to retrieve a DistGrid. For valid predefined values see section 30.2.1. The default value is ESMF_XGRIDSIDE_BALANCED.

[gridindex] Index to specify which DistGrid when on side A or side B. The default value is 1.

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

22.6.42 ESMF_FieldGet - Return info associated with a Field

INTERFACE:

```
   ! Private name; call using ESMF_FieldGet()
   subroutine ESMF_FieldGetDefault(field, arrayspec, &
     status, geomtype, grid, mesh, locstream, xgrid, array, &
     typekind, dimCount, rank, staggerloc, meshloc, xgridside, &
     gridindex, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
     totalLWidth, totalUWidth, localDeCount, name, rc)
```

ARGUMENTS:

```
     type(ESMF_Field), intent(in) :: field
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_ArraySpec), intent(out), optional :: arrayspec
     type(ESMF_FieldStatus_Flag),intent(out), optional :: status
     type(ESMF_GeomType_Flag), intent(out), optional :: geomtype
     type(ESMF_Grid), intent(out), optional :: grid
     type(ESMF_Mesh), intent(out), optional :: mesh
     type(ESMF_LocStream), intent(out), optional :: locstream
     type(ESMF_XGrid), intent(out), optional :: xgrid
     type(ESMF_Array), intent(out), optional :: array
     type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
     integer, intent(out), optional :: dimCount
     integer, intent(out), optional :: rank
     type(ESMF_StaggerLoc), intent(out), optional :: staggerloc
     type(ESMF_MeshLoc), intent(out), optional :: meshloc
     type(ESMF_XGridSide_Flag), intent(out), optional :: xgridside
     integer, intent(out), optional :: gridindex
     integer, intent(out), optional :: gridToFieldMap(:)
     integer, intent(out), optional :: ungriddedLBound(:)
     integer, intent(out), optional :: ungriddedUBound(:)
     integer, intent(out), optional :: totalLWidth(:,:)
     integer, intent(out), optional :: totalUWidth(:,:)
     integer, intent(out), optional :: localDeCount
     character(len=*), intent(out), optional :: name
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r - *except those arguments indicated below*.

DESCRIPTION:

Query an `ESMF_Field` for various things. All arguments after the `field` are optional. To select individual items use the named_argument=value syntax. For an example and associated documentation using this method see section 22.3.3.
The arguments are:

field `ESMF_Field` object to query.

[arrayspec] `ESMF_ArraySpec` object containing the type/kind/rank information of the Field object.

[status] The status of the Field. See section 22.2.1 for a complete list of values.

[geomtype] The type of geometry on which the Field is built. See section 9.19 for the range of values.

[grid] `ESMF_Grid`.

[mesh] STATUS:*This argument is excluded from the backward compatibility statement.*
ESMF_Mesh.

[locstream] STATUS:*This argument is excluded from the backward compatibility statement.*
ESMF_LocStream.

[xgrid] STATUS:*This argument is excluded from the backward compatibility statement.*
ESMF_XGrid.

[array] ESMF_Array.

[typekind] TypeKind specifier for Field. See section 9.45 for a complete list of values.

[dimCount] Number of geometrical dimensions in field. For an detailed discussion of this parameter, please see section 22.3.22 and section 22.3.23.

[rank] Number of dimensions in the physical memory of the field data. It is identical to dimCount when the corresponding grid is a non-arbitrary grid. It is less than dimCount when the grid is arbitrarily distributed. For an detailed discussion of this parameter, please see section 22.3.22 and section 22.3.23.

[staggerloc] Stagger location of data in grid cells. For valid predefined values and interpretation of results see section 27.2.7.

[meshloc] STATUS:*This argument is excluded from the backward compatibility statement.*
The part of the mesh to build the Field on. Can be either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

[xgridside] STATUS:*This argument is excluded from the backward compatibility statement.*
The side of the XGrid that the Field was created on. See section 30.2.1 for a complete list of values.

[gridIndex] STATUS:*This argument is excluded from the backward compatibility statement.*
If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid/Mesh on that side the Field was created on.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLWidth] Lower bound of halo region. The size of the first dimension of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount). The size of the 2nd dimension of this array is localDeCount.

[totalUWidth] Upper bound of halo region. The size of the first dimension of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`). The size of the 2nd dimension of this array is localDeCount.

[localDeCount] Upon return this holds the number of PET-local DEs defined in the DELayout associated with the Field object.

[name] Name of queried item.

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

22.6.43 ESMF_FieldGet - Get Fortran data pointer from a Field

INTERFACE:

```
! Private name; call using ESMF_FieldGet()
subroutine ESMF_FieldGetDataPtr<rank><type><kind>(field, localDe, &
farrayPtr, exclusiveLBound, exclusiveUBound, exclusiveCount, &
computationalLBound, computationalUBound, computationalCount, &
totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(in) :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer, intent(in), optional :: localDe
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
  integer, intent(out), optional :: exclusiveLBound(:)
  integer, intent(out), optional :: exclusiveUBound(:)
  integer, intent(out), optional :: exclusiveCount(:)
  integer, intent(out), optional :: computationalLBound(:)
  integer, intent(out), optional :: computationalUBound(:)
  integer, intent(out), optional :: computationalCount(:)
  integer, intent(out), optional :: totalLBound(:)
  integer, intent(out), optional :: totalUBound(:)
  integer, intent(out), optional :: totalCount(:)
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get a Fortran pointer to DE-local memory allocation within `field`. For convenience DE-local bounds can be queried at the same time. For an example and associated documentation using this method see section 22.3.2.
The arguments are:

field `ESMF_Field` object.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

farrayPtr Fortran array pointer which will be pointed at DE-local memory allocation. It depends on the specific entry point of ESMF_FieldCreate() used during field creation, which Fortran operations are supported on the returned farrayPtr. See 22.4 for more details.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. exclusiveLBound must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. exclusiveUBound must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items, exclusiveUBound-exclusiveLBound+1, in the exclusive region per dimension. exclusiveCount must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the computational region. computationalLBound must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the lower bounds of the computational region. computationalUBound must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. computationalUBound-computationalLBound+1). computationalCount must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. totalLBound must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[totalUBound] Upon return this holds the lower bounds of the total region. totalUBound must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. totalUBound-totalLBound+1). computationalCount must be allocated to be of size equal to field's dimCount. See section 24.2.6 for a description of the regions and their associated bounds and counts.

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

22.6.44 ESMF_FieldGetBounds - Get Field data bounds

INTERFACE:

```
! Private name; call using ESMF_FieldGetBounds()
subroutine ESMF_FieldGetBounds(field, localDe, &
  exclusiveLBound, exclusiveUBound, exclusiveCount, computationalLBound, &
  computationalUBound, computationalCount, totalLBound, &
  totalUBound, totalCount, rc)
```

ARGUMENTS:

```
     type(ESMF_Field), intent(in) :: field
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(in), optional :: localDe
     integer, intent(out), optional :: exclusiveLBound(:)
     integer, intent(out), optional :: exclusiveUBound(:)
     integer, intent(out), optional :: exclusiveCount(:)
     integer, intent(out), optional :: computationalLBound(:)
     integer, intent(out), optional :: computationalUBound(:)
     integer, intent(out), optional :: computationalCount(:)
     integer, intent(out), optional :: totalLBound(:)
     integer, intent(out), optional :: totalUBound(:)
     integer, intent(out), optional :: totalCount(:)
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method returns the bounds information of a field that consists of a internal grid and a internal array. The exclusive and computational bounds are shared between the grid and the array but the total bounds are the array bounds plus the halo width. The count is the number of elements between each bound pair.
The arguments are:

field Field to get the information from.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to the field rank. Please see section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to the field rank. Please see section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items, `exclusiveUBound-exclusiveLBound+1`, in the exclusive region per dimension. `exclusiveCount` must be allocated to be of size equal to the field rank. Please see section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the stagger region. `computationalLBound` must be allocated to be of size equal to the field rank. Please see section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the stagger region. `computationalUBound` must be allocated to be of size equal to the field rank. Please see section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to the field rank. Please see section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to the field rank.

[totalUBound] Upon return this holds the upper bounds of the total region. `totalUBound` must be allocated to be of size equal to the field rank.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `totalCount` must be allocated to be of size equal to the field rank.

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

22.6.45 ESMF_FieldPrint - Print the contents of a Field

INTERFACE:

```
      subroutine ESMF_FieldPrint(field, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Prints information about the `field` to `stdout`. This subroutine goes through the internal data members of a field data type and prints information of each data member.

The arguments are:

field An `ESMF_Field` object.

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

22.6.46 ESMF_FieldRead - Read Field data from a file

INTERFACE:

```
      subroutine ESMF_FieldRead(field, file, &
        timeslice, iofmt, rc)
```

ARGUMENTS:

```
      type(ESMF_Field),     intent(inout)          :: field
      character(*),         intent(in)             :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,              intent(in),  optional  :: timeslice
      type(ESMF_IOFmtFlag), intent(in),  optional  :: iofmt
      integer,              intent(out), optional  :: rc
```

DESCRIPTION:

Read Field data from a file and put it into an ESMF_Field object. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.

- Not supported in ESMF_COMM=mpiuni mode.

The arguments are:

field The ESMF_Field object in which the read data is returned.

file The name of the file from which Field data is read.

timeslice Number of slices to be read from file, starting from the 1st slice

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to ESMF_IOFMT_NETCDF.

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

22.6.47 ESMF_FieldValidate - Check validity of a Field

INTERFACE:

```
      subroutine ESMF_FieldValidate(field, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the field is internally consistent. Currently this method determines if the field is uninitialized or already destroyed. It validates the contained array and grid objects. The code also checks if the array and grid sizes agree. This check compares the distgrid contained in array and grid; then it proceeds to compare the computational bounds contained in array and grid.
The method returns an error code if problems are found.
The arguments are:

field ESMF_Field to validate.

[rc] Return code; equals ESMF_SUCCESS if the field is valid.

22.6.48 ESMF_FieldWrite - Write Field data into a file

INTERFACE:

```
      subroutine ESMF_FieldWrite(field, file, &
        append, timeslice, iofmt, rc)
```

ARGUMENTS:

```
      type(ESMF_Field),     intent(in)             :: field
      character(*),         intent(in)             :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      logical,              intent(in),  optional  :: append
      integer,              intent(in),  optional  :: timeslice
      type(ESMF_IOFmtFlag), intent(in),  optional  :: iofmt
      integer,              intent(out), optional  :: rc
```

DESCRIPTION:

Write Field data into a file. For this API to be functional, the environment variable ESMF_PIO should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in ESMF_COMM=mpiuni mode.

The arguments are:

field The ESMF_Field object that contains data to be written.

file The name of the output file to which Field data is written.

[append] Logical: if .true., data (with attributes) is appended to an existing file; default is .false.

[timeslice] Some IO formats (e.g. NetCDF) support the output of data in form of time slices. The timeslice argument provides access to this capability. Usage of this feature requires that the first slice is written with a positive timeslice value, and that subsequent slices are written with a timeslice argument that increments by one each time. By default, i.e. by omitting the timeslice argument, no provisions for time slicing are made in the output file.

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to ESMF_IOFMT_NETCDF.

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

22.7 Class API: Field Utilities

22.7.1 ESMF_GridGetFieldBounds - Get precomputed Fortran data array bounds for creating a Field from a Grid and Fortran array

INTERFACE:

```
subroutine ESMF_GridGetFieldBounds(grid, &
    localDe, staggerloc, gridToFieldMap, &
    ungriddedLBound, ungriddedUBound, &
    totalLWidth, totalUWidth, &
    totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
     type(ESMF_Grid),       intent(in)            :: grid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,               intent(in),  optional :: localDe
     type(ESMF_StaggerLoc), intent(in),  optional :: staggerloc
     integer,               intent(in),  optional :: gridToFieldMap(:)
     integer,               intent(in),  optional :: ungriddedLBound(:)
     integer,               intent(in),  optional :: ungriddedUBound(:)
     integer,               intent(in),  optional :: totalLWidth(:)
     integer,               intent(in),  optional :: totalUWidth(:)
     integer,               intent(out), optional :: totalLBound(:)
     integer,               intent(out), optional :: totalUBound(:)
     integer,               intent(out), optional :: totalCount(:)
     integer,               intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a ESMF_Field from a ESMF_Grid and the Fortran data array. For an example and associated documentation using this method see section 22.3.9.
The arguments are:

grid ESMF_Grid.

[localDe] Local DE for which information is requested. [0,..,localDeCount-1]. For localDeCount==1 the localDe argument may be omitted, in which case it will default to localDe=0.

[staggerloc] Stagger location of data in grid cells. For valid predefined values and interpretation of results see section 27.2.7.

[gridToFieldMap] List with number of elements equal to the grid|s dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid|s dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLWidth] Lower bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of dimensions in the grid. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalLBound] The relative lower bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_Grid and Fortran data array. This is an output variable from this user interface.

The relative lower bounds of Fortran data array to be used

[totalUBound] The relative upper bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_Grid and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in ESMF_FieldCreate from ESMF_Grid and Fortran data array. This is an output variable from this user interface.

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

22.7.2 ESMF_LocStreamGetFieldBounds - Get precomputed Fortran data array bounds for creating a Field from a LocStream and Fortran array

INTERFACE:

```
    subroutine ESMF_LocStreamGetFieldBounds(locstream, &
        localDe, gridToFieldMap, &
        ungriddedLBound, ungriddedUBound, &
        totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
    type(ESMF_LocStream), intent(in)            :: locstream
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,              intent(in),  optional :: localDe
    integer,              intent(in),  optional :: gridToFieldMap(:)
    integer,              intent(in),  optional :: ungriddedLBound(:)
    integer,              intent(in),  optional :: ungriddedUBound(:)
    integer,              intent(out), optional :: totalLBound(:)
    integer,              intent(out), optional :: totalUBound(:)
    integer,              intent(out), optional :: totalCount(:)
    integer,              intent(out), optional :: rc
```

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a `ESMF_Field` from a `ESMF_LocStream` and the Fortran data array. For an example and associated documentation using this method see section 22.3.9.
The arguments are:

locstream `ESMF_LocStream`.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[gridToFieldMap] List with number of elements equal to 1. The list elements map the dimension of the `locstream` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map the `locstream`|s dimension against the lowest dimension of the `field` in sequence, i.e. `gridToFieldMap` = (/1/). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLBound] The relative lower bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

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

22.7.3 ESMF_MeshGetFieldBounds - Get precomputed Fortran data array bounds for creating a Field from a Mesh and a Fortran array

INTERFACE:

```
     subroutine ESMF_MeshGetFieldBounds(mesh, &
         localDe, gridToFieldMap, &
         ungriddedLBound, ungriddedUBound, &
         totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
     type(ESMF_Mesh), intent(in)            :: mesh
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,         intent(in),  optional :: localDe
     integer,         intent(in),  optional :: gridToFieldMap(:)
     integer,         intent(in),  optional :: ungriddedLBound(:)
     integer,         intent(in),  optional :: ungriddedUBound(:)
     integer,         intent(out), optional :: totalLBound(:)
     integer,         intent(out), optional :: totalUBound(:)
     integer,         intent(out), optional :: totalCount(:)
     integer,         intent(out), optional :: rc
```

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a ESMF_Field from a ESMF_Mesh and the Fortran data array. For an example and associated documentation using this method see section 22.3.9.
The arguments are:

mesh ESMF_Mesh.

[localDe] Local DE for which information is requested. [0,..,localDeCount-1]. For localDeCount==1 the localDe argument may be omitted, in which case it will default to localDe=0.

[gridToFieldMap] List with number of elements equal to the grid|s dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid|s dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLBound] The relative lower bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_Mesh and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_Mesh and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in ESMF_FieldCreate from ESMF_Mesh and Fortran data array. This is an output variable from this user interface.

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

22.7.4 ESMF_XGridGetFieldBounds - Get precomputed Fortran data array bounds for creating a Field from an XGrid and a Fortran array

INTERFACE:

```
     subroutine ESMF_XGridGetFieldBounds(xgrid, &
         xgridside, gridindex, localDe, gridToFieldMap, &
         ungriddedLBound, ungriddedUBound, &
         totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
     type(ESMF_XGrid),          intent(in)            :: xgrid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_XGridSide_Flag), intent(in),  optional :: xgridside
     integer,                   intent(in),  optional :: gridindex
     integer,                   intent(in),  optional :: localDe
     integer,                   intent(in),  optional :: gridToFieldMap(:)
     integer,                   intent(in),  optional :: ungriddedLBound(:)
     integer,                   intent(in),  optional :: ungriddedUBound(:)
     integer,                   intent(out), optional :: totalLBound(:)
     integer,                   intent(out), optional :: totalUBound(:)
     integer,                   intent(out), optional :: totalCount(:)
     integer,                   intent(out), optional :: rc
```

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a ESMF_Field from a ESMF_XGrid and the Fortran data array. For an example and associated documentation using this method see section 22.3.9.
The arguments are:

xgrid `ESMF_XGrid` object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[gridToFieldMap] List with number of elements equal to 1. The list elements map the dimension of the `locstream` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map the `locstream`|s dimension against the lowest dimension of the `field` in sequence, i.e. `gridToFieldMap` = (/1/). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLBound] The relative lower bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_LocStream` and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_LocStream` and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_LocStream` and Fortran data array. This is an output variable from this user interface.

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

22.8 Class API: Field Communications

22.8.1 ESMF_FieldGather - Gather a Fortran array from an ESMF_Field

INTERFACE:

```
subroutine ESMF_FieldGather<rank><type><kind>(field, farray, &
rootPet, tile, vm, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(in) :: field
  <type>(ESMF_KIND_<kind>), intent(out), target :: farray(<rank>)
  integer, intent(in) :: rootPet
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
integer, intent(in), optional :: tile
type(ESMF_VM), intent(in), optional :: vm
integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gather the data of an ESMF_Field object into the `farray` located on `rootPET`. A single DistGrid tile of `array` must be gathered into `farray`. The optional `tile` argument allows selection of the tile. For Fields defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Field.
If the Field contains replicating DistGrid dimensions data will be gathered from the numerically higher DEs. Replicated data elements in numericaly lower DEs will be ignored.
The implementation of Scatter and Gather is not seqence index based. If the Field is built on arbitrarily distributed Grid, Mesh, LocStream or XGrid, Gather will not gather data to rootPet from source data points corresponding to the sequence index on rootPet. Instead Gather will gather a contiguous memory range from source PET to rootPet. The size of the memory range is equal to the number of data elements on the source PET. Vice versa for the Scatter operation. In this case, the user should use `ESMF_FieldRedist` to achieve the same data operation result. For examples how to use `ESMF_FieldRedist` to perform Gather and Scatter, please refer to 22.3.35 and 22.3.34.
This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.
For examples and associated documentations using this method see Section 22.3.31.
The arguments are:

field The `ESMF_Field` object from which data will be gathered.

{farray} The Fortran array into which to gather data. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid destination array, i.e. `farray`.

[tile] The DistGrid tile in `field` from which to gather `farray`. By default `farray` will be gathered from tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

22.8.2 ESMF_FieldHalo - Execute a FieldHalo operation

INTERFACE:

```
subroutine ESMF_FieldHalo(field, routehandle,  &
                          routesyncflag, finishedflag, checkflag, rc)
```

ARGUMENTS:

```
    type(ESMF_Field),       intent(inout)           :: field
    type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_RouteSync_Flag), intent(in), optional :: routesyncflag
    logical,                intent(out),  optional  :: finishedflag
    logical,                intent(in),   optional  :: checkflag
    integer,                intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed Field halo operation for `field`. The `field` argument must be weakly congruent and typekind conform to the Field used during `ESMF_FieldHaloStore()`. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions.
See `ESMF_FieldHaloStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.

field `ESMF_Field` containing data to be haloed.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section 9.38 for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[checkflag] If set to `.TRUE.` the input Field pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

22.8.3 ESMF_FieldHaloRelease - Release resources associated with a Field halo operation

INTERFACE:

```
   subroutine ESMF_FieldHaloRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a Field halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

22.8.4 ESMF_FieldHaloStore - Store a FieldHalo operation

INTERFACE:

```
subroutine ESMF_FieldHaloStore(field, routehandle,  &
  startregion, haloLDepth, haloUDepth, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),       intent(inout)                :: field
     type(ESMF_RouteHandle), intent(inout)                :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_StartRegion_Flag), intent(in),    optional :: startregion
     integer,                intent(in),         optional :: haloLDepth(:)
     integer,                intent(in),         optional :: haloUDepth(:)
     integer,                intent(out),        optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a Field halo operation over the data in `field`. By default, i.e. without specifying `startregion`, `haloLDepth` and `haloUDepth`, all elements in the total Field region that lie outside the exclusive region will be considered potential destination elements for halo. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.
Specifying `startregion` allows to change the shape of the effective halo region from the inside. Setting this flag to `ESMF_STARTREGION_COMPUTATIONAL` means that only elements outside the computational region of the Field are considered for potential destination elements for halo. The default is `ESMF_STARTREGION_EXCLUSIVE`.
The `haloLDepth` and `haloUDepth` arguments allow to reduce the extent of the effective halo region. Starting at the region specified by `startregion`, the `haloLDepth` and `haloUDepth` define a halo depth in each direction. Note that the maximum halo region is limited by the total Field region, independent of the actual `haloLDepth` and `haloUDepth` setting. The total Field region is local DE specific. The `haloLDepth` and `haloUDepth` are interpreted as the maximum desired extent, reducing the potentially larger region available for halo.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldHalo()` on any Field that is weakly congruent and typekind conform to `field`. Congruent Fields possess matching DistGrids, and the shape of the local field tiles matches between the Fieldss for every DE. For weakly congruent Fieldss the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fieldss that differ in the number of elements in the left most undistributed dimensions.
This call is *collective* across the current VM.

field `ESMF_Field` containing data to be haloed. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

[startregion] The start of the effective halo region on every DE. The default setting is `ESMF_STARTREGION_EXCLUSIVE`, rendering all non-exclusive elements potential halo destination elments. See section 9.41 for a complete list of valid settings.

[haloLDepth] This vector specifies the lower corner of the effective halo region with respect to the lower corner of `startregion`. The size of `haloLDepth` must equal the number of distributed Array dimensions.

[haloUDepth] This vector specifies the upper corner of the effective halo region with respect to the upper corner of `startregion`. The size of `haloUDepth` must equal the number of distributed Array dimensions.

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

22.8.5 ESMF_FieldRedist - Execute a Field redistribution

INTERFACE:

```
   subroutine ESMF_FieldRedist(srcField, dstField, routehandle,  &
     checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_Field),       intent(in),optional     :: srcField
         type(ESMF_Field),       intent(inout),optional  :: dstField
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),   optional  :: checkflag
         integer,                intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed Field redistribution from `srcField` to `dstField`. Both `srcField` and `dstField` must be congruent and typekind conform with the respective Fields used during `ESMF_FieldRedistStore()`. Congruent Fields possess matching DistGrids and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Because Grid dimensions are mapped to Field in a sequence order, it's necessary to map the ungridded dimensions to the first set of dimensions in order to use the weakly congruent Field redist feature. Not providing a non-default gridToFieldMap during Field creation and then using such Fields in a weakly congruent manner in Field communication methods leads to undefined behavior.
It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.
See `ESMF_FieldRedistStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.
For examples and associated documentations using this method see Section 22.3.33.

[srcField] `ESMF_Field` with source data.

[dstField] `ESMF_Field` with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input Field pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

22.8.6 ESMF_FieldRedistRelease - Release resources associated with Field redistribution

INTERFACE:

```
   subroutine ESMF_FieldRedistRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a Field redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

22.8.7 ESMF_FieldRedistStore - Precompute Field redistribution with a local factor argument

INTERFACE:

```
   ! Private name; call using ESMF_FieldRedistStore()
   subroutine ESMF_FieldRedistStore<type><kind>(srcField, dstField, &
          routehandle, factor, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),          intent(in)             :: srcField
     type(ESMF_Field),          intent(inout)          :: dstField
     type(ESMF_RouteHandle),    intent(inout)          :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)             :: factor
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                   intent(in), optional  :: srcToDstTransposeMap(:)
     integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

`ESMF_FieldRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_FieldRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_FieldRedistStore()` method, as provided through the separate entry points shown in 22.8.7 and 22.8.8, is described in the following paragraphs as a whole.

Store a Field redistribution operation from `srcField` to `dstField`. Interface 22.8.7 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 22.8.8. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*.

Source Field, destination Field, and the factor may be of different <type><kind>. Further, source and destination Fields may differ in shape, however, the number of elements must match.

If `srcToDstTransposeMap` is not specified the redistribution corresponds to an identity mapping of the sequentialized source Field to the sequentialized destination Field. If the `srcToDstTransposeMap` argument is provided

it must be identical on all PETs. The `srcToDstTransposeMap` allows source and destination Field dimensions to be transposed during the redistribution. The number of source and destination Field dimensions must be equal under this condition and the size of mapped dimensions must match.
It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldRedist()` on any pair of Fields that are weakly congruent and typekind conform with the `srcField`, `dstField` pair. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Because Grid dimensions are mapped to Field in a sequence order, it's necessary to map the ungridded dimensions to the first set of dimensions in order to use the weakly congruent Field redist feature. Not providing a non-default gridToFieldMap during Field creation and then using such Fields in a weakly congruent manner in Field communication methods leads to undefined behavior.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.
For examples and associated documentations using this method see Section 22.3.33.
The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multiply data. Default is 1. See full method description above for details on the interplay with other PETs.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcField`. Each entry maps the corresponding `srcField` dimension against the specified `dstField` dimension. Mixing of distributed and undistributed dimensions is supported.

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

22.8.8 ESMF_FieldRedistStore - Precompute Field redistribution without a local factor argument

INTERFACE:

```
   ! Private name; call using ESMF_FieldRedistStore()
   subroutine ESMF_FieldRedistStoreNF(srcField, dstField, &
          routehandle, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),       intent(in)            :: srcField
     type(ESMF_Field),       intent(inout)         :: dstField
     type(ESMF_RouteHandle), intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),  optional :: srcToDstTransposeMap(:)
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

`ESMF_FieldRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_FieldRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_FieldRedistStore()` method, as provided through the separate entry points shown in 22.8.7 and 22.8.8, is described in the following paragraphs as a whole.
Store a Field redistribution operation from `srcField` to `dstField`. Interface 22.8.7 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 22.8.8. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.
Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*.
Source Field, destination Field, and the factor may be of different <type><kind>. Further, source and destination Fields may differ in shape, however, the number of elements must match.
If `srcToDstTransposeMap` is not specified the redistribution corresponds to an identity mapping of the sequentialized source Field to the sequentialized destination Field. If the `srcToDstTransposeMap` argument is provided it must be identical on all PETs. The `srcToDstTransposeMap` allows source and destination Field dimensions to be transposed during the redistribution. The number of source and destination Field dimensions must be equal under this condition and the size of mapped dimensions must match.
It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldRedist()` on any pair of Fields that are weakly congruent and typekind conform with the `srcField`, `dstField` pair. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Because Grid dimensions are mapped to Field in a sequence order, it's necessary to map the ungridded dimensions to the first set of dimensions in order to use the weakly congruent Field redist feature. Not providing a non-default gridToFieldMap during Field creation and then using such Fields in a weakly congruent manner in Field communication methods leads to undefined behavior.
This call is *collective* across the current VM.
For examples and associated documentations using this method see Section 22.3.33.
The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcField`. Each entry maps the corresponding `srcField` dimension against the specified `dstField` dimension. Mixing of distributed and undistributed dimensions is supported.

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

22.8.9 ESMF_FieldRegrid - Compute a regridding operation

INTERFACE:

```
subroutine ESMF_FieldRegrid(srcField, dstField, &
             routehandle, zeroregion, checkflag, rc)
```

ARGUMENTS:

```
type(ESMF_Field),       intent(in),    optional :: srcField
type(ESMF_Field),       intent(inout), optional :: dstField
type(ESMF_RouteHandle), intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Region_Flag), intent(in),    optional :: zeroregion
logical,                intent(in),    optional :: checkflag
integer,                intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute the precomputed regrid operation stored in `routehandle` to interpolate from `srcField` to `dstField`. See `ESMF_FieldRegridStore()` on how to precompute the `routehandle`.
Both `srcField` and `dstField` must be congruent with the respective Fields used during `ESMF_FieldRegridStore()`. In the case of the Regrid operation congruent Fields are built upon the same stagger location and on the same Grid. The routehandle represents the interpolation between the Grids as they were during the `ESMF_FieldRegridStore()` call. So if the coordinates at the stagger location in the Grids change, a new call to `ESMF_FieldRegridStore()` is necessary to compute the interpolation between that new set of coordinates. Note `ESMF_FieldRegridStore()` assumes the coordinates used in the Grids upon which the Fields are built are in degrees.
It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.
This call is *collective* across the current VM.

[srcField] `ESMF_Field` with source data.

[dstField] `ESMF_Field` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in `dstField` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstField` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section 9.36 for a complete list of valid settings.

[checkflag] If set to `.TRUE.` the input Array pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

22.8.10 ESMF_FieldRegridRelease - Free resources used by a regridding operation

INTERFACE:

```
subroutine ESMF_FieldRegridRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Free resources used by regrid objec
The arguments are:

routehandle Handle carrying the sparse matrix

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

22.8.11 ESMF_FieldRegridStore - Precompute a Field regridding operation and return a RouteHandle and weights

INTERFACE:

```
   !   Private name; call using ESMF_FieldRegridStore()
       subroutine ESMF_FieldRegridStoreNX(srcField, dstField, &
                                          srcMaskValues, dstMaskValues, &
                                          regridmethod, &
                                          polemethod, regridPoleNPnts, &
                                          unmappedaction, &
                                          routehandle, indices, weights, &
                                          srcFracField, dstFracField, rc)
```

ARGUMENTS:

```
      type(ESMF_Field),            intent(in)             :: srcField
      type(ESMF_Field),            intent(inout)          :: dstField
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),       intent(in),   optional :: srcMaskValues(:)
      integer(ESMF_KIND_I4),       intent(in),   optional :: dstMaskValues(:)
      type(ESMF_RegridMethod_Flag),intent(in),   optional :: regridmethod
      type(ESMF_PoleMethod_Flag),  intent(in),   optional :: polemethod
      integer,                     intent(in),   optional :: regridPoleNPnts
      type(ESMF_UnmappedAction_Flag),intent(in), optional :: unmappedaction
      type(ESMF_RouteHandle),      intent(inout),optional :: routehandle
      integer(ESMF_KIND_I4),       pointer,      optional :: indices(:,:)
      real(ESMF_KIND_R8),          pointer,      optional :: weights(:)
      type(ESMF_Field),            intent(inout),optional :: srcFracField
      type(ESMF_Field),            intent(inout),optional :: dstFracField
      integer,                      intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates a sparse matrix operation (stored in `routehandle`) that contains the calculations and communications necessary to interpolate from `srcField` to `dstField`. The routehandle can then be used in the call `ESMF_FieldRegrid()` to interpolate between the Fields. The user may also get the interpolation matrix in sparse matrix form via the optional arguments `indices` and `weights`.

The routehandle generated by this call is based just on the coordinates at the Fields' stagger locations in the Grids contained in the Fields. If those coordinates don't change the routehandle can be used repeatedly to interpolate from the source Field to the destination Field. This is true even if the data in the Fields changes. The routehandle may also be used to interpolate between any source and destination Field which are created on the same stagger location and Grid as the original Fields.
When it's no longer needed the routehandle should be destroyed by using `ESMF_FieldRegridRelease()` to free the memory it's using. Note `ESMF_FieldRegridStore()` assumes the coordinates used in the Grids upon which the Fields are built are in degrees.
The arguments are:

srcField Source Field.

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

dstField Destination Field.

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

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

[routehandle] The handle that implements the regrid and that can be used in later `ESMF_FieldRegrid`.

[indices] The indices for the sparse matrix.

[weights] The weights for the sparse matrix.

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

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

[regridmethod] The type of interpolation. Please see Section 22.2.3 for a list of valid options. If not specified, defaults to `ESMF_REGRIDMETHOD_BILINEAR`.

[polemethod] Which type of artificial pole to construct on the source Grid for regridding. Please see Section 22.2.2 for a list of valid options. If not specified, defaults to `ESMF_POLEMETHOD_ALLAVG`.

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

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

22.8.12 ESMF_FieldRegridStore - Precompute a Field regridding operation and return a RouteHandle using XGrid

INTERFACE:

```
!   Private name; call using ESMF_FieldRegridStore()
    subroutine ESMF_FieldRegridStoreX(xgrid, srcField, dstField, &
                                      routehandle, rc)
```

ARGUMENTS:

```
      type(ESMF_XGrid),       intent(in)             :: xgrid
      type(ESMF_Field),       intent(in)             :: srcField
      type(ESMF_Field),       intent(inout)          :: dstField
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_RouteHandle), intent(inout), optional :: routehandle
      integer,                intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates a sparse matrix operation (stored in `routehandle`) that contains the calculations and communications necessary to interpolate from `srcField` to `dstField`. The routehandle can then be used in the call `ESMF_FieldRegrid()` to interpolate between the Fields. Informaton such as index mapping and weights are obtained from the XGrid by matching the Field Grids in the XGrid. It's important the Grids in the `srcField` and `dstField` donot match, i.e. they are different in either tological or geometric characteristic.
The routehandle generated by this call is subsequently computed based on these information. If those information don't change the routehandle can be used repeatedly to interpolate from the source Field to the destination Field. This is true even if the data in the Fields changes. The routehandle may also be used to interpolate between any source and destination Field which are created on the same stagger location and Grid as the original Fields.
When it's no longer needed the routehandle should be destroyed by using `ESMF_FieldRegridRelease()` to free the memory it's using. Note `ESMF_FieldRegridStore()` assumes the coordinates used in the Grids upon which the Fields are built are in degrees.
The arguments are:

xgrid Exchange Grid.

srcField Source Field.

dstField Destination Field.

[routehandle] The handle that implements the regrid and that can be used in later `ESMF_FieldRegrid`.

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

22.8.13 ESMF_FieldScatter - Scatter a Fortran array across the ESMF_Field

INTERFACE:

```
  subroutine ESMF_FieldScatter<rank><type><kind>(field, farray, &
  rootPet, tile, vm, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  mtype (ESMF_KIND_mtypekind),intent(in), target :: farray(mdim)
  integer, intent(in) :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer, intent(in), optional :: tile
  type(ESMF_VM), intent(in), optional :: vm
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Scatter the data of `farray` located on `rootPET` across an ESMF_Field object. A single `farray` must be scattered across a single DistGrid tile in Field. The optional `tile` argument allows selection of the tile. For Fields defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Field.

If the Field contains replicating DistGrid dimensions data will be scattered across all of the replicated pieces.

The implementation of Scatter and Gather is not seqence index based. If the Field is built on arbitrarily distributed Grid, Mesh, LocStream or XGrid, Scatter will not scatter data from rootPet to the destination data points corresponding to the sequence index on the rootPet. Instead Scatter will scatter a contiguous memory range from rootPet to destination PET. The size of the memory range is equal to the number of data elements on the destination PET. Vice versa for the Gather operation. In this case, the user should use `ESMF_FieldRedist` to achieve the same data operation result. For examples how to use `ESMF_FieldRedist` to perform Gather and Scatter, please refer to 22.3.35 and 22.3.34.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.

For examples and associated documentations using this method see Section 22.3.32.

The arguments are:

field The `ESMF_Field` object across which data will be scattered.

{farray} The Fortran array that is to be scattered. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid data in `farray`.

[tile] The DistGrid tile in `field` into which to scatter `farray`. By default `farray` will be scattered into tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

22.8.14 ESMF_FieldSMM - Execute a Field sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_FieldSMM(srcField, dstField, routehandle, &
              zeroregion, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_Field),       intent(in),   optional  :: srcField
         type(ESMF_Field),       intent(inout),optional  :: dstField
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         type(ESMF_Region_Flag),  intent(in),   optional :: zeroregion
         logical,                intent(in),   optional  :: checkflag
         integer,                intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed Field sparse matrix multiplication from `srcField` to `dstField`. Both `srcField` and `dstField` must be congruent and typekind conform with the respective Fields used during

ESMF_FieldSMMStore(). Congruent Fields possess matching DistGrids and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same routehandle can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Because Grid dimensions are mapped to Field in a sequence order, it's necessary to map the ungridded dimensions to the first set of dimensions in order to use the weakly congruent Field SMM feature. Not providing a non-default gridToFieldMap during Field creation and then using such Fields in a weakly congruent manner in Field communication methods leads to undefined behavior.

It is erroneous to specify the identical Field object for srcField and dstField arguments.

See ESMF_FieldSMMStore() on how to precompute routehandle.

This call is *collective* across the current VM.

For examples and associated documentations using this method see Section 22.3.36.

[srcField] ESMF_Field with source data.

[dstField] ESMF_Field with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to ESMF_REGION_TOTAL *(default)* the total regions of all DEs in dstField will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to ESMF_REGION_EMPTY the elements in dstField will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting zeroregion to ESMF_REGION_SELECT will only zero out those elements in the destination Field that will be updated by the sparse matrix multiplication. See section 9.36 for a complete list of valid settings.

[checkflag] If set to .TRUE. the input Field pair will be checked for consistency with the precomputed operation provided by routehandle. If set to .FALSE. *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to .FALSE. to achieve highest performance.

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

22.8.15 ESMF_FieldSMMRelease - Release resources associated with Field

sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_FieldSMMRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with a Field sparse matrix multiplication. After this call routehandle becomes invalid.

routehandle Handle to the precomputed Route.

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

22.8.16 ESMF_FieldSMMStore - Precompute Field sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_FieldSMMStore()
subroutine ESMF_FieldSMMStore<type><kind>(srcField, dstField, &
       routehandle, factorList, factorIndexList, rc)
```

ARGUMENTS:

```
    type(ESMF_Field),         intent(in)           :: srcField
    type(ESMF_Field),         intent(inout)        :: dstField
    type(ESMF_RouteHandle),   intent(inout)        :: routehandle
    <type>(ESMF_KIND_<kind>), intent(in)           :: factorList(:)
    integer,                  intent(in),          :: factorIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a Field sparse matrix multiplication operation from `srcField` to `dstField`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source Field vector to the destination Field vector.
Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical Field object for srcField and dstField arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldSMM()` on any pair of Fields that are weakly congruent and typekind conform with the `srcField`, `dstField` pair. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Because Grid dimensions are mapped to Field in a sequence order, it's necessary to map the ungridded dimensions to the first set of dimensions in order to use the weakly congruent Field SMM feature. Not providing a non-default gridToFieldMap during Field creation and then using such Fields in a weakly congruent manner in Field communication methods leads to undefined behavior.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.
For examples and associated documentations using this method see Section 22.3.36.
The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

The second dimension of `factorIndexList` steps through the list of

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the `srcField` while `factorIndexList(2,:)` specifies the sequence index of the destination element in `dstField`. For this format to be a valid option source and destination Fields must have matching number of tensor elements (the product of the sizes of all Field tensor dimensions). Under this condition an identiy matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the `factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the `srcField`. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the `dstField`.

See section 24.2.17 for details on the definition of Field *sequence indices* and *tensor sequence indices*.

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

22.8.17 ESMF_FieldSMMStore - Precompute Field sparse matrix multiplication without local factors

INTERFACE:

```
! Private name; call using ESMF_FieldSMMStore()
subroutine ESMF_FieldSMMStoreNF(srcField, dstField, &
       routehandle, factorList, factorIndexList, rc)
```

ARGUMENTS:

```
    type(ESMF_Field),         intent(in)            :: srcField
    type(ESMF_Field),         intent(inout)         :: dstField
    type(ESMF_RouteHandle),   intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store a Field sparse matrix multiplication operation from `srcField` to `dstField`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both srcField and dstField are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source Field vector to the destination Field vector.
Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.
It is erroneous to specify the identical Field object for srcField and dstField arguments.
The routine returns an ESMF_RouteHandle that can be used to call ESMF_FieldSMM() on any pair of Fields that are weakly congruent and typekind conform with the srcField, dstField pair. Congruent Fields possess matching DistGrids, and the shape of the local array tiles matches between the Fields for every DE. For weakly congruent Fields the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same routehandle can be applied to a large class of similar Fields that differ in the number of elements in the left most undistributed dimensions. Because Grid dimensions are mapped to Field in a sequence order, it's necessary to map the ungridded dimensions to the first set of dimensions in order to use the weakly congruent Field SMM feature. Not providing a non-default gridToFieldMap during Field creation and then using such Fields in a weakly congruent manner in Field communication methods leads to undefined behavior.
This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is collective across the current VM.
For examples and associated documentations using this method see Section 22.3.36.
The arguments are:

srcField ESMF_Field with source data.

dstField ESMF_Field with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

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

23 ArrayBundle Class

23.1 Description

The ESMF_ArrayBundle class allows a set of Arrays to be bundled into a single object. The Arrays in an ArrayBundle may be of different type, kind, rank and distribution. Besides ease of use resulting from bundling, the ArrayBundle class offers the opportunity for performance optimization when operating on a bundle of Arrays as a single entity. Communication methods are especially good candidates for performance optimization. Best optimization results are expected for ArrayBundles that contain Arrays that share a common distribution, i.e. DistGrid, and are of same type, kind and rank.
ArrayBundles are one of the data objects that can be added to States, which are used for providing to or receiving data from other Components.

23.2 Use and Examples

Examples of creating, destroying and accessing ArrayBundles and their constituent Arrays are provided in this section, along with some notes on ArrayBundle methods.

23.2.1 Create an ArrayBundle from a list of Arrays

First create a Fortran array of two ESMF_Array objects.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

```
allocate(arrayList(2))
arrayList(1) = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
               rc=rc)
```

```
arrayList(2) = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
               rc=rc)
```

Now the `arrayList` of Arrays can be used to create an ArrayBundle object.

```
arraybundle = ESMF_ArrayBundleCreate(arrayList=arrayList, &
  name="MyArrayBundle", rc=rc)
```

The temporary `arrayList` can be deallocated now. This will not affect the ESMF Array objects. The Array objects must not be deallocated while the ArrayBundle refers to them!

```
deallocate(arrayList)
```

The ArrayBundle object can be printed.

```
call ESMF_ArrayBundlePrint(arraybundle, rc=rc)
```

23.2.2 Access Arrays inside the ArrayBundle

Use `ESMF_ArrayBundleGet()` to determine how many Arrays are stored in an ArrayBundle.

```
call ESMF_ArrayBundleGet(arraybundle, arrayCount=arrayCount, rc=rc)
```

The `arrayCount` can be used to correctly allocate the `arrayList` variable for a second call to `ESMF_ArrayBundleGet()` to gain access to the bundled Array objects.

```
allocate(arrayList(arrayCount))
call ESMF_ArrayBundleGet(arraybundle, arrayList=arraylist, rc=rc)
```

The `arrayList` variable can be used to access the individual Arrays, e.g. to print them.

```
do i=1, arrayCount
  call ESMF_ArrayPrint(arrayList(i), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
enddo
```

23.2.3 Destroy an ArrayBundle and its constituents

The ArrayBundle object can be destroyed.

```
call ESMF_ArrayBundleDestroy(arraybundle, rc=rc)
```

After the ArrayBundle object has been destroyed it is safe to destroy its constituents.

```
call ESMF_ArrayDestroy(arrayList(1), rc=rc)


call ESMF_ArrayDestroy(arrayList(2), rc=rc)


deallocate(arrayList)

call ESMF_DistGridDestroy(distgrid, rc=rc)
```

23.2.4 Halo communication

One of the most fundamental communication pattern in domain decomposition codes is the *halo* operation. The ESMF Array class supports halos by allowing memory for extra elements to be allocated on each DE. See section 24.2.14 for a discussion of the Array level halo operation. The ArrayBundle level extents the Array halo operation to bundles of Arrays.
First create an `ESMF_ArrayBundle` object containing a set of ESMF Arrays.

```
arraybundle = ESMF_ArrayBundleCreate(arrayList=arrayList, &
  name="MyArrayBundle", rc=rc)
```

The ArrayBundle object can be treated as a single entity. The `ESMF_ArrayBundleHaloStore()` call determines the most efficient halo exchange pattern for *all* Arrays that are part of `arraybundle`.

```
call ESMF_ArrayBundleHaloStore(arraybundle=arraybundle, &
  routehandle=haloHandle, rc=rc)
```

The halo exchange pattern stored in `haloHandle` can now be applied to the `arraybundle` object, or any other ArrayBundle that is weakly congruent to the one used during the `ESMF_ArrayBundleHaloStore()` call.

```
call ESMF_ArrayBundleHalo(arraybundle=arraybundle, routehandle=haloHandle, &
  rc=rc)
```

Finally, when no longer needed, the resources held by `haloHandle` need to be returned to the system by calling `ESMF_ArrayBundleHaloRelease()`.

```
call ESMF_ArrayBundleHaloRelease(routehandle=haloHandle, rc=rc)
```

Finally the ArrayBundle object can be destroyed.

```
call ESMF_ArrayBundleDestroy(arraybundle, rc=rc)
```

23.3 Restrictions and Future Work

- **Adding Arrays** to an existing ArrayBundle is currently not supported. In the future this functionality will be provided via the `ESMF_ArrayBundleAdd()` method.
- **Removing Arrays** from an existing ArrayBundle is currently not supported. In the future this functionality will be provided via the `ESMF_ArrayBundleRemove()` method.
- **Non-blocking** ArrayBundle communications option is not yet implemented. In the future this functionality will be provided via the `routesyncflag` option.

23.4 Design and Implementation Notes

The following is a list of implementation specific details about the current ESMF ArrayBundle.

- Implementation language is C++.
- All precomputed communication methods are based on sparse matrix multiplication.

23.5 Class API

23.5.1 ESMF_ArrayBundleAssignment(=) - ArrayBundle assignment

INTERFACE:

```
interface assignment(=)
arraybundle1 = arraybundle2
```

ARGUMENTS:

```
type(ESMF_ArrayBundle) :: arraybundle1
type(ESMF_ArrayBundle) :: arraybundle2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign arraybundle1 as an alias to the same ESMF ArrayBundle object in memory as arraybundle2. If arraybundle2 is invalid, then arraybundle1 will be equally invalid after the assignment.
The arguments are:

arraybundle1 The `ESMF_ArrayBundle` object on the left hand side of the assignment.

arraybundle2 The `ESMF_ArrayBundle` object on the right hand side of the assignment.

23.5.2 ESMF_ArrayBundleOperator(==) - ArrayBundle equality operator

INTERFACE:

```
interface operator(==)
  if (arraybundle1 == arraybundle2) then ... endif
            OR
  result = (arraybundle1 == arraybundle2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArrayBundle), intent(in) :: arraybundle1
type(ESMF_ArrayBundle), intent(in) :: arraybundle2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether arraybundle1 and arraybundle2 are valid aliases to the same ESMF ArrayBundle object in memory. For a more general comparison of two ESMF ArrayBundles, going beyond the simple alias test, the ESMF_ArrayBundleMatch() function (not yet implemented) must be used.
The arguments are:

arraybundle1 The `ESMF_ArrayBundle` object on the left hand side of the equality operation.

arraybundle2 The `ESMF_ArrayBundle` object on the right hand side of the equality operation.

23.5.3 ESMF_ArrayBundleOperator(/=) - ArrayBundle not equal operator

INTERFACE:

```
   interface operator(/=)
     if (arraybundle1 /= arraybundle2) then ... endif
               OR
     result = (arraybundle1 /= arraybundle2)
```

RETURN VALUE:

```
     logical :: result
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in) :: arraybundle1
     type(ESMF_ArrayBundle), intent(in) :: arraybundle2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether arraybundle1 and arraybundle2 are *not* valid aliases to the same ESMF ArrayBundle object in memory. For a more general comparison of two ESMF ArrayBundles, going beyond the simple alias test, the ESMF_ArrayBundleMatch() function (not yet implemented) must be used.
The arguments are:

arraybundle1 The `ESMF_ArrayBundle` object on the left hand side of the non-equality operation.

arraybundle2 The `ESMF_ArrayBundle` object on the right hand side of the non-equality operation.

23.5.4 ESMF_ArrayBundleAdd - Add Arrays to an ArrayBundle

INTERFACE:

```
     subroutine ESMF_ArrayBundleAdd(arraybundle, arrayList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
     type(ESMF_Array),       intent(in)            :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Add Array(s) to an ArrayBundle. It is an error if `arrayList` contains Arrays that match by name Arrays already contained in `arraybundle`.

arraybundle `ESMF_ArrayBundle` to be added to.

arrayList List of `ESMF_Array` objects to be added.

[multiflag] A setting of `.true.` allows multiple items with the same name to be added to `arraybundle`. For `.false.` added items must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add" under `multiflag=.false.` mode, where it is *not* an error if `arrayList` contains items with names that are also found in `arraybundle`. The `arraybundle` 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.

23.5.5 ESMF_ArrayBundleAddReplace - Conditionally add or replace Arrays in an ArrayBundle

INTERFACE:

```
     subroutine ESMF_ArrayBundleAddReplace(arraybundle, arrayList, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
     type(ESMF_Array),       intent(in)            :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Arrays in `arrayList` that do not match any Arrays by name in `arraybundle` are added to the ArrayBundle. Arrays in `arraybundle` that match by name Arrays in `arrayList` are replaced by those Arrays.

arraybundle `ESMF_ArrayBundle` to be manipulated.

arrayList List of `ESMF_Array` objects to be added or used as replacement.

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

23.5.6 ESMF_ArrayBundleCreate - Create an ArrayBundle from a list of Arrays

INTERFACE:

```
   function ESMF_ArrayBundleCreate(arrayList, multiflag, &
     relaxedflag, name, rc)
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Array), intent(in),  optional :: arrayList(:)
     logical,          intent(in),  optional :: multiflag
     logical,          intent(in),  optional :: relaxedflag
     character(len=*), intent(in),  optional :: name
     integer,          intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_ArrayBundle) :: ESMF_ArrayBundleCreate
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_ArrayBundle` object from a list of existing Arrays.
The creation of an ArrayBundle leaves the bundled Arrays unchanged, they remain valid individual objects. An ArrayBundle is a light weight container of Array references. The actual data remains in place, there are no data movements or duplications associated with the creation of an ArrayBundle.

[arrayList] List of `ESMF_Array` objects to be bundled.

[multiflag] A setting of `.true.` allows multiple items with the same name to be added to `arraybundle`. For `.false.` added items must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add" under `multiflag=.false.` mode, where it is *not* an error if `arrayList` contains items with names that are also found in `arraybundle`. The `arraybundle` is left unchanged for these items. For `.false.` this is treated as an error condition. The default setting is `.false.`.

[name] Name of the created `ESMF_ArrayBundle`. A default name is generated if not specified.

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

23.5.7 ESMF_ArrayBundleDestroy - Release resources associated with an ArrayBundle

INTERFACE:

```
subroutine ESMF_ArrayBundleDestroy(arraybundle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroys an `ESMF_ArrayBundle` object. The member Arrays are not touched by this operation and remain valid objects that need to be destroyed individually if necessary.
The arguments are:

arraybundle `ESMF_ArrayBundle` object to be destroyed.

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

23.5.8 ESMF_ArrayBundleGet - Get information about an Array by name

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleGet()
subroutine ESMF_ArrayBundleGetItem(arraybundle, arrayName, &
  array, arrayCount, isPresent, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in)            :: arraybundle
     character(len=*),       intent(in)            :: arrayName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Array),       intent(out), optional :: array
     integer,                intent(out), optional :: arrayCount
     logical,                intent(out), optional :: isPresent
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get information about items that match `arrayName` in ArrayBundle.

arraybundle `ESMF_ArrayBundle` to be queried.

arrayName Specified name.

[array] Upon return holds the requested Array item. It is an error if this argument was specified and there is not exactly one Array item in `arraybundle` that matches `arrayName`.

[arrayCount] Number of Arrays with `arrayName` in `arraybundle`.

[isPresent] Upon return indicates whether Array(s) with `arrayName` exist in `arraybundle`.

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

23.5.9 ESMF_ArrayBundleGet - Get a list of Arrays by name

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleGet()
subroutine ESMF_ArrayBundleGetList(arraybundle, arrayName, arrayList, &
  rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in)            :: arraybundle
     character(len=*),       intent(in)            :: arrayName
     type(ESMF_Array),       intent(out)           :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get the list of Arrays from ArrayBundle that match `arrayName`.

arraybundle `ESMF_ArrayBundle` to be queried.

arrayName Specified name.

[arrayList] List of Arrays in `arraybundle` that match `arrayName`. The argument must be allocated to be at least of size `arrayCount` returned for this `arrayName`.

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

23.5.10 ESMF_ArrayBundleGet - Get general (not Array name specific) information

INTERFACE:

```
    ! Private name; call using ESMF_ArrayBundleGet()
    subroutine ESMF_ArrayBundleGetListAll(arraybundle, arrayCount, &
      arrayList, arrayNameList, name, rc)
```

ARGUMENTS:

```
    type(ESMF_ArrayBundle), intent(in)            :: arraybundle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                intent(out), optional :: arrayCount
    type(ESMF_Array),       intent(out), optional :: arrayList(:)
    character(len=*),       intent(out), optional :: arrayNameList(:)
    character(len=*),       intent(out), optional :: name
    integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get general, i.e. not Array name specific information from the ArrayBundle.

arraybundle `ESMF_ArrayBundle` to be queried.

[arrayCount] Upon return holds the number of Arrays bundled in the ArrayBundle.

[arrayList] Upon return holds a list of Arrays bundled in `arraybundle`. The argument must be allocated to be at least of size `arrayCount`.

[arrayNameList] Upon return holds a list of the names of the Array bundled in `arraybundle`. The argument must be allocated to be at least of size `arrayCount`.

[name] Name of the ArrayBundle object.

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

23.5.11 ESMF_ArrayBundleHalo - Execute an ArrayBundle halo operation

INTERFACE:

```
  subroutine ESMF_ArrayBundleHalo(arraybundle, routehandle, &
    checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)          :: arraybundle
     type(ESMF_RouteHandle), intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional :: checkflag
     integer,                intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed ArrayBundle halo operation for the Arrays in `arrayBundle`.
See `ESMF_ArrayBundleHaloStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.

arraybundle `ESMF_ArrayBundle` containing data to be haloed.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input Array pairs will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

23.5.12 ESMF_ArrayBundleHaloRelease - Release resources associated with an ArrayBundle halo operation

INTERFACE:

```
   subroutine ESMF_ArrayBundleHaloRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with an ArrayBundle halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

23.5.13 ESMF_ArrayBundleHaloStore - Precompute an ArrayBundle halo operation

INTERFACE:

```
     subroutine ESMF_ArrayBundleHaloStore(arraybundle, routehandle, &
       startregion, haloLDepth, haloUDepth, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle),     intent(inout)          :: arraybundle
     type(ESMF_RouteHandle),     intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_StartRegion_Flag),intent(in),  optional :: startregion
     integer,                    intent(in),  optional :: haloLDepth(:)
     integer,                    intent(in),  optional :: haloUDepth(:)
     integer,                    intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store an ArrayBundle halo operation over the data in `arraybundle`. By default, i.e. without specifying `startregion`, `haloLDepth` and `haloUDepth`, all elements in the total Array regions that lie outside the exclusive regions will be considered potential destination elements for halo. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.
Specifying `startregion` allows to change the shape of the effective halo region from the inside. Setting this flag to `ESMF_STARTREGION_COMPUTATIONAL` means that only elements outside the computational region for each Array are considered for potential destination elements for halo. The default is `ESMF_STARTREGION_EXCLUSIVE`.
The `haloLDepth` and `haloUDepth` arguments allow to reduce the extent of the effective halo region. Starting at the region specified by `startregion`, the `haloLDepth` and `haloUDepth` define a halo depth in each direction. Note that the maximum halo region is limited by the total region for each Array, independent of the actual `haloLDepth` and `haloUDepth` setting. The total Array regions are local DE specific. The `haloLDepth` and `haloUDepth` are interpreted as the maximum desired extent, reducing the potentially larger region available for halo.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleHalo()` on any ArrayBundle that is weakly congruent and typekind conform to `arraybundle`. Congruency for ArrayBundles is given by the congruency of its constituents. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
This call is *collective* across the current VM.

arraybundle `ESMF_ArrayBundle` containing data to be haloed. The data in the halo regions may be destroyed by this call.

routehandle Handle to the precomputed Route.

[startregion] The start of the effective halo region on every DE. The default setting is `ESMF_STARTREGION_EXCLUSIVE`, rendering all non-exclusive elements potential halo destination elments. See section 9.41 for a complete list of valid settings.

[haloLDepth] This vector specifies the lower corner of the effective halo region with respect to the lower corner of `startregion`. The size of `haloLDepth` must equal the number of distributed Array dimensions.

[haloUDepth] This vector specifies the upper corner of the effective halo region with respect to the upper corner of `startregion`. The size of `haloUDepth` must equal the number of distributed Array dimensions.

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

23.5.14 ESMF_ArrayBundlePrint - Print ArrayBundle internals

INTERFACE:

```
subroutine ESMF_ArrayBundlePrint(arraybundle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Print internal information of the specified `ESMF_ArrayBundle` object to `stdout`.

The arguments are:

arraybundle `ESMF_ArrayBundle` object.

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

23.5.15 ESMF_ArrayBundleRead - Read Arrays to an ArrayBundle from file(s)

INTERFACE:

```
subroutine ESMF_ArrayBundleRead(arraybundle, file, &
  singleFile, iofmt, rc)
```

ARGUMENTS:

```
  type(ESMF_ArrayBundle), intent(inout)          :: arraybundle
  character(*),           intent(in)             :: file
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                intent(in),  optional  :: singleFile
  type(ESMF_IOFmtFlag),   intent(in),  optional  :: iofmt
  integer,                intent(out), optional  :: rc
```

DESCRIPTION:

Read Array data to an ArrayBundle object from file(s). For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

arraybundle An `ESMF_ArrayBundle` object.

file The name of the file from which ArrayBundle data is read.

[singleFile] A logical flag, the default is .true., i.e., all Arrays in the bundle are stored in one single file. If .false., each Array is stored in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to `ESMF_IOFMT_NETCDF`.

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

23.5.16 ESMF_ArrayBundleRedist - Execute an ArrayBundle redistribution

INTERFACE:

```
   subroutine ESMF_ArrayBundleRedist(srcArrayBundle, dstArrayBundle, &
     routehandle, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in),    optional :: srcArrayBundle
     type(ESMF_ArrayBundle), intent(inout), optional :: dstArrayBundle
     type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),    optional :: checkflag
     integer,                intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed ArrayBundle redistribution from the Arrays in `srcArrayBundle` to the Arrays in `dstArrayBundle`.
This call is *collective* across the current VM.

[srcArrayBundle] `ESMF_ArrayBundle` with source data.

[dstArrayBundle] `ESMF_ArrayBundle` with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input Array pairs will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

23.5.17 ESMF_ArrayBundleRedistRelease - Release resources associated with ArrayBundle redistribution

INTERFACE:

```
   subroutine ESMF_ArrayBundleRedistRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with an ArrayBundle redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

23.5.18 ESMF_ArrayBundleRedistStore - Precompute an ArrayBundle redistribution with local factor argument

INTERFACE:

```
   ! Private name; call using ESMF_ArrayBundleRedistStore()
   subroutine ESMF_ArrayBundleRedistStore<type><kind>(srcArrayBundle, &
     dstArrayBundle, routehandle, factor, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle),   intent(in)            :: srcArrayBundle
     type(ESMF_ArrayBundle),   intent(inout)         :: dstArrayBundle
     type(ESMF_RouteHandle),   intent(inout)         :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)            :: factor
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                  intent(in),  optional :: srcToDstTransposeMap(:)
     integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store an ArrayBundle redistribution operation from `srcArrayBundle` to `dstArrayBundle`. The redistribution between ArrayBundles is defined as the sequence of individual Array redistributions over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.
The effect of this method on ArrayBundles that contain aliased members is undefined.
PETs that specify a `factor` argument must use the <type><kind> overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1.
See the description of method `ESMF_ArrayRedistStore()` for the definition of the Array based operation.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleRedist()` on any pair of ArrayBundles that are weakly congruent and typekind conform with the Arrays contained in `srcArrayBundle` and `dstArrayBundle`. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multipy source data.

[srcToDstTransposeMap] List with as many entries as there are dimensions in the Arrays in `srcArrayBundle`. Each entry maps the corresponding source Array dimension against the specified destination Array dimension. Mixing of distributed and undistributed dimensions is supported.

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

23.5.19 ESMF_ArrayBundleRedistStore - Precompute an ArrayBundle redistribution without local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleRedistStore()
subroutine ESMF_ArrayBundleRedistStoreNF(srcArrayBundle, dstArrayBundle, &
  routehandle, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
    type(ESMF_ArrayBundle), intent(in)            :: srcArrayBundle
    type(ESMF_ArrayBundle), intent(inout)         :: dstArrayBundle
    type(ESMF_RouteHandle), intent(inout)         :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                intent(in),  optional :: srcToDstTransposeMap(:)
    integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store an ArrayBundle redistribution operation from `srcArrayBundle` to `dstArrayBundle`. The redistribution between ArrayBundles is defined as the sequence of individual Array redistributions over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.
The effect of this method on ArrayBundles that contain aliased members is undefined.
PETs that specify a `factor` argument must use the <type><kind> overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1.
See the description of method `ESMF_ArrayRedistStore()` for the definition of the Array based operation.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleRedist()` on any pair of ArrayBundles that are weakly congruent and typekind conform with the Arrays contained in `srcArrayBundle` and `dstArrayBundle`. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap] List with as many entries as there are dimensions in the Arrays in `srcArrayBundle`. Each entry maps the corresponding source Array dimension against the specified destination Array dimension. Mixing of distributed and undistributed dimensions is supported.

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

23.5.20 ESMF_ArrayBundleRemove - Remove Arrays from ArrayBundle

INTERFACE:

```
subroutine ESMF_ArrayBundleRemove(arraybundle, arrayNameList, &
  multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
     character(len=*),       intent(in)            :: arrayNameList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Remove Array(s) by name from ArrayBundle. In the relaxed setting it is *not* an error if `arrayNameList` contains names that are not found in `arraybundle`.

arraybundle `ESMF_ArrayBundle` from which to remove items.

arrayNameList List of items to remove.

[multiflag] A setting of `.true.` allows multiple Arrays with the same name to be removed from `arraybundle`. For `.false.`, items to be removed must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "remove" where it is *not* an error if `arrayNameList` contains item names that are not found in `arraybundle`. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "remove" also covers the case where there are multiple items in `arraybundle` that match a single entry in `arrayNameList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

23.5.21 ESMF_ArrayBundleReplace - Replace Arrays in ArrayBundle

INTERFACE:

```
subroutine ESMF_ArrayBundleReplace(arraybundle, arrayList, &
  multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
     type(ESMF_Array),       intent(in)            :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Replace Array(s) by name in ArrayBundle. In the relaxed setting it is not an error if `arrayList` contains Arrays that do not match by name any item in `arraybundle`. These Arrays are simply ignored in this case.

arraybundle `ESMF_ArrayBundle` in which to replace items.

arrayList List of items to replace.

[multiflag] A setting of `.true.` allows multiple items with the same name to be replaced in `arraybundle`. For `.false.`, items to be replaced must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "replace" where it is *not* an error if `arrayList` contains items with names that are not found in `arraybundle`. These items in `arrayList` are ignored in the relaxed mode. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "replace" also covers the case where there are multiple items in `arraybundle` that match a single entry by name in `arrayList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

23.5.22 ESMF_ArrayBundleSMM - Execute an ArrayBundle sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_ArrayBundleSMM(srcArrayBundle, dstArrayBundle, &
     routehandle, zeroregion, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in),    optional :: srcArrayBundle
     type(ESMF_ArrayBundle), intent(inout), optional :: dstArrayBundle
     type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Region_Flag), intent(in),    optional :: zeroregion
     logical,                intent(in),    optional :: checkflag
     integer,                intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed ArrayBundle sparse matrix multiplication from the Arrays in `srcArrayBundle` to the Arrays in `dstArrayBundle`.
This call is *collective* across the current VM.

[srcArrayBundle] `ESMF_ArrayBundle` with source data.

[dstArrayBundle] `ESMF_ArrayBundle` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in all Arrays in `dstArrayBundle` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in the Arrays in `dstArrayBundle` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Arrays that will be updated by the sparse matrix multiplication. See section 9.36 for a complete list of valid settings.

[checkflag] If set to `.TRUE.` the input Array pairs will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

23.5.23 ESMF_ArrayBundleSMMRelease - Release resources associated with ArrayBundle sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_ArrayBundleSMMRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with an ArrayBundle sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

23.5.24 ESMF_ArrayBundleSMMStore - Precompute an ArrayBundle sparse matrix multiplication with local factors

INTERFACE:

```
   ! Private name; call using ESMF_ArrayBundleSMMStore()
   subroutine ESMF_ArrayBundleSMMStore<type><kind>(srcArrayBundle, &
     dstArrayBundle, routehandle, factorList, factorIndexList, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle),           intent(in)    :: srcArrayBundle
     type(ESMF_ArrayBundle),           intent(inout) :: dstArrayBundle
     type(ESMF_RouteHandle),           intent(inout) :: routehandle
     <type>(ESMF_KIND_<kind>), target, intent(in)    :: factorList(:)
     integer,                          intent(in)    :: factorIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                          intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store an ArrayBundle sparse matrix multiplication operation from `srcArrayBundle` to `dstArrayBundle`. The sparse matrix multiplication between ArrayBundles is defined as the sequence of individual Array sparse matrix multiplications over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.
The effect of this method on ArrayBundles that contain aliased members is undefined.
PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
See the description of method `ESMF_ArraySMMStore()` for the definition of the Array based operation.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleSMM()` on any pair of ArrayBundles that are weakly congruent and typekind conform with the Arrays contained in `srcArrayBundle` and `dstArrayBundle`. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the source Array while `factorIndexList(2,:)` specifies the sequence index of the destination element in the destination Array. For this format to be a valid option source and destination Arrays must have matching number of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identiy matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the `factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the source Array. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the destination Array.

See section 24.2.17 for details on the definition of Array *sequence indices* and *tensor sequence indices*.

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

23.5.25 ESMF_ArrayBundleSMMStore - Precompute an ArrayBundle sparse matrix multiplication without local factors

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleSMMStore()
subroutine ESMF_ArrayBundleSMMStoreNF(srcArrayBundle, dstArrayBundle, &
  routehandle, rc)
```

ARGUMENTS:

```
    type(ESMF_ArrayBundle),     intent(in)            :: srcArrayBundle
    type(ESMF_ArrayBundle),     intent(inout)         :: dstArrayBundle
    type(ESMF_RouteHandle),     intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                    intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store an ArrayBundle sparse matrix multiplication operation from `srcArrayBundle` to `dstArrayBundle`. The sparse matrix multiplication between ArrayBundles is defined as the sequence of individual Array sparse matrix multiplications over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.
The effect of this method on ArrayBundles that contain aliased members is undefined.
PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
See the description of method `ESMF_ArraySMMStore()` for the definition of the Array based operation.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleSMM()` on any pair of ArrayBundles that are weakly congruent and typekind conform with the Arrays contained in `srcArrayBundle` and `dstArrayBundle`. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

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

23.5.26 ESMF_ArrayBundleWrite - Write the Arrays into a file

INTERFACE:

```
   subroutine ESMF_ArrayBundleWrite(arraybundle, file, &
     singleFile, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in)              :: arraybundle
     character(*),           intent(in)              :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional  :: singleFile
     integer,                intent(in),   optional  :: timeslice
     type(ESMF_IOFmtFlag),   intent(in),   optional  :: iofmt
     integer,                intent(out),  optional  :: rc
```

DESCRIPTION:

Write the Arrays into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

arraybundle An `ESMF_ArrayBundle` object.

file The name of the output file to which array bundle data is written.

[singleFile] A logical flag, the default is .true., i.e., all arrays in the bundle are written in one single file. If .false., each array will be written in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[timeslice] Some IO formats (e.g. NetCDF) support the output of data in form of time slices. The `timeslice` argument provides access to this capability. Usage of this feature requires that the first slice is written with a positive `timeslice` value, and that subsequent slices are written with a `timeslice` argument that increments by one each time. By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file.

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to `ESMF_IOFMT_NETCDF`.

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

24 Array Class

24.1 Description

The Array class is an alternative to the Field class for representing distributed, structured data. Unlike Fields, which are built to carry grid coordinate information, Arrays can only carry information about the *indices* associated with grid cells. Since they do not have coordinate information, Arrays cannot be used to calculate interpolation weights. However, if the user can supply interpolation weights, the Array sparse matrix multiply operation can be used to apply the weights and transfer data to the new grid. Arrays can also perform redistribution, scatter, and gather communication operations.
Like Fields, Arrays can be added to a State and used in inter-Component data communications. Arrays can also be grouped together into ArrayBundles so that collective operations can be performed on the whole group. One motivation for this is convenience; another is the ability to schedule optimized, collective data transfers.
From a technical standpoint, the ESMF Array class is an index space based, distributed data storage class. It provides DE-local memory allocations within DE-centric index regions and defines the relationship to the index space described by the ESMF DistGrid. The Array class offers common communication patterns within the index space formalism. As part of the ESMF index space layer, Array has close relationship to the DistGrid and DELayout classes.

24.2 Use and Examples

An ESMF_Array is a distributed object that must exist on all PETs of the current context. Each PET-local instance of an Array object contains memory allocations for all PET-local DEs. There may be 0, 1, or more DEs per PET and the number of DEs per PET can differ between PETs for the same Array object. Memory allocations may be provided for each PET by the user during Array creation or can be allocated as part of the Array create call. Many of the concepts of the proposed ESMF_Array class are illustrated by the following examples.

24.2.1 Array from native Fortran array with 1 DE per PET

The create call of the ESMF_Array class has been overloaded extensively to facilitate the need for generality while keeping simple cases simple. The following program demonstrates one of the simpler cases, where existing local Fortran arrays are to be used to provide the PET-local memory allocations for the Array object.

```
program ESMF_ArrayFarrayEx

  use ESMF

  implicit none
```

The Fortran language provides a variety of ways to define and allocate an array. Actual Fortran array objects must either be explicit-shape or deferred-shape. In the first case the memory allocation and deallocation is automatic from the user's perspective and the details of the allocation (static or dynamic, heap or stack) are left to the compiler. (Compiler flags may be used to control some of the details). In the second case, i.e. for deferred-shape actual objects, the array definition must include the pointer or allocatable attribute and it is the user's responsibility to allocate memory. While it is also the user's responsibility to deallocate memory for arrays with the pointer attribute the compiler will automatically deallocate allocatable arrays under certain circumstances defined by the Fortran standard. The ESMF_ArrayCreate() interface has been written to accept native Fortran arrays of any flavor as a means to allow user-controlled memory management. The Array create call will check on each PET if sufficient memory has been provided by the specified Fortran arrays and will indicate an error if a problem is detected. However, the Array create call cannot validate the lifetime of the provided memory allocations. If, for instance, an Array object was created in a subroutine from an automatic explicit-shape array or an allocatable array, the memory allocations referenced by the Array object will be automatically deallocated on return from the subroutine unless provisions are made by the application writer to prevent such behavior. The Array object cannot control when memory that has been provided by the user during Array creation becomes deallocated, however, the Array will indicate an error if its memory references have been invalidated.
The easiest, portable way to provide safe native Fortran memory allocations to Array create is to use arrays with the pointer attribute. Memory allocated for an array pointer will not be deallocated automatically. However, in this case the possibility of memory leaks becomes an issue of concern. The deallocation of memory provided to an Array in form of a native Fortran allocation will remain the users responsibility.
None of the concerns discussed above are an issue in this example where the native Fortran array farray is defined in the main program. All different types of array memory allocation are demonstrated in this example. First farrayE is defined as a 2D explicit-shape array on each PET which will automatically provide memory for 10×10 elements.

```
  ! local variables
  real(ESMF_KIND_R8)       :: farrayE(10,10)  ! explicit shape Fortran array
```

Then an allocatable array farrayA is declared which will be used to show user-controlled dynamic memory allocation.

```
  real(ESMF_KIND_R8), allocatable :: farrayA(:,:) ! allocatable Fortran array
```

Finally an array with pointer attribute farrayP is declared, also used for user-controlled dynamic memory allocation.

```
real(ESMF_KIND_R8), pointer :: farrayP(:,:)   ! Fortran array pointer
```

A matching array pointer must also be available to gain access to the arrays held by an Array object.

```
real(ESMF_KIND_R8), pointer :: farrayPtr(:,:) ! matching Fortran array ptr
type(ESMF_DistGrid)         :: distgrid       ! DistGrid object
type(ESMF_Array)            :: array          ! Array object
integer                     :: rc
```

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

On each PET `farrayE` can be accessed directly to initialize the entire PET-local array.

```
farrayE = 12.45d0 ! initialize to some value
```

In order to create an Array object a DistGrid must first be created that describes the total index space and how it is decomposed and distributed. In the simplest case only the `minIndex` and `maxIndex` of the total space must be provided.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

This example is assumed to run on 4 PETs. The default 2D decomposition will then be into 4 x 1 DEs as to ensure 1 DE per PET.
Now the Array object can be created using the `farrayE` and the DistGrid just created.

```
array = ESMF_ArrayCreate(farray=farrayE, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

The 40 x 10 index space defined by the `minIndex` and `maxIndex` arguments paired with the default decomposition will result in the following distributed Array.

```
+---------------------------> 2nd dimension
|   (1,1)-------+
|      |        |
|      |   DE 0 |   <--- farray on PET 0
|      |        |
|      +------(10,10)
|  (11,1)-------+
|      |        |
|      |   DE 1 |   <--- farray on PET 1
|      |        |
|      +------(20,10)
|  (21,1)-------+
|      |        |
|      |   DE 2 |   <--- farray on PET 2
|      |        |
|      +------(30,10)
```

```
         |  (31,1)-------+
         |      |        |
         |      |   DE 3 |   <--- farray on PET 3
         |      |        |
         |      +------(40,10)
         v
       1st dimension
```

Providing `farrayE` during Array creation does not change anything about the actual `farrayE` object. This means that each PET can use its local `farrayE` directly to access the memory referenced by the Array object.

```
print *, farrayE
```

Another way of accessing the memory associated with an Array object is to use `ArrayGet()` to obtain an Fortran pointer that references the PET-local array.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)


print *, farrayPtr
```

Finally the Array object must be destroyed. The PET-local memory of the `farrayE`s will remain in user control and will not be altered by `ArrayDestroy()`.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

Since the memory allocation for each `farrayE` is automatic there is nothing more to do.
The interaction between `farrayE` and the Array class is representative also for the two other cases `farrayA` and `farrayP`. The only difference is in the handling of memory allocations.

```
allocate(farrayA(10,10))    ! user controlled allocation
farrayA = 23.67d0           ! initialize to some value
array = ESMF_ArrayCreate(farray=farrayA, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)


print *, farrayA            ! print PET-local farrayA directly
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)! obtain array pointer
print *, farrayPtr          ! print PET-local piece of Array through pointer
call ESMF_ArrayDestroy(array, rc=rc) ! destroy the Array
deallocate(farrayA)         ! user controlled de-allocation
```

The `farrayP` case is identical.

```
allocate(farrayP(10,10))    ! user controlled allocation
farrayP = 56.81d0           ! initialize to some value
array = ESMF_ArrayCreate(farray=farrayP, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

```
  print *, farrayP            ! print PET-local farrayA directly
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)! obtain array pointer
  print *, farrayPtr          ! print PET-local piece of Array through pointer
  call ESMF_ArrayDestroy(array, rc=rc) ! destroy the Array
  deallocate(farrayP)         ! user controlled de-allocation
```

To wrap things up the DistGrid object is destroyed and ESMF can be finalized.

```
  call ESMF_DistGridDestroy(distgrid, rc=rc) ! destroy the DistGrid
```

```
  call ESMF_Finalize(rc=rc)
```

```
end program
```

24.2.2 Array from native Fortran array with extra elements for halo or padding

The example of the previous section showed how easy it is to create an Array object from existing PET-local Fortran arrays. The example did, however, not define any halo elements around the DE-local regions. The following code demonstrates how an Array object with space for a halo can be set up.

```
program ESMF_ArrayFarrayHaloEx

  use ESMF

  implicit none
```

The allocatable array `farrayA` will be used to provide the PET-local Fortran array for this example.

```
  ! local variables
  real(ESMF_KIND_R8), allocatable :: farrayA(:,:) ! allocatable Fortran array
  real(ESMF_KIND_R8), pointer :: farrayPtr(:,:)   ! matching Fortran array ptr
  type(ESMF_DistGrid)         :: distgrid         ! DistGrid object
  type(ESMF_Array)            :: array            ! Array object
  integer                     :: rc, i, j
  real                        :: localSum
```

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

The Array is to cover the exact same index space as in the previous example. Furthermore decomposition and distribution are also kept the same. Hence the same DistGrid object will be created and it is expected to execute this example with 4 PETs.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

This DistGrid describes a 40 x 10 index space that will be decomposed into 4 DEs when executed on 4 PETs, associating 1 DE per PET. Each DE-local exclusive region contains 10 x 10 elements. The DistGrid also stores and provides information about the relationship between DEs in index space, however, DistGrid does not contain information about halos. Arrays contain halo information and it is possible to create multiple Arrays covering the same index space with identical decomposition and distribution using the same DistGrid object, while defining different, Array-specific halo regions.
The extra memory required to cover the halo in the Array object must be taken into account when allocating the PET-local `farrayA` arrays. For a halo of 2 elements in each direction the following allocation will suffice.

```
allocate(farrayA(14,14))    ! Fortran array with halo: 14 = 10 + 2 * 2
```

The `farrayA` can now be used to create an Array object with enough space for a two element halo in each direction. The Array creation method checks for each PET that the local Fortran array can accommodate the requested regions.
The default behavior of ArrayCreate() is to center the exclusive region within the total region. Consequently the following call will provide the 2 extra elements on each side of the exclusive 10 x 10 region without having to specify any additional arguments.

```
array = ESMF_ArrayCreate(farray=farrayA, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

The exclusive Array region on each PET can be accessed through a suitable Fortran array pointer. See section 24.2.6 for more details on Array regions.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)
```

Following Array bounds convention, which by default puts the beginning of the exclusive region at (1, 1, ...), the following loop will add up the values of the local exclusive region for each DE, regardless of how the bounds were chosen for the original PET-local `farrayA` arrays.

```
localSum = 0.
do j=1, 10
  do i=1, 10
    localSum = localSum + farrayPtr(i, j)
  enddo
enddo
```

Elements with i or j in the [-1,0] or [11,12] ranges are located outside the exclusive region and may be used to define extra computational points or halo operations.
Cleanup and shut down ESMF.

```
call ESMF_ArrayDestroy(array, rc=rc)
deallocate(farrayA)
call ESMF_DistGridDestroy(distgrid, rc=rc)


call ESMF_Finalize(rc=rc)

end program
```

24.2.3 Array from ESMF_LocalArray

Alternative to the direct usage of Fortran arrays during Array creation it is also possible to first create an ESMF_LocalArray and create the Array from it. While this may seem more burdensome for the 1 DE per PET cases discussed in the previous sections it allows a straight forward generalization to the multiple DE per PET case. The following example first recaptures the previous example using an ESMF_LocalArray and then expands to the multiple DE per PET case.

```
program ESMF_ArrayLarrayEx

  use ESMF

  implicit none
```

The current ESMF_LocalArray interface requires Fortran arrays to be defined with pointer attribute.

```
  ! local variables
  real(ESMF_KIND_R8), pointer :: farrayP(:,:)   ! Fortran array pointer
  real(ESMF_KIND_R8), pointer :: farrayPtr(:,:) ! matching Fortran array ptr
  type(ESMF_LocalArray)       :: larray         ! ESMF_LocalArray object
  type(ESMF_LocalArray)       :: larrayRef      ! ESMF_LocalArray object
  type(ESMF_DistGrid)         :: distgrid       ! DistGrid object
  type(ESMF_Array)            :: array          ! Array object
  integer                     :: rc, i, j, de
  real                        :: localSum
  type(ESMF_LocalArray), allocatable :: larrayList(:) ! LocalArray object list
  type(ESMF_LocalArray), allocatable :: larrayRefList(:)!LocalArray obj. list

  type(ESMF_VM):: vm
  integer:: localPet, petCount



  call ESMF_Initialize(vm=vm, defaultlogfilename="ArrayLarrayEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  call ESMF_VMGet(vm, localPet=localPet, petCount=petCount, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

  if (petCount /= 4) then
    finalrc = ESMF_FAILURE
    goto 10
  endif
```

DistGrid and array allocation remains unchanged.

```
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)


  allocate(farrayP(14,14))    ! allocate Fortran array on each PET with halo
```

Now instead of directly creating an Array object using the PET-local farrayPs an ESMF_LocalArray object will be created on each PET.

```
larray = ESMF_LocalArrayCreate(farrayP, &
             datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
```

The Array object can now be created from `larray`. The Array creation method checks for each PET that the LocalArray can accommodate the requested regions.

```
array = ESMF_ArrayCreate(localarrayList=(/larray/), distgrid=distgrid, rc=rc)
```

Once created there is no difference in how the Array object can be used. The exclusive Array region on each PET can be accessed through a suitable Fortran array pointer as before.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)
```

Alternatively it is also possible (independent of how the Array object was created) to obtain the reference to the array allocation held by Array in form of an `ESMF_LocalArray` object. The `farrayPtr` can then be extracted using LocalArray methods.

```
call ESMF_ArrayGet(array, localarray=larrayRef, rc=rc)
```

```
call ESMF_LocalArrayGet(larrayRef, farrayPtr, rc=rc)
```

Either way the `farrayPtr` reference can be used now to add up the values of the local exclusive region for each DE. The following loop works regardless of how the bounds were chosen for the original PET-local `farrayP` arrays and consequently the PET-local `larray` objects.

```
localSum = 0.
do j=1, 10
  do i=1, 10
    localSum = localSum + farrayPtr(i, j)
  enddo
enddo
print *, "localSum=", localSum
```

Cleanup.

```
call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_LocalArrayDestroy(larray, rc=rc)
deallocate(farrayP)   ! use the pointer that was used in allocate statement
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

While the usage of LocalArrays is unnecessarily cumbersome for 1 DE per PET Arrays, it provides a straight forward path for extending the interfaces to multiple DEs per PET.
In the following example a 8 x 8 index space will be decomposed into 2 x 4 = 8 DEs. The situation is captured by the following DistGrid object.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/8,8/), &
  regDecomp=(/2,4/), rc=rc)
```

The `distgrid` object created in this manner will contain 8 DEs no matter how many PETs are available during execution. Assuming an execution on 4 PETs will result in the following distribution of the decomposition.

```
 +-----------------------------------> 2nd dimension
 |  (1,1)
 |    +-----------+-----------+-----------+-----------+
 |    | DE0, PET0 | DE2, PET1 | DE4, PET2 | DE6, PET3 |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    |           |           |           |           |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    |           |           |           |           |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    |           |           |           |           |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    +-----------+-----------+-----------+-----------+
 |    | DE1, PET0 | DE3, PET1 | DE5, PET2 | DE7, PET3 |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    |           |           |           |           |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    |           |           |           |           |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    |           |           |           |           |
 |    |  *    *   |  *    *   |  *    *   |  *    *   |
 |    +-----------+-----------+-----------+-----------+
 |                                                  (8,8)
 v
1st dimension
```

Obviously each PET is associated with 2 DEs. Each PET must allocate enough space for *all* its DEs. This is done by allocating as many DE-local arrays as there are DEs on the PET. The reference to these array allocations is passed into ArrayCreate via a LocalArray list argument that holds as many elements as there are DEs on the PET. Here each PET must allocate for two DEs.

```
allocate(larrayList(2))   ! 2 DEs per PET
allocate(farrayP(4, 2))   ! without halo each DE is of size 4 x 2
farrayP = 123.456d0
larrayList(1) = ESMF_LocalArrayCreate(farrayP, &
  datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc) !1st DE
allocate(farrayP(4, 2))   ! without halo each DE is of size 4 x 2
farrayP = 456.789d0
larrayList(2) = ESMF_LocalArrayCreate(farrayP, &
  datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc) !2nd DE
```

Notice that it is perfectly fine to *re*-use `farrayP` for all allocations of DE-local Fortran arrays. The allocated memory can be deallocated at the end using the array pointer contained in the `larrayList`.
With this information an Array object can be created. The `distgrid` object indicates 2 DEs for each PET and ArrayCreate() expects to find two LocalArray elements in `larrayList`.

```
array = ESMF_ArrayCreate(localarrayList=larrayList, distgrid=distgrid, rc=rc)
```

Usage of a LocalArray list is the only way to provide a list of variable length of Fortran array allocations to ArrayCreate() for each PET. The `array` object created by the above call is an ESMF distributed object. As such it must follow the ESMF convention that requires that the call to `ESMF_ArrayCreate()` must be issued in unison by all PETs of the current context. Each PET only calls ArrayCreate() once, even if there are multiple DEs per PET.
The ArrayGet() method provides access to the list of LocalArrays on each PET.

```
allocate(larrayRefList(2))
call ESMF_ArrayGet(array, localarrayList=larrayRefList, rc=rc)
```

Finally, access to the actual Fortran pointers is done on a per DE basis. Generally each PET will loop over its DEs.

```
do de=1, 2
  call ESMF_LocalArrayGet(larrayRefList(de), farrayPtr, rc=rc)
  localSum = 0.
  do j=1, 2
    do i=1, 4
      localSum = localSum + farrayPtr(i, j)
    enddo
  enddo
  print *, "localSum=", localSum
enddo
```

Note: If the VM associates multiple PEs with a PET the application writer may decide to use OpenMP loop parallelization on the `de` loop.
Cleanup requires that the PET-local deallocations are done before the pointers to the actual Fortran arrays are lost. Notice that `larrayList` is used to obtain the pointers used in the deallocate statement. Pointers obtained from the `larrayRefList`, while pointing to the same data, *cannot* be used to deallocated the array allocations!

```
do de=1, 2
  call ESMF_LocalArrayGet(larrayList(de), farrayPtr, rc=rc)
  deallocate(farrayPtr)
  call ESMF_LocalArrayDestroy(larrayList(de), rc=rc)
enddo
deallocate(larrayList)
deallocate(larrayRefList)
call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

With that ESMF can be shut down cleanly.

```
call ESMF_Finalize(rc=rc)
```

```
end program
```

24.2.4 Create Array with automatic memory allocation

In the examples of the previous sections the user provided memory allocations for each of the DE-local regions for an Array object. The user was able to use any of the Fortran methods to allocate memory, or go through the `ESMF_LocalArray` interfaces to obtain memory allocations before passing them into ArrayCreate(). Alternatively ESMF offers methods that handle Array memory allocations inside the library.
As before, to create an `ESMF_Array` object an `ESMF_DistGrid` must be created. The DistGrid object holds information about the entire index space and how it is dcomposed into DE-local exclusive regions. The following line of code creates a DistGrid for a 5x5 global index space that is decomposed into 2 x 3 = 6 DEs.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

The following is a representation of the index space and its decompositon into DEs. Each asterix (*) represents a single element.

```
 +---------------------------------------> 2nd dimension
 |  (1,1)
 |    +-----------+-----------+------+
 |    | DE 0      | DE 2      | DE 4 |
 |    |           |           |      |
 |    |  *    *   |  *    *   |  *   |
 |    |           |           |      |
 |    |  *    *   |  *    *   |  *   |
 |    |           |           |      |
 |    |  *    *   |  *    *   |  *   |
 |    +-----------+-----------+------+
 |    |           |           |      |
 |    | DE 1      | DE 3      | DE 5 |
 |    |           |           |      |
 |    |  *    *   |  *    *   |  *   |
 |    |           |           |      |
 |    |  *    *   |  *    *   |  *   |
 |    +-----------+-----------+------+
 |                                 (5,5)
 v
1st dimension
```

Besides the DistGrid it is the *type*, *kind* and *rank* information, "tkr" for short, that is required to create an Array object. It turns out that the rank of the Array object is fully determined by the DistGrid and other (optional) arguments passed into ArrayCreate(), so that explicit specification of the Array rank is redundant.
The simplest way to supply the type and kind information of the Array is directly through the `typekind` argument. Here a double precision Array is created on the previously created DistGrid. Since no other arguments are specified that could alter the rank of the Array it becomes equal to the dimCount of the DistGrid, i.e a 2D Array is created on top of the DistGrid.

```
  array = ESMF_ArrayCreate(typekind=ESMF_TYPEKIND_R8, distgrid=distgrid, rc=rc)
```

The different methods on how an Array object is created have no effect on the use of `ESMF_ArrayDestroy()`.

```
  call ESMF_ArrayDestroy(array, rc=rc)
```

Alternatively the same Array can be created specifying the "tkr" information in form of an ArraySpec variable. The ArraySpec explicitly contains the Array rank and thus results in an overspecification on the ArrayCreate() interface. ESMF checks all input information for consistency and returns appropriate error codes in case any inconsistencies are found.

```
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)

  array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

The Array object created by the above call is an ESMF distributed object. As such it must follow the ESMF convention that requires that the call to `ESMF_ArrayCreate()` must be issued in unison by all PETs of the current context.

24.2.5 Native language memory access

Access to the data held inside an ESMF Array object is provided through native language objects. Specifically, the `farrayPtr` argument returned by the `ESMF_ArrayGet()` method is a Fortran array pointer that can be used do access the PET-local data inside the Array object.
Many applications work in the 1 DE per PET mode, i.e. there is only a single DE on each PET. The Array class does not assume this special case, instead it supports multiple separate memory allocations on each PET. The number of such PET-local allocations is given by the `localDeCount` of the underlying DistGrid. Access to the DE-local memory allocations in this general case requires a loop over `localDeCount`.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)


do de=0, localDeCount-1
  call ESMF_ArrayGet(array, farrayPtr=myFarray, localDe=de, rc=rc)


  ! use myFarray to access local DE data
enddo
```

The 1 DE per PET case is so common that the ESMF Array provides simplified support for it. In this case the `ESMF_ArrayGet()` can be called without specifying `localDe` to access the unique PET-local `farrayPtr`. An error will be returned if `localDe` was omitted for an Array that holds multiple DEs per PET.
Besides direct access to the DE-local memory allocation through the Fortran array pointer, the Array can also be queried for a list of PET-local LocalArray objects. See section 24.2.3 for more on LocalArray usage in Array. In most cases this approach is less convenient than the direct `farrayPtr` method, because it adds an extra object level between the Array and the native language array. Further, the 1 DE per PET case is not treated in a simplified manner.

```
allocate(larrayList(localDeCount))
call ESMF_ArrayGet(array, localarrayList=larrayList, rc=rc)


do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myFarray, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)


  ! use myFarray to access local DE data
enddo
```

24.2.6 Regions and default bounds

Each `ESMF_Array` object is decomposed into DEs as specified by the associated `ESMF_DistGrid` object. Each piece of this decomposition, i.e. each DE, holds a chunk of the Array data in its own local piece of memory. The details of the Array decomposition are described in the following paragraphs.
At the center of the Array decomposition is the `ESMF_DistGrid` class. The DistGrid object specified during Array creation contains three essential pieces of information:

- The extent and topology of the global domain covered by the Array object in terms of indexed elements. The total extent may be a composition or tilework of smaller logically rectangular (LR) domain pieces or tiles.
- The decomposition of the entire domain into "element exclusive" DE-local LR chunks. *Element exclusive* means that there is no element overlap between DE-local chunks. This, however, does not exclude degeneracies on edge boundaries for certain topologies (e.g. bipolar).
- The layout of DEs over the available PETs and thus the distribution of the Array data.

Each element of an Array is associated with a *single* DE. The union of elements associated with a DE, as defined by the DistGrid above, corresponds to a LR chunk of index space, called the *exclusive region* of the DE.
There is a hierarchy of four regions that can be identified for each DE in an Array object. Their definition and relationship to each other is as follows:

- *Interior Region*: Region that only contains local elements that are *not* mapped into the halo of any other DE. The shape and size of this region for a particular DE depends non-locally on the halos defined by other DEs and may change during computation as halo operations are precomputed and released. Knowledge of the interior elements may be used to improve performance by overlapping communications with ongoing computation for a DE.
- *Exclusive Region*: Elements for which a DE claims exclusive ownership. Practically this means that the DE will be the sole source for these elements in halo and reduce operations. There are exceptions to this in some topologies. The exclusive region includes all elements of the interior region.
- *Computational Region*: Region that can be set arbitrarily within the bounds of the total region (defined next). The typical use of the computation region is to define bounds that only include elements that are updated by a DE-local computation kernel. The computational region does not need to include all exclusive elements and it may also contain elements that lie outside the exclusive region.
- *Total (Memory) Region*: Total of all DE-locally allocated elements. The size and shape of the total memory region must accommodate the union of exclusive and computational region but may contain additional elements. Elements outside the exclusive region may overlap with the exclusive region of another DE which makes them potential receivers for Array halo operations. Elements outside the exclusive region that do not overlap with the exclusive region of another DE can be used to set boundary conditions and/or serve as extra memory padding.

```
+-totalLBound(:)-----------------------------------+
|\                                                 |
| \ <--- totalLWidth(:)                            |
|  \                                               |
|   +-computationalLBound(:)--------------------+   |
|   |\                                          |   |
|   | \ <--- computationalLWidth(:)             |   |
|   |  \                                        |   |
|   |   +-exclusiveLBound(:)--------------+    |   |
|   |   |                                 |    |   |
|   |   |     +------+       +-----+      |    |   |
|   |   |     |      |       |     |      |    |   |
|   |   |     |      +-------+     |      |    |   |
|   |   |     | "Interior Region"  |      |    |   |
|   |   |     +------+             |      |    |   |
|   |   |            |             |      |    |   |
|   |   |            +-------------+      |    |   |
|   |   |                                 |    |   |
|   |   | "Exclusive Region"              |    |   |
|   |   +--------------exclusiveUBound(:)-+    |   |
|   |                                      \   |   |
|   |            computationalUWidth(:) --> \  |   |
|   |                                        \ |   |
|   | "Computational Region"                  \|   |
|   +-------------------computationalUBound(:)-+   |
|                                               \  |
|                              totalUWidth(:) -> \ |
| "Total Region"                                  \|
+---------------------------------- totalUBound(:)-+
```

With the following definitions:

```
computationalLWidth(:) = exclusiveLBound(:) - computationalLBound(:)
computationalUWidth(:) = computationalUBound(:) - exclusiveUBound(:)
```

and

```
totalLWidth(:) = exclusiveLBound(:) - totalLBound(:)
totalUWidth(:) = totalUBound(:) - exclusiveUBound(:)
```

The *exclusive region* is determined during Array creation by the DistGrid argument. Optional arguments may be used to specify the *computational region* when the Array is created, by default it will be set equal to the exclusive region. The *total region*, i.e. the actual memory allocation for each DE, is also determined during Array creation. When creating the Array object from existing Fortran arrays the total region is set equal to the memory provided by the Fortran arrays. Otherwise the default is to allocate as much memory as is needed to accommodate the union of the DE-local exclusive and computational region. Finally it is also possible to use optional arguments to the ArrayCreate() call to specify the total region of the object explicitly.
The `ESMF_ArrayCreate()` call checks that the input parameters are consistent and will result in an Array that fulfills all of the above mentioned requirements for its DE-local regions.
Once an Array object has been created the exclusive and total regions are fixed. The computational region, however, may be adjusted within the limits of the total region using the `ArraySet()` call.
The *interior region* is very different from the other regions in that it cannot be specified. The *interior region* for each DE is a *consequence* of the choices made for the other regions collectively across all DEs into which an Array object is decomposed. An Array object can be queried for its DE-local *interior regions* as to offer additional information to the user necessary to write more efficient code.
By default the bounds of each DE-local *total region* are defined as to put the start of the DE-local *exclusive region* at the "origin" of the local index space, i.e. at `(1, 1, ..., 1)`. With that definition the following loop will access each element of the DE-local memory segment for each PET-local DE of the Array object used in the previous sections and print its content.

```
do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myFarray, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
  do i=1, size(myFarray, 1)
    do j=1, size(myFarray, 2)
      print *, "PET-local DE=", de, ": array(",i,",",j,")=", myFarray(i,j)
    enddo
  enddo
enddo
```

24.2.7 Array bounds

The loop over Array elements at the end of the last section only works correctly because of the default definition of the *computational* and *total regions* used in the example. In general, without such specific knowledge about an Array object, it is necessary to use a more formal approach to access its regions with DE-local indices.
The DE-local *exclusive region* takes a central role in the definition of Array bounds. Even as the *computational region* may adjust during the course of execution the *exclusive region* remains unchanged. The *exclusive region* provides a unique reference frame for the index space of all Arrays associated with the same DistGrid.
There is a choice between two indexing options that needs to be made during Array creation. By default each DE-local exclusive region starts at `(1, 1, ..., 1)`. However, for some computational kernels it may be more convenient

to choose the index bounds of the DE-local exclusive regions to match the index space coordinates as they are defined in the corresponding DistGrid object. The second option is only available if the DistGrid object does not contain any non-contiguous decompositions (such as cyclically decomposed dimensions).
The following example code demonstrates the safe way of dereferencing the DE-local exclusive regions of the previously created `array` object.

```
  allocate(exclusiveUBound(2, localDeCount))  ! dimCount=2
  allocate(exclusiveLBound(2, localDeCount))  ! dimCount=2
  call ESMF_ArrayGet(array, indexflag=indexflag, &
    exclusiveLBound=exclusiveLBound, exclusiveUBound=exclusiveUBound, rc=rc)
  if (indexflag == ESMF_INDEX_DELOCAL) then
    ! this is the default
!    print *, "DE-local exclusive regions start at (1,1)"
    do de=1, localDeCount
      call ESMF_LocalArrayGet(larrayList(de), myFarray, &
          datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
      do i=1, exclusiveUBound(1, de)
        do j=1, exclusiveUBound(2, de)
!          print *, "DE-local exclusive region for PET-local DE=", de, &
!            ": array(",i,",",j,")=", myFarray(i,j)
        enddo
      enddo
    enddo
  else if (indexflag == ESMF_INDEX_GLOBAL) then
    ! only if set during ESMF_ArrayCreate()
!    print *, "DE-local exclusive regions of this Array have global bounds"
    do de=1, localDeCount
      call ESMF_LocalArrayGet(larrayList(de), myFarray, &
         datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
      do i=exclusiveLBound(1, de), exclusiveUBound(1, de)
        do j=exclusiveLBound(2, de), exclusiveUBound(2, de)
!          print *, "DE-local exclusive region for PET-local DE=", de, &
!            ": array(",i,",",j,")=", myFarray(i,j)
        enddo
      enddo
    enddo
  endif
  call ESMF_ArrayDestroy(array, rc=rc) ! destroy the array object
```

Obviously the second branch of this simple code will work for either case, however, if a complex computational kernel was written assuming `ESMF_INDEX_DELOCAL` type bounds the second branch would simply be used to indicate the problem and bail out.
The advantage of the `ESMF_INDEX_GLOBAL` index option is that the Array bounds directly contain information on where the DE-local Array piece is located in a global index space sense. When the `ESMF_INDEX_DELOCAL` option is used the correspondence between local and global index space must be made by querying the associated DistGrid for the DE-local `indexList` arguments.

24.2.8 Computational region and extra elements for halo or padding

In the previous examples the computational region of `array` was chosen by default to be identical to the exclusive region defined by the DistGrid argument during Array creation. In the following the same `arrayspec` and `distgrid` objects as before will be used to create an Array but now a larger computational region shall be defined around each DE-local exclusive region. Furthermore, extra space will be defined around the computational region of each DE to accommodate a halo and/or serve as memory padding.
In this example the `indexflag` argument is set to `ESMF_INDEX_GLOBAL` indicating that the bounds of the exclusive region correspond to the index space coordinates as they are defined by the DistGrid object.

The same `arrayspec` and `distgrid` objects as before are used which also allows the reuse of the already allocated `larrayList` variable.

```
  array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
    computationalLWidth=(/0,3/), computationalUWidth=(/1,1/), &
    totalLWidth=(/1,4/), totalUWidth=(/3,1/), &
    indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

Obtain the `larrayList` on every PET.

```
  allocate(localDeList(localDeCount))
  call ESMF_ArrayGet(array, localarrayList=larrayList, &
    localDeList=localDeList, rc=rc)
```

The bounds of DE 1 for `array` are shown in the following diagram to illustrate the situation. Notice that the `totalLWidth` and `totalUWidth` arguments in the ArrayCreate() call define the total region with respect to the exclusive region given for each DE by the `distgrid` argument.

```
+-(3,-3)--------------------------------+
|\                                      |
| +-(4,-2)-+-(4,1)--------------------+--+
| |        |                          |  |
| |        |                          |  |
| |        |          DE 1            |  |
| |        |                          |  |
| |        |                          |  |
| |        | Exclusive Region         |  |
| |        +--------------------(5,2)-+  |
| | Computational Region                 |
| +------------------------------(6,3)--+
|                                        |
| Total Region                           |
+--------------------------------(8,3)--+
```

When working with this `array` it is possible for the computational kernel to overstep the exclusive region for both read/write access (computational region) and potentially read-only access into the total region outside of the computational region, if a halo operation provides valid entries for these elements.
The Array object can be queried for absolute *bounds*

```
  allocate(computationalLBound(2, localDeCount))  ! dimCount=2
  allocate(computationalUBound(2, localDeCount))  ! dimCount=2
  allocate(totalLBound(2, localDeCount))          ! dimCount=2
  allocate(totalUBound(2, localDeCount))          ! dimCount=2
  call ESMF_ArrayGet(array, exclusiveLBound=exclusiveLBound, &
    exclusiveUBound=exclusiveUBound, &
    computationalLBound=computationalLBound, &
    computationalUBound=computationalUBound, &
    totalLBound=totalLBound, &
    totalUBound=totalUBound, rc=rc)
```

or for the relative *widths*.

```
  allocate(computationalLWidth(2, localDeCount))  ! dimCount=2
  allocate(computationalUWidth(2, localDeCount))  ! dimCount=2
  allocate(totalLWidth(2, localDeCount))          ! dimCount=2
  allocate(totalUWidth(2, localDeCount))          ! dimCount=2
  call ESMF_ArrayGet(array, computationalLWidth=computationalLWidth, &
    computationalUWidth=computationalUWidth, totalLWidth=totalLWidth, &
    totalUWidth=totalUWidth, rc=rc)
```

Either way the dereferencing of Array data is centered around the DE-local exclusive region:

```
  do de=1, localDeCount
    call ESMF_LocalArrayGet(larrayList(de), myFarray, &
       datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
    ! initialize the DE-local array
    myFarray = 0.1d0 * localDeList(de)
    ! first time through the total region of array
!    print *, "myFarray bounds for DE=", localDeList(de), &
!      lbound(myFarray),  ubound(myFarray)
    do j=exclusiveLBound(2, de), exclusiveUBound(2, de)
      do i=exclusiveLBound(1, de), exclusiveUBound(1, de)
!        print *, "Excl region DE=", localDeList(de), &
!        ": array(",i,",",j,")=",  myFarray(i,j)
      enddo
    enddo
    do j=computationalLBound(2, de), computationalUBound(2, de)
      do i=computationalLBound(1, de), computationalUBound(1, de)
!        print *, "Excl region DE=", localDeList(de), &
!        ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo
    do j=totalLBound(2, de), totalUBound(2, de)
      do i=totalLBound(1, de), totalUBound(1, de)
!        print *, "Total region DE=", localDeList(de), &
!        ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo

    ! second time through the total region of array
    do j=exclusiveLBound(2, de)-totalLWidth(2, de), &
      exclusiveUBound(2, de)+totalUWidth(2, de)
      do i=exclusiveLBound(1, de)-totalLWidth(1, de), &
        exclusiveUBound(1, de)+totalUWidth(1, de)
!        print *, "Excl region DE=", localDeList(de), &
!        ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo
  enddo
```

24.2.9 Create 1D and 3D Arrays

All previous examples were written for the 2D case. There is, however, no restriction within the Array or DistGrid class that limits the dimensionality of Array objects beyond the language specific limitations (7D for Fortran).
In order to create an `n`-dimensional Array the rank indicated by both the `arrayspec` and the `distgrid` arguments specified during Array create must be equal to `n`. A 1D Array of double precision real data hence requires the following `arrayspec`.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=1, rc=rc)
```

The index space covered by the Array and the decomposition description is provided to the Array create method by the `distgrid` argument. The index space in this example has 16 elements and covers the interval $[-10, 5]$. It is decomposed into as many DEs as there are PETs in the current context.

```
distgrid1D = ESMF_DistGridCreate(minIndex=(/-10/), maxIndex=(/5/), &
  regDecomp=(/petCount/), rc=rc)
```

A 1D Array object with default regions can now be created.

```
array1D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid1D, rc=rc)
```

The creation of a 3D Array proceeds analogous to the 1D case. The rank of the `arrayspec` must be changed to 3

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
```

and an appropriate 3D DistGrid object must be created

```
distgrid3D = ESMF_DistGridCreate(minIndex=(/1,1,1/), &
  maxIndex=(/16,16,16/), regDecomp=(/4,4,4/), rc=rc)
```

before an Array object can be created.

```
array3D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid3D, rc=rc)
```

The `distgrid3D` object decomposes the 3-dimensional index space into $4 \times 4 \times 4 = 64$ DEs. These DEs are laid out across the computational resources (PETs) of the current component according to a default DELayout that is created during the DistGrid create call. Notice that in the index space proposal a DELayout does not have a sense of dimensionality. The DELayout function is simply to map DEs to PETs. The DistGrid maps chunks of index space against DEs and thus its rank is equal to the number of index space dimensions.
The previously defined DistGrid and the derived Array object decompose the index space along all three dimension. It is, however, not a requirement that the decomposition be along all dimensions. An Array with the same 3D index space could as well be decomposed along just one or along two of the dimensions. The following example shows how for the same index space only the last two dimensions are decomposed while the first Array dimension has full extent on all DEs.

```
call ESMF_ArrayDestroy(array3D, rc=rc)
call ESMF_DistGridDestroy(distgrid3D, rc=rc)
distgrid3D = ESMF_DistGridCreate(minIndex=(/1,1,1/), &
  maxIndex=(/16,16,16/), regDecomp=(/1,4,4/), rc=rc)
array3D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid3D, rc=rc)
```

24.2.10 Working with Arrays of different rank

Assume a computational kernel that involves the `array3D` object as it was created at the end of the previous section. Assume further that the kernel also involves a 2D Array on a 16x16 index space where each point (j,k) was interacting with each (i,j,k) column of the 3D Array. An efficient formulation would require that the decomposition of the 2D Array must match that of the 3D Array and further the DELayout be identical. The following code shows how this can be accomplished.

```
  call ESMF_DistGridGet(distgrid3D, delayout=delayout, rc=rc) ! get DELayout
  distgrid2D = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/16,16/), &
    regDecomp=(/4,4/), delayout=delayout, rc=rc)
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
  array2D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid2D, rc=rc)
```

Now the following kernel is sure to work with `array3D` and `array2D`.

```
  call ESMF_DELayoutGet(delayout, localDeCount=localDeCount, rc=rc)
  allocate(larrayList1(localDeCount))
  call ESMF_ArrayGet(array3D, localarrayList=larrayList1, rc=rc)
  allocate(larrayList2(localDeCount))
  call ESMF_ArrayGet(array2D, localarrayList=larrayList2, rc=rc)
  do de=1, localDeCount
    call ESMF_LocalArrayGet(larrayList1(de), myFarray3D, &
      datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
    myFarray3D = 0.1d0 * de ! initialize
    call ESMF_LocalArrayGet(larrayList2(de), myFarray2D, &
      datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
    myFarray2D = 0.5d0 * de ! initialize
    do k=1, 4
      do j=1, 4
        dummySum = 0.d0
        do i=1, 16
          dummySum = dummySum + myFarray3D(i,j,k) ! sum up the (j,k) column
        enddo
        dummySum = dummySum * myFarray2D(j,k) ! multiply with local 2D element
!        print *, "dummySum(",j,k,")=",dummySum
      enddo
    enddo
  enddo
```

24.2.11 Array and DistGrid rank – 2D+1 Arrays

Except for the special Array create interface that implements a copy from an existing Array object all other Array create interfaces require the specification of at least two arguments: `farray` and `distgrid`, `larrayList` and `distgrid`, or `arrayspec` and `distgrid`. In all these cases both required arguments contain a sense of dimensionality. The relationship between these two arguments deserves extra attention.

The first argument, `farray`, `larrayList` or `arrayspec`, determines the rank of the created Array object, i.e. the dimensionality of the actual data storage. The rank of a native language array, extracted from an Array object, is equal to the rank specified by either of these arguments. So is the `rank` that is returned by the `ESMF_ArrayGet()` call.

The rank specification contained in the `distgrid` argument, which is of type `ESMF_DistGrid`, on the other hand has no affect on the rank of the Array. The `dimCount` specified by the DistGrid object, which may be equal, greater or less than the Array rank, determines the dimensionality of the *decomposition*.

While there is no constraint between DistGrid `dimCount` and Array `rank`, there is an important relationship between the two, resulting in the concept of index space dimensionality. Array dimensions can be arbitrarily mapped against DistGrid dimension, rendering them *decomposed* dimensions. The index space dimensionality is equal to the number of decomposed Array dimensions.

Array dimensions that are not mapped to DistGrid dimensions are the *undistributed* dimensions of the Array. They are not part of the index space. The mapping is specified during `ESMF_ArrayCreate()` via the `distgridToArrayMap` argument. DistGrid dimensions that have not been associated with Array dimensions are *replicating* dimensions. The Array will be replicated across the DEs that lie along replication DistGrid dimensions.

Undistributed Array dimensions can be used to store multi-dimensional data for each Array index space element. One application of this is to store the components of a vector quantity in a single Array. The same 2D `distgrid` object as before will be used.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

The rank in the `arrayspec` argument, however, must change from 2 to 3 in order to provide for the extra Array dimension.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
```

During Array creation with extra dimension(s) it is necessary to specify the bounds of these undistributed dimension(s). This requires two additional arguments, `undistLBound` and `undistUBound`, which are vectors in order to accommodate multiple undistributed dimensions. The other arguments remain unchanged and apply across all undistributed components.

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  totalLWidth=(/0,1/), totalUWidth=(/0,1/), &
  undistLBound=(/1/), undistUBound=(/2/), rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

This will create `array` with 2+1 dimensions. The 2D DistGrid is used to describe decomposition into DEs with 2 Array dimensions mapped to the DistGrid dimensions resulting in a 2D index space. The extra Array dimension provides storage for multi component user data within the Array object.
By default the `distgrid` dimensions are associated with the first Array dimensions in sequence. For the example above this means that the first 2 Array dimensions are decomposed according to the provided 2D DistGrid. The 3rd Array dimension does not have an associated DistGrid dimension, rendering it an undistributed Array dimension.
Native language access to an Array with undistributed dimensions is in principle the same as without extra dimensions.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)
allocate(larrayList(localDeCount))
call ESMF_ArrayGet(array, localarrayList=larrayList, rc=rc)
```

The following loop shows how a Fortran pointer to the DE-local data chunks can be obtained and used to set data values in the exclusive regions. The `myFarray3D` variable must be of rank 3 to match the Array rank of `array`. However, variables such as `exclusiveUBound` that store the information about the decomposition, remain to be allocated for the 2D index space.

```
call ESMF_ArrayGet(array, exclusiveLBound=exclusiveLBound, &
  exclusiveUBound=exclusiveUBound, rc=rc)
do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myFarray3D, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
  myFarray3D = 0.0 ! initialize
  myFarray3D(exclusiveLBound(1,de):exclusiveUBound(1,de), &
    exclusiveLBound(2,de):exclusiveUBound(2,de), 1) = 5.1 ! dummy assignment
  myFarray3D(exclusiveLBound(1,de):exclusiveUBound(1,de), &
    exclusiveLBound(2,de):exclusiveUBound(2,de), 2) = 2.5 ! dummy assignment
enddo
deallocate(larrayList)
```

For some applications the default association rules between DistGrid and Array dimensions may not satisfy the user's needs. The optional `distgridToArrayMap` argument can be used during Array creation to explicitly specify the mapping between DistGrid and Array dimensions. To demonstrate this the following lines of code reproduce the above example but with rearranged dimensions. Here the `distgridToArrayMap` argument is a list with two elements corresponding to the DistGrid `dimCount` of 2. The first element indicates which Array dimension the first DistGrid dimension is mapped against. Here the 1st DistGrid dimension maps against the 3rd Array dimension and the 2nd DistGrid dimension maps against the 1st Array dimension. This leaves the 2nd Array dimension to be the extra and undistributed dimension in the resulting Array object.

```
call ESMF_ArrayDestroy(array, rc=rc)
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  distgridToArrayMap=(/3, 1/), totalLWidth=(/0,1/), totalUWidth=(/0,1/), &
  undistLBound=(/1/), undistUBound=(/2/), rc=rc)
```

Operations on the Array object as a whole are unchanged by the different mapping of dimensions.
When working with Arrays that contain explicitly mapped Array and DistGrid dimensions it is critical to know the order in which the entries of *width* and *bound* arguments that are associated with distributed Array dimensions are specified. The size of these arguments is equal to the DistGrid `dimCount`, because the maximum number of distributed Array dimensions is given by the dimensionality of the index space.
The order of dimensions in these arguments, however, is *not* that of the associated DistGrid. Instead each entry corresponds to the distributed Array dimensions in sequence. In the example above the entries in `totalLWidth` and `totalUWidth` correspond to Array dimensions 1 and 3 in this sequence.
The `distgridToArrrayMap` argument optionally provided during Array create indicates how the DistGrid dimensions map to Array dimensions. The inverse mapping, i.e. Array to DistGrid dimensions, is just as important. The `ESMF_ArrayGet()` call offers both mappings as `distgridToArrrayMap` and `arrayToDistGridMap`, respectively. The number of elements in `arrayToDistGridMap` is equal to the rank of the Array. Each element corresponds to an Array dimension and indicates the associated DistGrid dimension by an integer number. An entry of "0" in `arrayToDistGridMap` indicates that the corresponding Array dimension is undistributed.
Correct understanding about the association between Array and DistGrid dimensions becomes critical for correct data access into the Array.

```
allocate(arrayToDistGridMap(3))  ! arrayRank = 3
call ESMF_ArrayGet(array, arrayToDistGridMap=arrayToDistGridMap, &
  exclusiveLBound=exclusiveLBound, exclusiveUBound=exclusiveUBound, &
  localDeCount=localDeCount, rc=rc)
if (arrayToDistGridMap(2) /= 0) then   ! check if extra dimension at
  ! expected index indicate problem and bail out
endif
! obtain larrayList for local DEs
allocate(larrayList(localDeCount))
call ESMF_ArrayGet(array, localarrayList=larrayList, rc=rc)
do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myFarray3D, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
  myFarray3D(exclusiveLBound(1,de):exclusiveUBound(1,de), &
    1, exclusiveLBound(2,de):exclusiveUBound(2,de)) = 10.5 !dummy assignment
  myFarray3D(exclusiveLBound(1,de):exclusiveUBound(1,de), &
    2, exclusiveLBound(2,de):exclusiveUBound(2,de)) = 23.3 !dummy assignment
enddo
deallocate(exclusiveLBound, exclusiveUBound)
deallocate(arrayToDistGridMap)
deallocate(larrayList)
call ESMF_ArrayDestroy(array, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

24.2.12 Arrays with replicated dimensions

Thus far most examples demonstrated cases where the DistGrid `dimCount` was equal to the Array `rank`. The previous section introduced the concept of Array *tensor* dimensions when `dimCount < rank`. In this section `dimCount` and `rank` are assumed completely unconstrained and the relationship to `distgridToArrayMap` and `arrayToDistGridMap` will be discussed.

The Array class allows completely arbitrary mapping between Array and DistGrid dimensions. Most cases considered in the previous sections used the default mapping which assigns the DistGrid dimensions in sequence to the lower Array dimensions. Extra Array dimensions, if present, are considered non-distributed tensor dimensions for which the optional `undistLBound` and `undistUBound` arguments must be specified.

The optional `distgridToArrayMap` argument provides the option to override the default DistGrid to Array dimension mapping. The entries of the `distgridToArrayMap` array correspond to the DistGrid dimensions in sequence and assign a unique Array dimension to each DistGrid dimension. DistGrid and Array dimensions are indexed starting at `1` for the lowest dimension. A value of `"0"` in the `distgridToArrayMap` array indicates that the respective DistGrid dimension is *not* mapped against any Array dimension. What this means is that the Array will be replicated along this DistGrid dimension.

As a first example consider the case where a 1D Array

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=1, rc=rc)
```

is created on the 2D DistGrid used during the previous section.

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

Here the default DistGrid to Array dimension mapping is used which assigns the Array dimensions in sequence to the DistGrid dimensions starting with dimension "1". Extra DistGrid dimensions are considered replicator dimensions because the Array will be replicated along those dimensions. In the above example the 2nd DistGrid dimension will cause 1D Array pieces to be replicated along the DEs of the 2nd DistGrid dimension. Replication in the context of `ESMF_ArrayCreate()` does not mean that data values are communicated and replicated between different DEs, but it means that different DEs provide memory allocations for *identical* exclusive elements.

Access to the data storage of an Array that has been replicated along DistGrid dimensions is the same as for Arrays without replication.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)


allocate(larrayList(localDeCount))
allocate(localDeList(localDeCount))
call ESMF_ArrayGet(array, localarrayList=larrayList, &
  localDeList=localDeList, rc=rc)
```

The `array` object was created without additional padding which means that the bounds of the Fortran array pointer correspond to the bounds of the exclusive region. The following loop will cycle through all local DEs, print the DE number as well as the Fortran array pointer bounds. The bounds should be:

```
            lbound        ubound

 DE 0:        1             3         --+
 DE 2:        1             3         --|  1st replication set
 DE 4:        1             3         --+

 DE 1:        1             2         --+
 DE 3:        1             2         --|  2nd replication set
 DE 5:        1             2         --+
```

```
do de=1, localDeCount
  call ESMF_LocalArrayGet(larrayList(de), myFarray1D, &
    datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
```

```
    print *, "DE ",localDeList(de)," [", lbound(myFarray1D), &
      ubound(myFarray1D),"]"
  enddo
  deallocate(larrayList)
  deallocate(localDeList)
  call ESMF_ArrayDestroy(array, rc=rc)
```

The Fortran array pointer in the above loop was of rank 1 because the Array object was of rank 1. However, the `distgrid` object associated with `array` is 2-dimensional! Consequently DistGrid based information queried from `array` will be 2D. The `distgridToArrayMap` and `arrayToDistGridMap` arrays provide the necessary mapping to correctly associate DistGrid based information with Array dimensions.
The next example creates a 2D Array

```
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

on the previously used 2D DistGrid. By default, i.e. without the `distgridToArrayMap` argument, both DistGrid dimensions would be associated with the two Array dimensions. However, the `distgridToArrayMap` specified in the following call will only associate the second DistGrid dimension with the first Array dimension. This will render the first DistGrid dimension a replicator dimension and the second Array dimension a tensor dimension for which 1D `undistLBound` and `undistUBound` arguments must be supplied.

```
  array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
    distgridToArrayMap=(/0,1/), undistLBound=(/11/), &
    undistUBound=(/14/), rc=rc)
```

```
  call ESMF_ArrayDestroy(array, rc=rc)
```

Finally, the same `arrayspec` and `distgrid` arguments are used to create a 2D Array that is fully replicated in both dimensions of the DistGrid. Both Array dimensions are now tensor dimensions and both DistGrid dimensions are replicator dimensions.

```
  array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
    distgridToArrayMap=(/0,0/), undistLBound=(/11,21/), &
    undistUBound=(/14,22/), rc=rc)
```

The result will be an Array with local lower bound (/11,21/) and upper bound (/14,22/) on all 6 DEs of the DistGrid.

```
  call ESMF_ArrayDestroy(array, rc=rc)
```

```
  call ESMF_DistGridDestroy(distgrid, rc=rc)
```

Replicated Arrays can also be created from existing local Fortran arrays. The following Fortran array allocation will provide a 3 x 10 array on each PET.

```
  allocate(myFarray2D(3,10))
```

Assuming a petCount of 4 the following DistGrid defines a 2D index space that is distributed across the PETs along the first dimension.

```
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

The following call creates an Array object on the above distgrid using the locally existing `myFarray2D` Fortran arrays. The difference compared to the case with automatic memory allocation is that instead of `arrayspec` the Fortran array is provided as argument. Furthermore, the `undistLBound` and `undistUBound` arguments can be omitted, defaulting into Array tensor dimension lower bound of 1 and an upper bound equal to the size of the respective Fortran array dimension.

```
array = ESMF_ArrayCreate(farray=myFarray2D, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, distgridToArrayMap=(/0,2/), rc=rc)
```

The `array` object associates the 2nd DistGrid dimension with the 2nd Array dimension. The first DistGrid dimension is not associated with any Array dimension and will lead to replication of the Array along the DEs of this direction.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

24.2.13 Communication – Scatter and Gather

It is a common situation, particularly in legacy code, that an ESMF Array object must be filled with data originating from a large Fortran array stored on a single PET.

```
if (localPet == 0) then
  allocate(farray(10,20,30))
  do k=1, 30
    do j=1, 20
      do i=1, 10
        farray(i, j, k) = k*1000 + j*100 +  i
      enddo
    enddo
  enddo
endif
```

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1,1/), maxIndex=(/10,20,30/), &
  rc=rc)
```

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=3, rc=rc)
```

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

The `ESMF_ArrayScatter()` method provides a convenient way of scattering array data from a single root PET across the DEs of an ESMF Array object.

```
call ESMF_ArrayScatter(array, farray=farray, rootPet=0, rc=rc)
```

```
if (localPet == 0) then
  deallocate(farray)
endif
```

The destination of the ArrayScatter() operation are all the DEs of a single tile. For multi-tile Arrays the destination tile can be specified. The shape of the scattered Fortran array must match the shape of the destination tile in the ESMF Array.

Gathering data decomposed and distributed across the DEs of an ESMF Array object into a single Fortran array on root PET is accomplished by calling `ESMF_ArrayGather()`.

```
if (localPet == 3) then
  allocate(farray(10,20,30))
endif

call ESMF_ArrayGather(array, farray=farray, rootPet=3, rc=rc)


if (localPet == 3) then
  deallocate(farray)
endif
```

The source of the ArrayGather() operation are all the DEs of a single tile. For multi-tile Arrays the source tile can be specified. The shape of the gathered Fortran array must match the shape of the source tile in the ESMF Array.

The `ESMF_ArrayScatter()` operation allows to fill entire replicated Array objects with data coming from a single root PET.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)


call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)


array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  distgridToArrayMap=(/0,0/), undistLBound=(/11,21/), &
  undistUBound=(/14,22/), rc=rc)
```

The shape of the Fortran source array used in the Scatter() call must be that of the contracted Array, i.e. contracted DistGrid dimensions do not count. For the `array` just created this means that the source array on `rootPet` must be of shape 4 x 2.

```
if (localPet == 0) then
  allocate(myFarray2D(4,2))
  do j=1,2
    do i=1,4
      myFarray2D(i,j) = i * 100.d0 + j * 1.2345d0 ! initialize
    enddo
  enddo
endif

call ESMF_ArrayScatter(array, farray=myFarray2D, rootPet=0, rc=rc)


if (localPet == 0) then
  deallocate(myFarray2D)
endif
```

This will have filled each local 4 x 2 Array piece with the replicated data of `myFarray2D`.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

As a second example for the use of Scatter() and Gather() consider the following replicated Array created from existing local Fortran arrays.

```
allocate(myFarray2D(3,10))
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

```
array = ESMF_ArrayCreate(farray=myFarray2D, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, distgridToArrayMap=(/0,2/), rc=rc)
```

The `array` object associates the 2nd DistGrid dimension with the 2nd Array dimension. The first DistGrid dimension is not associated with any Array dimension and will lead to replication of the Array along the DEs of this direction. Still, the local arrays that comprise the `array` object refer to independent pieces of memory and can be initialized independently.

```
myFarray2D = localPet ! initialize
```

However, the notion of replication becomes visible when an array of shape 3 x 10 on root PET 0 is scattered across the Array object.

```
if (localPet == 0) then
  allocate(myFarray2D2(5:7,11:20))

  do j=11,20
    do i=5,7
      myFarray2D2(i,j) = i * 100.d0 + j * 1.2345d0 ! initialize
    enddo
  enddo
endif

call ESMF_ArrayScatter(array, farray=myFarray2D2, rootPet=0, rc=rc)
```

```
if (localPet == 0) then
  deallocate(myFarray2D2)
endif
```

The Array pieces on every DE will receive the same source data, resulting in a replication of data along DistGrid dimension 1.

When the inverse operation, i.e. `ESMF_ArrayGather()`, is applied to a replicated Array an intrinsic ambiguity needs to be considered. ESMF defines the gathering of data of a replicated Array as the collection of data originating from the numerically higher DEs. This means that data in replicated elements associated with numerically lower DEs will be ignored during `ESMF_ArrayGather()`. For the current example this means that changing the Array contents on PET 1, which here corresponds to DE 1,

```
if (localPet == 1) then
  myFarray2D = real(1.2345, ESMF_KIND_R8)
endif
```

will *not* affect the result of

```
allocate(myFarray2D2(3,10))
myFarray2D2 = 0.d0    ! initialize to a known value
call ESMF_ArrayGather(array, farray=myFarray2D2, rootPet=0, rc=rc)
```

The result remains completely defined by the unmodified values of Array in DE 3, the numerically highest DE. However, overriding the DE-local Array piece on DE 3

```
if (localPet==3) then
  myFarray2D = real(5.4321, ESMF_KIND_R8)
endif
```

will change the outcome of

```
call ESMF_ArrayGather(array, farray=myFarray2D2, rootPet=0, rc=rc)
```

as expected.

```
deallocate(myFarray2D2)

call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

24.2.14 Communication – Halo

One of the most fundamental communication pattern in domain decomposition codes is the *halo* operation. The ESMF Array class supports halos by allowing memory for extra elements to be allocated on each DE. See sections 24.2.2 and 24.2.8 for examples and details on how to create an Array with extra DE-local elements.
Here we consider an Array object that is created on a DistGrid that defines a 10 x 20 index space, decomposed into 4 DEs using a regular 2 x 2 decomposition.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/2,2/), rc=rc)
```

The Array holds 2D double precision float data.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

The `totalLWidth` and `totalUWidth` arguments are used during Array creation to allocate 2 extra elements along every direction outside the exclusive region defined by the DistGrid for every DE. (The `indexflag` set to `ESMF_INDEX_GLOBAL` in this example does not affect the halo behavior of Array. The setting is simply more convenient for the following code.)

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  totalLWidth=(/2,2/), totalUWidth=(/2,2/), indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
```

Without the explicit definition of boundary conditions in the DistGrid the following inner connections are defined.

```
 +-------------------+        +-------------------+
 | \       2       / |        | \       2       / |
 |  +-------------+  |        |  +-------------+  |
 |  |     DE 0    |  |        |  |     DE 2    |  |
 |  |             |  |        |  |             |  |
 |2 |    5 x 10   | 2|   <->  |2 |    5 x 10   | 2|
 |  |             |  |        |  |             |  |
 |  |             |  |        |  |             |  |
 |  +-------------+  |        |  +-------------+  |
 | /       2       \ |        | /       2       \ |
 +-------------------+        +-------------------+

           ^            \/              ^
           |            /\              |
           v                            v

 +-------------------+        +-------------------+
 | \       2       / |        | \       2       / |
 |  +-------------+  |        |  +-------------+  |
 |  |     DE 1    |  |        |  |     DE 3    |  |
 |  |             |  |        |  |             |  |
 |2 |    5 x 10   | 2|   <->  |2 |    5 x 10   | 2|
 |  |             |  |        |  |             |  |
 |  |             |  |        |  |             |  |
 |  +-------------+  |        |  +-------------+  |
 | /       2       \ |        | /       2       \ |
 +-------------------+        +-------------------+
```

The exclusive region on each DE is of shape 5 x 10, while the total region on each DE is of shape (5+2+2) x (10+2+2) = 9 x 14. In a typical application the elements in the exclusive region are updated exclusively by the PET that owns the DE. In this example the exclusive elements on every DE are initialized to the value $f(i,j)$ of the geometric function

$$f(i,j) = \sin(\alpha i)\cos(\beta j), \tag{1}$$

where

$$\alpha = 2\pi/N_i, i = 1, ...N_i \tag{2}$$

and

$$\beta = 2\pi/N_j, j = 1, ...N_j, \tag{3}$$

with $N_i = 10$ and $N_j = 20$.

```
  a = 2. * 3.14159 / 10.
  b = 2. * 3.14159 / 20.

  call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)



  call ESMF_ArrayGet(array, exclusiveLBound=eLB, exclusiveUBound=eUB, rc=rc)
```

```
do j=eLB(2,1), eUB(2,1)
  do i=eLB(1,1), eUB(1,1)
    farrayPtr(i,j) = sin(a*i) * cos(b*j)  ! test function
  enddo
enddo
```

The above loop only initializes the exclusive elements on each DE. The extra elements, outside the exclusive region, are left untouched, holding undefined values. Elements outside the exclusive region that correspond to exclusive elements in neighboring DEs can be filled with the data values in those neighboring elements. This is the definition of the halo operation.

In ESMF the halo communication pattern is first precomputed and stored in a RouteHandle object. This RouteHandle can then be used repeatedly to perform the same halo operation in the most efficient way.

The default halo operation for an Array is precomputed by the following call.

```
call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle, rc=rc)
```

The `haloHandle` now holds the default halo operation for `array`, which matches as many elements as possible outside the exclusive region to their corresponding halo source elements in neighboring DEs. Elements that could not be matched, e.g. at the edge of the global domain with open boundary conditions, will not be updated by the halo operation.

The `haloHandle` is applied through the `ESMF_ArrayHalo()` method.

```
call ESMF_ArrayHalo(array=array, routehandle=haloHandle, rc=rc)
```

Finally the resources held by `haloHandle` need to be released.

```
call ESMF_ArrayHaloRelease(routehandle=haloHandle, rc=rc)
```

The `array` object created above defines a 2 element wide rim around the exclusive region on each DE. Consequently the default halo operation used above will have resulted in updating both elements along the inside edges. For simple numerical kernels often a single halo element is sufficient. One way to achieve this would be to reduce the size of the rim surrounding the exclusive region to 1 element along each direction. However, if the same Array object is also used for higher order kernels during a different phase of the calculation, a larger element rim is required. For this case `ESMF_ArrayHaloStore()` offers two optional arguments `haloLDepth` and `haloUDepth`. Using these arguments a reduced halo depth can be specified.

```
call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle, &
  haloLDepth=(/1,1/), haloUDepth=(/1,1/), rc=rc)
```

This halo operation with a depth of 1 is sufficient to support a simple quadratic differentiation kernel.

```
allocate(farrayTemp(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)))

do step=1, 4
  call ESMF_ArrayHalo(array=array, routehandle=haloHandle, rc=rc)


  do j=eLB(2,1), eUB(2,1)
    do i=eLB(1,1), eUB(1,1)
      if (i==1) then
        ! global edge
```

```
        farrayTemp(i,j) = 0.5 * (-farrayPtr(i+2,j) + 4.*farrayPtr(i+1,j) &
          - 3.*farrayPtr(i,j)) / a
      else if (i==10) then
        ! global edge
        farrayTemp(i,j) = 0.5 * (farrayPtr(i-2,j) - 4.*farrayPtr(i-1,j) &
          + 3.*farrayPtr(i,j)) / a
      else
        farrayTemp(i,j) = 0.5 * (farrayPtr(i+1,j) - farrayPtr(i-1,j)) / a
      endif
    enddo
  enddo
  farrayPtr(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)) = farrayTemp
enddo

deallocate(farrayTemp)

call ESMF_ArrayHaloRelease(routehandle=haloHandle, rc=rc)
```

The special treatment of the global edges in the above kernel is due to the fact that the underlying DistGrid object does not define any special boundary conditions. By default open global boundaries are assumed which means that the rim elements on the global edges are untouched during the halo operation, and cannot be used in the symmetric numerical derivative formula. The kernel can be simplified (and the calculation is more precise) with periodic boundary conditions along the first Array dimension.
First destroy the current Array and DistGrid objects.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

Create a DistGrid with periodic boundary condition along the first dimension.

```
allocate(connectionList(1))  ! one connection
call ESMF_DistGridConnectionSet(connection=connectionList(1), &
   tileIndexA=1, tileIndexB=1, positionVector=(/10, 0/), rc=rc)
```

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/2,2/), connectionList=connectionList, rc=rc)
```

```
deallocate(connectionList)
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  totalLWidth=(/2,2/), totalUWidth=(/2,2/), indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
```

Initialize the exclusive elements to the same geometric function as before.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)
```

```
call ESMF_ArrayGet(array, exclusiveLBound=eLB, exclusiveUBound=eUB, rc=rc)
```

```
do j=eLB(2,1), eUB(2,1)
  do i=eLB(1,1), eUB(1,1)
    farrayPtr(i,j) = sin(a*i) * cos(b*j)  ! test function
  enddo
enddo
```

The numerical kernel only operates along the first dimension. An asymmetric halo depth can be used to take this fact into account.

```
call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle, &
  haloLDepth=(/1,0/), haloUDepth=(/1,0/), rc=rc)
```

Now the same numerical kernel can be used without special treatment of global edge elements. The symmetric derivative formula can be used for all exclusive elements.

```
allocate(farrayTemp(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)))

do step=1, 4
  call ESMF_ArrayHalo(array=array, routehandle=haloHandle, rc=rc)


  do j=eLB(2,1), eUB(2,1)
    do i=eLB(1,1), eUB(1,1)
      farrayTemp(i,j) = 0.5 * (farrayPtr(i+1,j) - farrayPtr(i-1,j)) / a
    enddo
  enddo
  farrayPtr(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)) = farrayTemp
enddo
```

The precision of the above kernel can be improved by going to a higher order interpolation. Doing so requires that the halo depth must be increased. The following code resets the exclusive Array elements to the test function, precomputes a RouteHandle for a halo operation with depth 2 along the first dimension, and finally uses the deeper halo in the higher order kernel.

```
do j=eLB(2,1), eUB(2,1)
  do i=eLB(1,1), eUB(1,1)
    farrayPtr(i,j) = sin(a*i) * cos(b*j)  ! test function
  enddo
enddo

call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle2, &
  haloLDepth=(/2,0/), haloUDepth=(/2,0/), rc=rc)



do step=1, 4
  call ESMF_ArrayHalo(array=array, routehandle=haloHandle2, rc=rc)


  do j=eLB(2,1), eUB(2,1)
    do i=eLB(1,1), eUB(1,1)
```

```
      farrayTemp(i,j) = (-farrayPtr(i+2,j) + 8.*farrayPtr(i+1,j) &
        - 8.*farrayPtr(i-1,j) + farrayPtr(i-2,j)) / (12.*a)
    enddo
  enddo
  farrayPtr(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)) = farrayTemp
enddo

deallocate(farrayTemp)
```

ESMF supports having multiple halo operations defined on the same Array object at the same time. Each operation can be accessed through its unique RouteHandle. The above kernel could have made `ESMF_ArrayHalo()` calls with a depth of 1 along the first dimension using the previously precomputed `haloHandle` if it needed to. Both RouteHandles need to release their resources when no longer used.

```
call ESMF_ArrayHaloRelease(routehandle=haloHandle, rc=rc)
```

```
call ESMF_ArrayHaloRelease(routehandle=haloHandle2, rc=rc)
```

Finally the Array and DistGrid objects can be destroyed.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

24.2.15 Communication – Halo for arbitrary distribution

In the previous section the Array *halo* operation was demonstrated for regularly decomposed ESMF Arrays. However, the ESMF halo operation is not restricted to regular decompositions. The same Array halo methods apply unchanged to Arrays that are created on arbitrarily distributed DistGrids. This includes the non-blocking features discussed in section 24.2.19.

All of the examples in this section are based on the same arbitrarily distributed DistGrid. Section 31.3.6 discusses DistGrids with user-supplied, arbitrary sequence indices in detail. Here a global index space range from 1 through 20 is decomposed across 4 DEs. There are 4 PETs in this example with 1 DE per PET. Each PET constructs its local `seqIndexList` variable.

```
do i=1, 5
  seqIndexList(i) = localPet + (i - 1) * petCount + 1
enddo
```

This results in the following cylic distribution scheme:

```
 DE 0 on PET 0: seqIndexList = (/1, 5, 9, 13, 17/)
 DE 1 on PET 1: seqIndexList = (/2, 6, 10, 14, 18/)
 DE 2 on PET 2: seqIndexList = (/3, 7, 11, 15, 19/)
 DE 3 on PET 3: seqIndexList = (/4, 8, 12, 16, 20/)
```

The local `arbIndexList` variables are then used to create a DistGrid with the indicated arbitrary distribution pattern.

```
distgrid = ESMF_DistGridCreate(arbSeqIndexList=seqIndexList, rc=rc)
```

The resulting DistGrid is one-dimensional, although the user code may interpret the sequence indices as a 1D map into a problem of higher dimensionality.

In this example the local DE on each PET is associated with a 5 element exclusive region. Providing `arbIndexList` of different size on the different PETs is supported and would result in different number of exclusive elements on each PET.

Creating an ESMF Array on top of a DistGrid with arbitrary sequence indices is in principle no different from creating an Array on a regular DistGrid. However, while an Array that was created on a regular DistGrid automatically inherits the index space topology information that is contained within the DistGrid object, there is no such topology information available for DistGrid objects with arbitrary sequence indices. As a consequence of this, Arrays created on arbitrary DistGrids do not automatically have the information that is required to associated halo elements with the exclusive elements across DEs. Instead the user must supply this information explicitly during Array creation.

Multiple ArrayCreate() interfaces exist that allow the creation of an Array on a DistGrid with arbitrary sequence indices, while supplying the sequence indices for the halo region of the local DE through an additional argument with dummy name `haloSeqIndexList`. As in the regular case the ArrayCreate() interfaces differ in the way that the memory allocations for the Array elements are passed into the call. The following code shows how an ESMF Array can be wrapped around existing PET-local memory allocations. The allocations are of different size on each PET as to accommodate the correct number of local Array elements.

```
allocate(farrayPtr1d(5+localPet+1)) !use explicit Fortran allocate statement

if (localPet==0) then
  array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
    haloSeqIndexList=(/1/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==1) then
  array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
    haloSeqIndexList=(/1,2/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==2) then
  array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
    haloSeqIndexList=(/1,2,3/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==3) then
  array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
    haloSeqIndexList=(/1,2,3,4/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

The `haloSeqIndexList` arguments are 1D arrays of sequence indices. It is through this argument that the user associates the halo elements with exclusive Array elements covered by the DistGrid. In this example there are different number of halo elements on each DE. They are associated with exclusive elements as follows:

```
halo on DE 0 on PET 0: <seqIndex=1> first exclusive element on DE 0
halo on DE 1 on PET 1: <seqIndex=1> first exclusive element on DE 0
                       <seqIndex=2> first exclusive element on DE 1
halo on DE 2 on PET 2: <seqIndex=1> first exclusive element on DE 0
                       <seqIndex=2> first exclusive element on DE 1
                       <seqIndex=3> first exclusive element on DE 2
halo on DE 3 on PET 3: <seqIndex=1> first exclusive element on DE 0
```

```
                    <seqIndex=2> first exclusive element on DE 1
                    <seqIndex=3> first exclusive element on DE 2
                    <seqIndex=4> first exclusive element on DE 3
```

The ArrayCreate() call checks that the provided Fortran memory allocation is correctly sized to hold the exclusive elements, as indicated by the DistGrid object, plus the halo elements as indicated by the local `haloSeqIndexList` argument. The size of the Fortran allocation must match exactly or a runtime error will be returned.
Analogous to the case of Arrays on regular DistGrids, it is the exclusive region of the local DE that is typically modified by the code running on each PET. All of the ArrayCreate() calls that accept the `haloSeqIndexList` argument place the exclusive region at the beginning of the memory allocation on each DE and use the remaining space for the halo elements. The following loop demonstrates this by filling the exclusive elements on each DE with initial values. Remember that in this example each DE holds 5 exclusive elements associated with different arbitrary sequence indices.

```
do i=1, 5
  farrayPtr1d(i) = seqIndexList(i) / 10.
enddo
```

Now the exclusive elements of `array` are initialized on each DE, however, the halo elements remain unchanged. A RouteHandle can be set up that encodes the required communication pattern for a halo exchange. The halo exchange is precomputed according to the arbitrary sequence indices specified for the exclusive elements by the DistGrid and the sequence indices provided by the user for each halo element on the local DE in form of the `haloSeqIndexList` argument during ArrayCreate().

```
call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)
```

Executing this halo operation will update the local halo elements according to the associated sequence indices.

```
call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)
```

As always it is good practice to release the RouteHandle when done with it.

```
call ESMF_ArrayHaloRelease(haloHandle, rc=rc)
```

Also the Array object should be destroyed when no longer needed.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

Further, since the memory allocation was done explicitly using the Fortran `allocate()` statement, it is necessary to explicitly deallocate in order to prevent memory leaks in the user application.

```
deallocate(farrayPtr1d)
```

Alternatively the exact same Array can be created where ESMF does the memory allocation and deallocation. In this case the `typekind` of the Array must be specified explicitly.

```
if (localPet==0) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=(/1/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

```
  if (localPet==1) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1,2/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==2) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1,2,3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==3) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1,2,3,4/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
```

Use ArrayGet() to gain access to the local memory allocation.

```
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr1d, rc=rc)
```

The returned Fortran pointer can now be used to initialize the exclusive elements on each DE as in the previous case.

```
  do i=1, 5
    farrayPtr1d(i) = seqIndexList(i) / 10.
  enddo
```

Identical halo operations are constructed and used.

```
  call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)

  call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)

  call ESMF_ArrayHaloRelease(haloHandle, rc=rc)

  call ESMF_ArrayDestroy(array, rc=rc)
```

A current limitation of the Array implementation restricts DistGrids that contain user-specified, arbitrary sequence indices to be exactly 1D when used to create Arrays. See section 24.3 for a list of current implementation restrictions. However, an Array created on such a 1D arbitrary DistGrid is allowed to have undistributed dimensions. The following example creates an Array on the same arbitrary DistGrid, with the same arbitrary sequence indices for the halo elements as before, but with one undistributed dimension with a size of 3.

```
  if (localPet==0) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1/), undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==1) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1,2/), undistLBound=(/1/), undistUBound=(/3/), &
```

```
      rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==2) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1,2,3/), undistLBound=(/1/), undistUBound=(/3/), &
      rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==3) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      haloSeqIndexList=(/1,2,3,4/), undistLBound=(/1/), undistUBound=(/3/), &
      rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
```

By default the DistGrid dimension is mapped to the first Array dimension, associating the remaining Array dimensions with the undistributed dimensions in sequence. The dimension order is important when accessing the individual Array elements. Here the same initialization as before is extended to cover the undistributed dimension.

```
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr2d, rc=rc)
```

```
  do j=1, 3
    do i=1, 5
      farrayPtr2d(i,j) = seqIndexList(i) / 10. + 100.*j
    enddo
  enddo
```

In the context of the Array halo operation additional undistributed dimensions are treated in a simple factorized manner. The same halo association between elements that is encoded in the 1D arbitrary sequence index scheme is applied to each undistributed element separately. This is completely transparent on the user level and the same halo methods are used as before.

```
  call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)
```

```
  call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)
```

```
  call ESMF_ArrayHaloRelease(haloHandle, rc=rc)
```

```
  call ESMF_ArrayDestroy(array, rc=rc)
```

In some situations it is more convenient to associate some or all of the undistributed dimensions with the first Array dimensions. This can be done easily by explicitly mapping the DistGrid dimension to an Array dimension other than the first one. The following code creates essentially the same Array as before, but with swapped dimension order.

```
  if (localPet==0) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1/), &
      undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

```
  endif
  if (localPet==1) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1,2/), &
      undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==2) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1,2,3/), &
      undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==3) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1,2,3,4/), &
      undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
```

The swapped dimension order results in a swapping of `i` and `j` when accessing Array elements in the loop.

```
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr2d, rc=rc)
```

```
  do j=1, 3
    do i=1, 5
      farrayPtr2d(j,i) = seqIndexList(i) / 10. + 100.*j
    enddo
  enddo
```

Again there is no difference in how the the halo operations are applied.

```
  call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)
```

```
  call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)
```

```
  call ESMF_ArrayDestroy(array, rc=rc)
```

One of the benefits of mapping the undistributed dimension(s) to the "left side" of the Array dimensions is that Arrays that only differ in the size of the undistributed dimension(s) are weakly congruent in this arrangement. Weakly congruent Arrays can reuse the same RouteHandle, saving the overhead that is caused by the precompute step. In order to demonstrate this the RouteHandle of the previous halo call was not yet released and will be applied to a weakly congruent Array.
The following code creates an Array that is weakly congruent to the the previous Array by using the same input information as before, only that the size of the undistributed dimension is now 6 instead of 3.

```
  if (localPet==0) then
    array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1/), &
      undistLBound=(/1/), undistUBound=(/6/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

```
  endif
  if (localPet==1) then
    array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1,2/), &
      undistLBound=(/1/), undistUBound=(/6/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==2) then
    array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1,2,3/), &
      undistLBound=(/1/), undistUBound=(/6/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==3) then
    array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=(/1,2,3,4/), &
      undistLBound=(/1/), undistUBound=(/6/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
```

Again the exclusive Array elements must be initialized.

```
  call ESMF_ArrayGet(array2, farrayPtr=farrayPtr2d, rc=rc)


  do j=1, 6
    do i=1, 5
      farrayPtr2d(j,i) = seqIndexList(i) / 10. + 100.*j
    enddo
  enddo
```

Now the `haloHandle` that was previously pre-computed for `array` can be used directly for the weakly congruent `array2`.

```
  call ESMF_ArrayHalo(array2, routehandle=haloHandle, rc=rc)
```

Release the RouteHandle after its last use and clean up the remaining Array and DistGrid objects.

```
  call ESMF_ArrayHaloRelease(haloHandle, rc=rc)


  call ESMF_ArrayDestroy(array2, rc=rc)


  call ESMF_DistGridDestroy(distgrid, rc=rc)
```

24.2.16 Communication – Redist

Arrays used in different models often cover the same index space region, however, the distribution of the Arrays may be different, e.g. the models run on exclusive sets of PETs. Even if the Arrays are defined on the same list of PETs the decomposition may be different.

```
srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/4,1/), rc=rc)
```

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/1,4/), rc=rc)
```

The number of elements covered by `srcDistgrid` is identical to the number of elements covered by `dstDistgrid` – in fact the index space regions covered by both DistGrid objects are congruent.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

```
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, rc=rc)
```

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

By construction `srcArray` and `dstArray` are of identical type and kind. Further the number of exclusive elements matches between both Arrays. These are the prerequisites for the application of an Array redistribution in default mode. In order to increase performance of the actual redistribution the communication patter must be precomputed and stored.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The `redistHandle` can now be used repeatedly on the `srcArray`, `dstArray` pair to redistributed data from source to destination Array.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The use of the precomputed `redistHandle` is *not* restricted to `srcArray` and `dstArray`. The `redistHandle` can be used to redistribute data between any Array pairs that are weakly congruent to the Array pair used during precomputation. Arrays are congruent if they are defined on matching DistGrids and the shape of local array allocations match for all DEs. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `redistHandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
Neither `srcArray` nor `dstArray` from above hold an undistributed dimension. However, the following `srcArray1` and `dstArray1` objects are constructed to have an undistributed dimension each, that varies fastest with memory. There is only one element in the undistributed dimension in each Array.

```
call ESMF_ArraySpecSet(arrayspec3d, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
```

```
srcArray1 = ESMF_ArrayCreate(arrayspec=arrayspec3d, distgrid=srcDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1/), undistUBound=(/1/), rc=rc)
```

```
dstArray1 = ESMF_ArrayCreate(arrayspec=arrayspec3d, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1/), undistUBound=(/1/), rc=rc)
```

```
call ESMF_ArrayRedistStore(srcArray=srcArray1, dstArray=dstArray1, &
  routehandle=redistHandle, rc=rc)
```

The weak congruency feature permits the `redistHandle` to be used on Array pairs that have the same arrangement of distributed and undistributed dimensions, but where the first dimension is of different size, e.g. 10 elements instead of 1.

```
srcArray2 = ESMF_ArrayCreate(arrayspec=arrayspec3d, distgrid=srcDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1/), undistUBound=(/10/), rc=rc)

dstArray2 = ESMF_ArrayCreate(arrayspec=arrayspec3d, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1/), undistUBound=(/10/), rc=rc)

call ESMF_ArrayRedist(srcArray=srcArray2, dstArray=dstArray2, &
  routehandle=redistHandle, rc=rc)
```

When done, the resources held by `redistHandle` need to be deallocated by the user code before the handle becomes inaccessible.

```
call ESMF_ArrayRedistRelease(routehandle=redistHandle, rc=rc)
```

In *default* mode, i.e. without providing the optional `srcToDstTransposeMap` argument, `ESMF_ArrayRedistStore()` does not require equal number of dimensions in source and destination Array. Only the total number of elements must match.
Specifying `srcToDstTransposeMap` switches `ESMF_ArrayRedistStore()` into *transpose* mode. In this mode each dimension of `srcArray` is uniquely associated with a dimension in `dstArray`. The sizes of associated dimensions must match for each pair.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/20,10/), &
    rc=rc)

dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

This `dstArray` object covers a 20 x 10 index space while the `srcArray`, defined further up, covers a 10 x 20 index space. Setting `srcToDstTransposeMap = (/2,1/)` will associate the first and second dimension of `srcArray` with the second and first dimension of `dstArray`, respectively. This corresponds to a transpose of dimensions. Since the decomposition and distribution of dimensions may be different for source and destination redistribution may occur at the same time.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/2,1/), rc=rc)

call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The transpose mode of `ESMF_ArrayRedist()` is not limited to distributed dimensions of Arrays. The `srcToDstTransposeMap` argument can be used to transpose undistributed dimensions in the same manner. Furthermore transposing distributed and undistributed dimensions between Arrays is also supported.
The `srcArray` used in the following examples is of rank 4 with 2 distributed and 2 undistributed dimensions. The distributed dimensions are the two first dimensions of the Array and are distributed according to the `srcDistgrid` which describes a total index space region of 100 x 200 elements. The last two Array dimensions are undistributed dimensions of size 2 and 3, respectively.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)
```

```
srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/100,200/), &
  rc=rc)
```

```
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/2,3/), rc=rc)
```

The first `dstArray` to consider is defined on a DistGrid that also describes a 100 x 200 index space region. The distribution indicated by `dstDistgrid` may be different from the source distribution. Again the first two Array dimensions are associated with the DistGrid dimensions in sequence. Furthermore, the last two Array dimensions are undistributed dimensions, however, the sizes are 3 and 2, respectively.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/100,200/), &
  rc=rc)
```

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/3,2/), rc=rc)
```

The desired mapping between `srcArray` and `dstArray` dimensions is expressed by `srcToDstTransposeMap = (/1,2,4,3/)`, transposing only the two undistributed dimensions.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/1,2,4,3/), rc=rc)
```

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

Next consider a `dstArray` that is defined on the same `dstDistgrid`, but with a different order of Array dimensions. The desired order is specified during Array creation using the argument `distgridToArrayMap = (/2,3/)`. This map associates the first and second DistGrid dimensions with the second and third Array dimensions, respectively, leaving Array dimensions one and four undistributed.

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1,1/), undistUBound=(/3,2/), &
  rc=rc)
```

Again the sizes of the undistributed dimensions are chosen in reverse order compared to `srcArray`. The desired transpose mapping in this case will be `srcToDstTransposeMap = (/2,3,4,1/)`.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/2,3,4,1/), rc=rc)
```

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

Finally consider the case where `dstArray` is constructed on a 200 x 3 index space and where the undistributed dimensions are of size 100 and 2.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/200,3/), &
  rc=rc)
```

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/100,2/), rc=rc)
```

By construction `srcArray` and `dstArray` hold the same number of elements, albeit in a very different layout. Nevertheless, with a `srcToDstTransposeMap` that maps matching dimensions from source to destination an Array redistribution becomes a well defined operation between `srcArray` and `dstArray`.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/3,1,4,2/), rc=rc)
```

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The default mode of Array redistribution, i.e. without providing a `srcToDstTransposeMap` to `ESMF_ArrayRedistStore()`, also supports undistributed Array dimensions. The requirement in this case is that the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, be the same for source and destination Array. In this mode the number of undistributed dimensions need not match between source and destination.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)
```

```
srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/4,1/), rc=rc)
```

```
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/2,4/), rc=rc)
```

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/1,4/), rc=rc)
```

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1,1/), undistUBound=(/2,4/), &
  rc=rc)
```

Both `srcArray` and `dstArray` have two undistributed dimensions and a total count of undistributed elements of $2 \times 4 = 8$.

The Array redistribution operation is defined in terms of sequentialized undistributed dimensions. In the above case this means that a unique sequence index will be assigned to each of the 8 undistributed elements. The sequence indices will be 1, 2, ..., 8, where sequence index 1 is assigned to the first element in the first (i.e. fastest varying in memory) undistributed dimension. The following undistributed elements are labeled in consecutive order as they are stored in memory.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The redistribution operation by default applies the identity operation between the elements of undistributed dimensions. This means that source element with sequence index 1 will be mapped against destination element with sequence index 1 and so forth. Because of the way source and destination Arrays in the current example were constructed this corresponds to a mapping of dimensions 3 and 4 on `srcArray` to dimensions 1 and 4 on `dstArray`, respectively.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

Array redistribution does *not* require the same number of undistributed dimensions in source and destination Array, merely the total number of undistributed elements must match.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/1,3/), undistLBound=(/11/), undistUBound=(/18/), &
  rc=rc)
```

This `dstArray` object only has a single undistributed dimension, while the `srcArray`, defined further back, has two undistributed dimensions. However, the total undistributed element count for both Arrays is 8.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

In this case the default identity operation between the elements of undistributed dimensions corresponds to a *merging* of dimensions 3 and 4 on `srcArray` into dimension 2 on `dstArray`.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

24.2.17 Communication – SparseMatMul

Sparse matrix multiplication is a fundamental Array communication method. One frequently used application of this method is the interpolation between pairs of Arrays. The principle is this: the value of each element in the exclusive region of the destination Array is expressed as a linear combination of *potentially all* the exclusive elements of the source Array. Naturally most of the coefficients of these linear combinations will be zero and it is more efficient to store explicit information about the non-zero elements than to keep track of all the coefficients.
There is a choice to be made with respect to the format in which to store the information about the non-zero elements. One option is to store the value of each coefficient together with the corresponding destination element index and source element index. Destination and source indices could be expressed in terms of the corresponding DistGrid tile index together with the coordinate tuple within the tile. While this format may be the most natural way to express elements in the source and destination Array, it has two major drawbacks. First the coordinate tuple is `dimCount` specific and second the format is extremely bulky. For 2D source and destination Arrays it would require 6 integers to store the source and destination element information for each non-zero coefficient and matters get worse for higher dimensions.
Both problems can be circumvented by *interpreting* source and destination Arrays as sequentialized strings or *vectors* of elements. This is done by assigning a unique *sequence index* to each exclusive element in both Arrays. With that the operation of updating the elements in the destination Array as linear combinations of source Array elements takes the form of a *sparse matrix multiplication*.
The default sequence index rule assigns index 1 to the `minIndex` corner element of the first tile of the DistGrid on which the Array is defined. It then increments the sequence index by 1 for each element running through the DistGrid dimensions by order. The index space position of the DistGrid tiles does not affect the sequence labeling of elements. The default sequence indices for

```
srcDistgrid = ESMF_DistGridCreate(minIndex=(/-1,0/), maxIndex=(/1,3/), rc=rc)
```

for each element are:

```
 -------------------------------------> 2nd dim
 |
 |    +------+------+------+------+
 |    |(-1,0)|      |      |(-1,3)|
 |    |      |      |      |      |
 |    |   1  |   4  |   7  |  10  |
 |    +------+------+------+------+
 |    |      |      |      |      |
 |    |      |      |      |      |
 |    |   2  |   5  |   8  |  11  |
 |    +------+------+------+------+
 |    | (1,0)|      |      | (1,3)|
 |    |      |      |      |      |
 |    |   3  |   6  |   9  |  12  |
 |    +------+------+------+------+
 |
 v
1st dim
```

The assigned sequence indices are decomposition and distribution invariant by construction. Furthermore, when an Array is created with extra elements per DE on a DistGrid the sequence indices (which only cover the exclusive elements) remain unchanged.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)


srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
```

The extra padding of 1 element in each direction around the exclusive elements on each DE are "invisible" to the Array spare matrix multiplication method. These extra elements are either updated by the computational kernel or by Array halo operations (not yet implemented!).
An alternative way to assign sequence indices to all the elements in the tiles covered by a DistGrid object is to use a special `ESMF_DistGridCreate()` call. This call has been specifically designed for 1D cases with arbitrary, user-supplied sequence indices.

```
seqIndexList(1) = localPet*10
seqIndexList(2) = localPet*10 + 1
dstDistgrid = ESMF_DistGridCreate(arbSeqIndexList=seqIndexList, rc=rc)
```

This call to `ESMF_DistGridCreate()` is collective across the current VM. The `arbSeqIndexList` argument specifies the PET-local arbitrary sequence indices that need to be covered by the local DE. The resulting DistGrid has one local DE per PET which covers the entire PET-local index range. The user supplied sequence indices must be unique, but the sequence may be interrupted. The four DEs of `dstDistgrid` have the following local 1D index space coordinates (given between "()") and sequence indices:

```
  covered by DE 0    covered by DE 1   covered by DE 2   covered by DE 3
  on PET 0           on PET 1          on PET 2          on PET 3
```

```
   ----------------------------------------------------------------------
   (1) : 0              (1) : 10          (1) : 20          (1) : 30
   (2) : 1              (2) : 11          (2) : 21          (2) : 31
```

Again the DistGrid object provides the sequence index labeling for the exclusive elements of an Array created on the DistGrid regardless of extra, non-exclusive elements.

```
  dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

With the definition of sequence indices, either by the default rule or as user provided arbitrary sequence indices, it is now possible to uniquely identify each exclusive element in the source and destination Array by a single integer number. Specifying a pair of source and destination elements takes two integer number regardless of the number of dimensions.

The information required to carry out a sparse matrix multiplication are the pair of source and destination sequence indices and the associated multiplication factor for each pair. ESMF requires this information in form of two Fortran arrays. The factors are stored in a 1D array of the appropriate type and kind, e.g. `real(ESMF_KIND_R8)::factorList(:)`. Array sparse matrix multiplications are supported between Arrays of different type and kind. The type and kind of the factors can also be chosen freely. The sequence index pairs associated with the factors provided by `factorList` are stored in a 2D Fortran array of default integer kind of the shape `integer::factorIndexList(2,:)`. The sequence indices of the source Array elements are stored in the first row of `factorIndexList` while the sequence indices of the destination Array elements are stored in the second row.

Each PET in the current VM must call into `ESMF_ArraySMMStore()` to precompute and store the communication pattern for the sparse matrix multiplication. The multiplication factors may be provided in parallel, i.e. multiple PETs may specify `factorList` and `factorIndexList` arguments when calling into `ESMF_ArraySMMStore()`. PETs that do not provide factors either call with `factorList` and `factorIndexList` arguments containing zero elements or issue the call omitting both arguments.

```
  if (localPet == 0) then
    allocate(factorList(1))               ! PET 0 specifies 1 factor
    allocate(factorIndexList(2,1))
    factorList = (/0.2/)                  ! factors
    factorIndexList(1,:) = (/5/)          ! seq indices into srcArray
    factorIndexList(2,:) = (/30/)         ! seq indices into dstArray

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, factorList=factorList, &
      factorIndexList=factorIndexList, rc=rc)

    deallocate(factorList)
    deallocate(factorIndexList)
  else if (localPet == 1) then
    allocate(factorList(3))               ! PET 1 specifies 3 factor
    allocate(factorIndexList(2,3))
    factorList = (/0.5, 0.5, 0.8/)        ! factors
    factorIndexList(1,:) = (/8, 2, 12/)   ! seq indices into srcArray
    factorIndexList(2,:) = (/11, 11, 30/) ! seq indices into dstArray

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, factorList=factorList, &
      factorIndexList=factorIndexList, rc=rc)

    deallocate(factorList)
    deallocate(factorIndexList)
  else
```

```
    ! PETs 2 and 3 do not provide factors

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, rc=rc)

  endif
```

The RouteHandle object `sparseMatMulHandle` produced by `ESMF_ArraySMMStore()` can now be used to call `ESMF_ArraySMM()` collectively across all PETs of the current VM to perform

```
     dstArray = 0.0
     do n=1, size(combinedFactorList)
         dstArray(combinedFactorIndexList(2, n)) +=
           combinedFactorList(n) * srcArray(combinedFactorIndexList(1, n))
     enddo
```

in parallel. Here `combinedFactorList` and `combinedFactorIndexList` are the combined lists defined by the respective local lists provided by PETs 0 and 1 in parallel. For this example

```
  call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, rc=rc)
```

will initialize the entire `dstArray` to 0.0 and then update two elements:

```
   on DE 1:
   dstArray(2) = 0.5 * srcArray(0,0)  +  0.5 * srcArray(0,2)
```

and

```
   on DE 3:
   dstArray(1) = 0.2 * srcArray(0,1)  +  0.8 * srcArray(1,3).
```

The call to `ESMF_ArraySMM()` does provide the option to turn the default `dstArray` initialization off. If argument `zeroregion` is set to `ESMF_REGION_EMPTY`

```
  call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, zeroregion=ESMF_REGION_EMPTY, rc=rc)
```

skips the initialization and elements in `dstArray` are updated according to:

```
     do n=1, size(combinedFactorList)
         dstArray(combinedFactorIndexList(2, n)) +=
           combinedFactorList(n) * srcArray(combinedFactorIndexList(1, n)).
     enddo
```

The `ESMF_RouteHandle` object returned by `ESMF_ArraySMMStore()` can be applied to any src/dst Array pairs that are weakly congruent to the Array pair used during precomputation. Arrays are congruent if they are defined on matching DistGrids and the shape of local array allocations match for all DEs. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. See section 24.2.16 for an example of this feature demonstrated for the Redist case. The exact same principle applies to the SMM case.
The resources held by `sparseMatMulHandle` need to be deallocated by the user code before the handle becomes inaccessible.

```
call ESMF_ArraySMMRelease(routehandle=sparseMatMulHandle, rc=rc)
```

The Array sparse matrix multiplication also applies to Arrays with undistributed dimensions. The undistributed dimensions are interpreted in a sequentialized manner, much like the distributed dimensions, introducing a second sequence index for source and destination elements. Sequence index 1 is assigned to the first element in the first (i.e. fastest varying in memory) undistributed dimension. The following undistributed elements are labeled in consecutive order as they are stored in memory.
In the simplest case the Array sparse matrix multiplication will apply an identity matrix to the vector of sequentialized undistributed Array elements for every non-zero element in the sparse matrix. The requirement in this case is that the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, be the same for source and destination Array.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
  distgridToArrayMap=(/1,2/), undistLBound=(/1/), undistUBound=(/2/), rc=rc)


call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2/), undistLBound=(/1/), undistUBound=(/2/), rc=rc)
```

Setting up `factorList` and `factorIndexList` is identical to the case for Arrays without undistributed dimensions. Also the call to `ESMF_ArraySMMStore()` remains unchanged. Internally, however, the source and destination Arrays are checked to make sure the total undistributed element count matches.

```
if (localPet == 0) then
  allocate(factorList(1))               ! PET 0 specifies 1 factor
  allocate(factorIndexList(2,1))
  factorList = (/0.2/)                  ! factors
  factorIndexList(1,:) = (/5/)          ! seq indices into srcArray
  factorIndexList(2,:) = (/30/)         ! seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
    factorIndexList=factorIndexList, rc=rc)

  deallocate(factorList)
  deallocate(factorIndexList)
else if (localPet == 1) then
  allocate(factorList(3))               ! PET 1 specifies 3 factor
  allocate(factorIndexList(2,3))
  factorList = (/0.5, 0.5, 0.8/)        ! factors
  factorIndexList(1,:) = (/8, 2, 12/)   ! seq indices into srcArray
  factorIndexList(2,:) = (/11, 11, 30/) ! seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
    factorIndexList=factorIndexList, rc=rc)

  deallocate(factorList)
  deallocate(factorIndexList)
else
  ! PETs 2 and 3 do not provide factors
```

```
    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, rc=rc)
  endif
```

The call into the `ESMF_ArraySMM()` operation is completely transparent with respect to whether source and/or destination Arrays contain undistributed dimensions.

```
  call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, rc=rc)
```

This operation will initialize the entire `dstArray` to 0.0 and then update four elements:

```
   on DE 1:
   dstArray[1](2) = 0.5 * srcArray(0,0)[1]  +  0.5 * srcArray(0,2)[1],
   dstArray[2](2) = 0.5 * srcArray(0,0)[2]  +  0.5 * srcArray(0,2)[2]
```

and

```
   on DE 3:
   dstArray[1](1) = 0.2 * srcArray(0,1)[1]  +  0.8 * srcArray(1,3)[1],
   dstArray[2](1) = 0.2 * srcArray(0,1)[2]  +  0.8 * srcArray(1,3)[2].
```

Here indices between "()" refer to distributed dimensions while indices between "[]" correspond to undistributed dimensions.
In a more general version of the Array sparse matrix multiplication the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, need not be the same for source and destination Array. In this formulation each non-zero element of the sparse matrix is identified with a unique element in the source and destination Array. This requires a generalization of the `factorIndexList` argument which now must contain four integer numbers for each element. These numbers in sequence are the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the source Array, followed by the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the destination Array.

```
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
  srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
    totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
    distgridToArrayMap=(/1,2/), undistLBound=(/1/), undistUBound=(/2/), rc=rc)


  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
  dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
    distgridToArrayMap=(/2/), undistLBound=(/1/), undistUBound=(/4/), rc=rc)
```

Setting up `factorList` is identical to the previous cases since there is still only one value associated with each non-zero matrix element. However, each entry in `factorIndexList` now has 4 instead of just 2 components.

```
  if (localPet == 0) then
    allocate(factorList(1))              ! PET 0 specifies 1 factor
    allocate(factorIndexList(4,1))
    factorList = (/0.2/)                 ! factors
    factorIndexList(1,:) = (/5/)         ! seq indices into srcArray
    factorIndexList(2,:) = (/1/)         ! undistr. seq indices into srcArray
```

```
    factorIndexList(3,:) = (/30/)         ! seq indices into dstArray
    factorIndexList(4,:) = (/2/)          ! undistr. seq indices into dstArray

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, factorList=factorList, &
      factorIndexList=factorIndexList, rc=rc)

    deallocate(factorList)
    deallocate(factorIndexList)
  else if (localPet == 1) then
    allocate(factorList(3))               ! PET 1 specifies 3 factor
    allocate(factorIndexList(4,3))
    factorList = (/0.5, 0.5, 0.8/)        ! factors
    factorIndexList(1,:) = (/8, 2, 12/)   ! seq indices into srcArray
    factorIndexList(2,:) = (/2, 1, 1/)    ! undistr. seq indices into srcArray
    factorIndexList(3,:) = (/11, 11, 30/) ! seq indices into dstArray
    factorIndexList(4,:) = (/4, 4, 2/)    ! undistr. seq indices into dstArray

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, factorList=factorList, &
      factorIndexList=factorIndexList, rc=rc)

    deallocate(factorList)
    deallocate(factorIndexList)
  else
    ! PETs 2 and 3 do not provide factors

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, rc=rc)
  endif
```

The call into the `ESMF_ArraySMM()` operation remains unchanged.

```
  call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, rc=rc)
```

This operation will initialize the entire `dstArray` to 0.0 and then update two elements:

```
 on DE 1:
 dstArray[4](2) = 0.5 * srcArray(0,0)[1]  +  0.5 * srcArray(0,2)[2],
```

and

```
 on DE 3:
 dstArray[2](1) = 0.2 * srcArray(0,1)[1]  +  0.8 * srcArray(1,3)[1],
```

Here indices in () refer to distributed dimensions while indices in [] correspond to undistributed dimensions.

24.2.18 Communication – Scatter and Gather, revisited

The `ESMF_ArrayScatter()` and `ESMF_ArrayGather()` calls, introduced in section 24.2.13, provide a convenient way of communicating data between a Fortran array and all of the DEs of a single Array tile. A key requirement of `ESMF_ArrayScatter()` and `ESMF_ArrayGather()` is that the *shape* of the Fortran array and the Array tile must match. This means that the `dimCount` must be equal, and that the size of each dimension must match. Element reordering during scatter and gather is only supported on a per dimension level, based on the `decompflag` option available during DistGrid creation.

While the `ESMF_ArrayScatter()` and `ESMF_ArrayGather()` methods cover a broad, and important spectrum of cases, there are situations that require a different set of rules to scatter and gather data between a Fortran array and an ESMF Array object. For instance, it is often convenient to create an Array on a DistGrid that was created with arbitrary, user-supplied sequence indices. See section 31.3.6 for more background on DistGrids with arbitrary sequence indices.

```
allocate(arbSeqIndexList(10))   ! each PET will have 10 elements

do i=1, 10
  arbSeqIndexList(i) = (i-1)*petCount + localPet+1 ! initialize unique
                                                   ! seq. indices
enddo

distgrid = ESMF_DistGridCreate(arbSeqIndexList=arbSeqIndexList, rc=rc)


deallocate(arbSeqIndexList)

call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=1, rc=rc)


array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

This `array` object holds 10 elements on each DE, and there is one DE per PET, for a total element count of 10 x `petCount`. The `arbSeqIndexList`, used during DistGrid creation, was constructed cyclic across all DEs. DE 0, for example, on a 4 PET run, would hold sequence indices 1, 5, 9, ... DE 1 would hold 2, 6, 10, ..., and so on.

The usefulness of the user-specified arbitrary sequence indices becomes clear when they are interpreted as global element ids. The ArrayRedist() and ArraySMM() communication methods are based on sequence index mapping between source and destination Arrays. Other than providing a canonical sequence index order via the default sequence scheme, outlined in 24.2.17, ESMF does not place any restrictions on the sequence indices. Objects that were not created with user supplied sequence indices default to the ESMF sequence index order.

A common, and useful interpretation of the arbitrary sequence indices, specified during DistGrid creation, is that of relating them to the canonical ESMF sequence index order of another data object. Within this interpretation the `array` object created above could be viewed as an arbitrary distribution of a (`petCount` x 10) 2D array.

```
if (localPet == 0) then
  allocate(farray(petCount,10)) ! allocate 2D Fortran array petCount x 10
  do j=1, 10
    do i=1, petCount
      farray(i,j) = 100 + (j-1)*petCount + i    ! initialize to something
    enddo
  enddo
else
  allocate(farray(0,0)) ! must allocate an array of size 0 on all other PETs
endif
```

For a 4 PET run, `farray` on PET 0 now holds the following data.

```
   -----1----2----3------------10-----> j
   |
   1   101, 105, 109, ....  , 137
   |
   2   102, 106, 110, ....  , 138
```

```
 |
 3   103, 107, 111, ....  , 139
 |
 4   104, 108, 112, ....  , 140
 |
 |
 v
i
```

On all other PETs `farray` has a zero size allocation.
Following the sequence index interpretation from above, scattering the data contained in `farray` on PET 0 across the `array` object created further up, seems like a well defined operation. Looking at it a bit closer, it becomes clear that it is in fact more of a redistribution than a simple scatter operation. The general rule for such a "redist-scatter" operation, of a Fortran array, located on a single PET, into an ESMF Array, is to use the canonical ESMF sequence index scheme to label the elements of the Fortran array, and to send the data to the Array element with the same sequence index.
The just described "redist-scatter" operation is much more general than the standard `ESMF_ArrayScatter()` method. It does not require shape matching, and supports full element reordering based on the sequence indices. Before `farray` can be scattered across `array` in the described way, it must be wrapped into an ESMF Array object itself, essentially labeling the array elements according to the canonical sequence index scheme.

```
distgridAux = ESMF_DistGridCreate(minIndex=(/1,1/), &
  maxIndex=(/petCount,10/), &
  regDecomp=(/1,1/), rc=rc) ! DistGrid with only 1 DE
```

The first step is to create a DistGrid object with only a single DE. This DE must be located on the PET on which the Fortran data array resides. In this example `farray` holds data on PET 0, which is where the default DELayout will place the single DE defined in the DistGrid. If the `farray` was setup on a different PET, an explicit DELayout would need to be created first, mapping the only DE to the PET on which the data is defined.
Next the Array wrapper object can be created from the `farray` and the just created DistGrid object.

```
arrayAux = ESMF_ArrayCreate(farray=farray, distgrid=distgridAux, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

At this point all of the pieces are in place to use `ESMF_ArrayRedist()` to do the "redist-scatter" operation. The typical store/execute/release pattern must be followed.

```
call ESMF_ArrayRedistStore(srcArray=arrayAux, dstArray=array, &
  routehandle=scatterHandle, rc=rc)
```

```
call ESMF_ArrayRedist(srcArray=arrayAux, dstArray=array, &
  routehandle=scatterHandle, rc=rc)
```

In this example, after `ESMF_ArrayRedist()` was called, the content of `array` on a 4 PET run would look like this:

```
PET 0:   101, 105, 109, ....  , 137
PET 1:   102, 106, 110, ....  , 138
PET 2:   103, 107, 111, ....  , 139
PET 3:   104, 108, 112, ....  , 140
```

Once set up, `scatterHandle` can be used repeatedly to scatter data from `farray` on PET 0 to all the DEs of `array`. All of the resources should be released once `scatterHandle` is no longer needed.

```
call ESMF_ArrayRedistRelease(routehandle=scatterHandle, rc=rc)
```

The opposite operation, i.e. *gathering* of the `array` data into `farray` on PET 0, follows a very similar setup. In fact, the `arrayAux` object already constructed for the scatter direction, can directly be re-used. The only thing that is different for the "redist-gather", are the `srcArray` and `dstArray` argument assignments, reflecting the opposite direction of data movement.

```
call ESMF_ArrayRedistStore(srcArray=array, dstArray=arrayAux, &
  routehandle=gatherHandle, rc=rc)
```

```
call ESMF_ArrayRedist(srcArray=array, dstArray=arrayAux, &
  routehandle=gatherHandle, rc=rc)
```

Just as for the scatter case, the `gatherHandle` can be used repeatedly to gather data from `array` into `farray` on PET 0. All of the resources should be released once `gatherHandle` is no longer needed.

```
call ESMF_ArrayRedistRelease(routehandle=gatherHandle, rc=rc)
```

Finally the wrapper Array `arrayAux` and the associated DistGrid object can also be destroyed.

```
call ESMF_ArrayDestroy(arrayAux, rc=rc)
```

```
call ESMF_DistGridDestroy(distgridAux, rc=rc)
```

Further, the primary data objects of this example must be deallocated and destroyed.

```
deallocate(farray)

call ESMF_ArrayDestroy(array, rc=rc)
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

24.2.19 Non-blocking Communications

All `ESMF_RouteHandle` based communcation methods, like `ESMF_ArrayRedist()`, `ESMF_ArrayHalo()` and `ESMF_ArraySMM()`, can be executed in blocking or non-blocking mode. The non-blocking feature is useful, for example, to overlap computation with communication, or to implement a more loosely synchronized inter-Component interaction scheme than is possible with the blocking communication mode.
Access to the non-blocking execution mode is provided uniformly across all RouteHandle based communication calls. Every such call contains the optional `routesyncflag` argument of type `ESMF_RouteSync_Flag`. Section 9.38 lists all of the valid settings for this flag.
It is an execution time decision to select whether to invoke a precomputed communication pattern, stored in a RouteHandle, in the blocking or non-blocking mode. Neither requires specifically precomputed RouteHandles - i.e. a RouteHandle is neither specifically blocking nor specifically non-blocking.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, rc=rc)
```

The returned RouteHandle `routehandle` can be used in blocking or non-blocking execution calls. The application is free to switch between both modes for the same RouteHandle.

By default `routesyncflag` is set to `ESMF_ROUTESYNC_BLOCKING` in all of the RouteHandle execution methods, and the behavior is that of the VM-wide collective communication calls described in the previous sections. In the blocking mode the user must assume that the communication call will not return until all PETs have exchanged the precomputed information. On the other hand, the user has no guarante about the exact synchronization behavior, and it is unsafe to make specific assumtions. What is guaranteed in the blocking communication mode is that when the call returns on the local PET, all data exchanges associated with all local DEs have finished. This means that all in-bound data elements are valid and that all out-bound data elements can safely be overwritten by the user.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_BLOCKING, rc=rc)
```

The same exchange pattern, that is encoded in `routehandle`, can be executed in non-blocking mode, simply by setting the appropriate `routesyncflag` when calling into `ESMF_ArrayRedist()`.

At first sight there are obvious similarities between the non-blocking RouteHandle based execution paradigm and the non-blocking message passing calls provided by MPI. However, there are significant differences in the behavior of the non-blocking point-to-point calls that MPI defines and the non-blocking mode of the collective exchange patterns described by ESMF RouteHandles.

Setting `routesyncflag` to `ESMF_ROUTESYNC_NBSTART` in any RouteHandle execution call returns immediatly after all out-bound data has been moved into ESMF internal transfer buffers and the exchange has been initiated.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_NBSTART, rc=rc)
```

Once a call with `routesyncflag = ESMF_ROUTESYNC_NBSTART` returns, it is safe to modify the out-bound data elements in the `srcArray` object. However, no guarantees are made for the in-bound data elements in `dstArray` at this phase of the non-blocking execution. It is unsafe to access these elements until the exchange has finished locally.

One way to ensure that the exchange has finished locally is to call with `routesyncflag` set to `ESMF_ROUTESYNC_NBWAITFINISH`.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_NBWAITFINISH, rc=rc)
```

Calling with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH` instructs the communication method to wait and block until the previously started exchange has finished, and has been processed locally according to the RouteHandle. Once the call returns, it is safe to access both in-bound and out-bound data elements in `dstArray` and `srcArray`, respectively.

Some situations require more flexibility than is provided by the `ESMF_ROUTESYNC_NBSTART` - `ESMF_ROUTESYNC_NBWAITFINISH` pair. For instance, a Component that needs to interact with several other Components, virtually simultanously, would initiated several different exchanges with `ESMF_ROUTESYNC_NBSTART`. Calling with `ESMF_ROUTESYNC_NBWAITFINISH` for any of the outstanding exchanges may potentially block for a long time, lowering the throughput. In the worst case a dead lock situation may arrise. Calling with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH` addresses this problem.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_NBTESTFINISH, &
  finishedflag=finishflag, rc=rc)
```

This call tests the locally outstanding data transfer operation in `routehandle`, and finishes the exchange as much as currently possible. It does not block until the entire exchange has finished locally, instead it returns immediatly after

one round of testing has been completed. The optional return argument `finishedflag` is set to `.true.` if the exchange is completely finished locally, and set to `.false.` otherwise.
The user code must decide, depending on the value of the returned `finishedflag`, whether additional calls are required to finish an outstanding non-blocking exchange. If so, it can be done by calling `ESMF_ArrayRedist()` repeatly with `ESMF_ROUTESYNC_NBTESTFINISH` until `finishedflag` comes back with a value of `.true.`. Such a loop allows other pieces of user code to be executed between the calls. A call with `ESMF_ROUTESYNC_NBWAITFINISH` can alternatively be used to block until the exchange has locally finished.
Noteworthy property. It is allowable to invoke a RouteHandle based communication call with `routesyncflag` set to `ESMF_ROUTESYNC_NBTESTFINISH` or `ESMF_ROUTESYNC_NBWAITFINISH` on a specific RouteHandle without there being an outstanding non-blocking exchange. As a matter of fact, it is not required that there was ever a call made with `ESMF_ROUTESYNC_NBSTART` for the RouteHandle. In these cases the calls made with `ESMF_ROUTESYNC_NBTESTFINISH` or `ESMF_ROUTESYNC_NBWAITFINISH` will simply return immediately (with `finishedflag` set to `.true.`).
Noteworthy property. It is fine to mix blocking and non-blocking invokations of the same RouteHandle based communication call across the PETs. This means that it is fine for some PETs to issue the call with `ESMF_ROUTESYNC_BLOCKING` (or using the default), while other PETs call the same communication call with `ESMF_ROUTESYNC_NBSTART`.
Noteworthy restriction. A RouteHandle that is currently involved in an outstanding non-blocking exchange may *not* be used to start any further exchanges, neither blocking nor non-blocking. This restriction is independent of whether the newly started RouteHandle based exchange is made for the same or for different data objects.

24.3 Restrictions and Future Work

- **CAUTION:** Depending on the specific `ESMF_ArrayCreate()` entry point used during Array creation, certain Fortran operations are not supported on the Fortran array pointer `farrayPtr`, returned by `ESMF_ArrayGet()`. Only if the `ESMF_ArrayCreate()` *from pointer* variant was used, will the returned `farrayPtr` variable contain the original bounds information, and be suitable for the Fortran `deallocate()` call. This limitation is a direct consequence of the Fortran 95 standard relating to the passing of array arguments. Fortran array pointers returned from an Array that was created through the *assumed shape array* variant of `ESMF_ArrayCreate()` will have bounds that are consistent with the other arguments specified during Array creation. These pointers are not suitable for deallocation in accordance to the Fortran 95 standard.
- **1D limit:** ArrayHalo(), ArrayRedist() and ArraySMM() operations on Arrays created on DistGrids with arbitrary sequence indices are currently limited to 1D arbitrary DistGrids. There is no restriction on the number, size and mapping of undistributed Array dimensions in the presence of such a 1D arbitrary DistGrid.

24.4 Design and Implementation Notes

The Array class is part of the ESMF index space layer and is built ontop of the DistGrid and DELayout classes. The DELayout class introduces the notion of *decomposition elements* (DEs) and their layout across the available PETs. The DistGrid describes how index space is decomposed by assigning *logically rectangular index space pieces* or *DE-local tiles* to the DEs. The Array finally associates a *local memory allocation* with each local DE.
The following is a list of implementation specific details about the current ESMF Array.

- Implementation language is C++.
- Local memory allocations are internally held in `ESMF_LocalArray` objects.
- All precomputed communication methods are based on sparse matrix multiplication.

24.5 Class API

24.5.1 ESMF_ArrayAssignment(=) - Array assignment

INTERFACE:

```
interface assignment(=)
array1 = array2
```

ARGUMENTS:

```
type(ESMF_Array) :: array1
type(ESMF_Array) :: array2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign array1 as an alias to the same ESMF Array object in memory as array2. If array2 is invalid, then array1 will be equally invalid after the assignment.
The arguments are:

array1 The `ESMF_Array` object on the left hand side of the assignment.

array2 The `ESMF_Array` object on the right hand side of the assignment.

24.5.2 ESMF_ArrayOperator(==) - Array equality operator

INTERFACE:

```
interface operator(==)
if (array1 == array2) then ... endif
OR
result = (array1 == array2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array1
type(ESMF_Array), intent(in) :: array2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether array1 and array2 are valid aliases to the same ESMF Array object in memory. For a more general comparison of two ESMF Arrays, going beyond the simple alias test, the ESMF_ArrayMatch() function (not yet implemented) must be used.
The arguments are:

array1 The `ESMF_Array` object on the left hand side of the equality operation.

array2 The `ESMF_Array` object on the right hand side of the equality operation.

24.5.3 ESMF_ArrayOperator(/=) - Array not equal operator

INTERFACE:

```
interface operator(/=)
if (array1 /= array2) then ... endif
OR
result = (array1 /= array2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array1
type(ESMF_Array), intent(in) :: array2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether array1 and array2 are *not* valid aliases to the same ESMF Array object in memory. For a more general comparison of two ESMF Arrays, going beyond the simple alias test, the ESMF_ArrayMatch() function (not yet implemented) must be used.
The arguments are:

array1 The ESMF_Array object on the left hand side of the non-equality operation.

array2 The ESMF_Array object on the right hand side of the non-equality operation.

24.5.4 ESMF_ArrayCreate - Create Array object from Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateFromPtr<rank><type><kind>(distgrid, farrayPtr, &
datacopyflag, distgridToArrayMap, computationalEdgeLWidth, &
computationalEdgeUWidth, computationalLWidth, &
computationalUWidth, totalLWidth, &
totalUWidth, name, rc)
```

ARGUMENTS:

```
  type(ESMF_DistGrid), intent(in) :: distgrid
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: distgridToArrayMap(:)
  integer, intent(in), optional :: computationalEdgeLWidth(:)
  integer, intent(in), optional :: computationalEdgeUWidth(:)
  integer, intent(in), optional :: computationalLWidth(:)
  integer, intent(in), optional :: computationalUWidth(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  character (len=*), intent(in), optional :: name
  integer, intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataPtr<rank><type><kind>
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object from existing local native Fortran arrays with pointer attribute, according to distgrid. Besides `farrayPtr` each PET must issue this call with identical arguments in order to create a consistent Array object. The bounds of the local arrays are preserved by this call and determine the bounds of the total region of the resulting Array object. Bounds of the DE-local exclusive regions are set to be consistent with the total regions and the specified distgrid argument. Bounds for Array dimensions that are not distributed are automatically set to the bounds provided by `farrayPtr`.
This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.
The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. By default all tensor elements are associated with stagger location 0. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use `ESMF_ArraySet()` to change these default settings after the Array object has been created.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of `farrayPtr`.

farrayPtr Valid native Fortran array with pointer attribute. Memory must be associated with the actual argument. The type/kind/rank information of `farrayPtr` will be used to set `Array`'s properties accordingly. The shape of `farrayPtr` will be checked against the information contained in the `distgrid`. The bounds of `farrayPtr` will be preserved by this call and the bounds of the resulting Array object are set accordingly.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by `farrayPtr` directly or will copy the data from `farrayPtr` into a new memory allocation. Valid options are `ESMF_DATACOPY_REFERENCE` (default) or `ESMF_DATACOPY_VALUE`. Depending on the specific situation the `ESMF_DATACOPY_REFERENCE` option may be unsafe when specifying an array slice for `farrayPtr`.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in `farrayPtr` by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the `farrayPtr` argument in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped `farrayPtr` dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the computational region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `farrayPtr` argument.

[name] Name of the Array object.

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

24.5.5 ESMF_ArrayCreate - Create Array object from Fortran array pointer w/ arbitrary seqIndices for halo

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateFromPtrArb<rank><type><kind>(distgrid, farrayPtr, &
haloSeqIndexList, keywordEnforcer datacopyflag, distgridToArrayMap, &
name, rc)
```

ARGUMENTS:

```
   type(ESMF_DistGrid), intent(in) :: distgrid
   <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
   integer, intent(in) :: haloSeqIndexList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   integer, intent(in), optional :: distgridToArrayMap(:)
   character (len=*), intent(in), optional :: name
   integer, intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataPtrArb<rank><type><kind>
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object from existing local native Fortran arrays with pointer attribute, according to distgrid. Besides `farrayPtr` each PET must issue this call with identical arguments in order to create a consistent Array object. The bounds of the local arrays are preserved by this call and determine the bounds of the total region of the resulting Array object. Bounds of the DE-local exclusive regions are set to be consistent with the total regions and the specified distgrid argument. Bounds for Array dimensions that are not distributed are automatically set to the bounds provided by `farrayPtr`.
This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. By default all tensor elements are associated with stagger location 0. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use `ESMF_ArraySet()` to change these default settings after the Array object has been created.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of `farrayPtr`.

farrayPtr Valid native Fortran array with pointer attribute. Memory must be associated with the actual argument. The type/kind/rank information of `farrayPtr` will be used to set `Array`'s properties accordingly. The shape of `farrayPtr` will be checked against the information contained in the `distgrid`. The bounds of `farrayPtr` will be preserved by this call and the bounds of the resulting Array object are set accordingly.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of `haloSeqIndexList` can (and typically will) be different on each PET.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by `farrayPtr` directly or will copy the data from `farrayPtr` into a new memory allocation. Valid options are `ESMF_DATACOPY_REFERENCE` (default) or `ESMF_DATACOPY_VALUE`. Depending on the specific situation the `ESMF_DATACOPY_REFERENCE` option may be unsafe when specifying an array slice for `farrayPtr`.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in `farrayPtr` by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the `farrayPtr` argument in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped `farrayPtr` dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[name] Name of the Array object.

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

24.5.6 ESMF_ArrayCreate - Create Array object from Fortran array

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAssmdShape<rank><type><kind>(distgrid, farray, &
indexflag, datacopyflag, distgridToArrayMap, &
computationalEdgeLWidth, computationalEdgeUWidth, computationalLWidth, &
computationalUWidth, totalLWidth, &
totalUWidth, undistLBound, undistUBound, name, rc)
```

ARGUMENTS:

```
  type(ESMF_DistGrid), intent(in) :: distgrid
  <type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
  type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: computationalEdgeLWidth(:)
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataAssmdShape<rank><type><kind>
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object from an existing local native Fortran array according to distgrid. Besides `farray` each PET must issue this call with identical arguments in order to create a consistent Array object. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for Array dimensions that are not distributed can be chosen freely using the `undistLBound` and `undistUBound` arguments.
This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.
The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. By default all tensor elements are associated with stagger location 0. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use `ESMF_ArraySet()` to change these default settings after the Array object has been created.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of farray.

farray Valid native Fortran array, i.e. memory must be associated with the actual argument. The type/kind/rank information of `farray` will be used to set `Array`'s properties accordingly. The shape of `farray` will be checked against the information contained in the `distgrid`.

indexflag Indicate how DE-local indices are defined. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by `farray` directly or will copy the data from `farray` into a new memory allocation. Valid options are `ESMF_DATACOPY_REFERENCE` (default) or `ESMF_DATACOPY_VALUE`. Depending on the specific situation the `ESMF_DATACOPY_REFERENCE` option may be unsafe when specifying an array slice for `farray`.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in `farray` by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the `farray` argument in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped `farray` dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the

DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the computational region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `farray` argument.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `farray`.

[name] Name of the Array object.

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

24.5.7 ESMF_ArrayCreate - Create Array object from a list of LocalArray objects

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateLocalArray(distgrid, localarrayList, &
  indexflag, datacopyflag, distgridToArrayMap, computationalEdgeLWidth, &
  computationalEdgeUWidth, computationalLWidth, computationalUWidth, &
  totalLWidth, totalUWidth, undistLBound, undistUBound, name, rc)
```

ARGUMENTS:

```
      type(ESMF_DistGrid), intent(in) :: distgrid
      type(ESMF_LocalArray), intent(in) :: localarrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Index_Flag), intent(in), optional :: indexflag
      type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
      integer, intent(in), optional :: distgridToArrayMap(:)
      integer, intent(in), optional :: computationalEdgeLWidth(:)
```

```
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateLocalArray
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object from existing `ESMF_LocalArray` objects according to distgrid. Besides `localarrayList` each PET must issue this call with identical arguments in order to create a consistent Array object. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for array dimensions that are not distributed can be chosen freely using the `undistLBound` and `undistUBound` arguments.
This interface is able to handle multiple DEs per PET.
The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. By default all tensor elements are associated with stagger location 0. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use `ESMF_ArraySet()` to change these default settings after the Array object has been created.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

localarrayList List of valid `ESMF_LocalArray` objects, i.e. memory must be associated with the actual arguments. The type/kind/rank information of all `localarrayList` elements must be identical and will be used to set `Array`'s properties accordingly. The shape of each `localarrayList` element will be checked against the information contained in the `distgrid`.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section 9.24 for a list of valid indexflag options.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by `farray` directly or will copy the data from `farray` into a new memory allocation. Valid options are `ESMF_DATACOPY_REFERENCE` (default) or `ESMF_DATACOPY_VALUE`. Depending on the specific situation the `ESMF_DATACOPY_REFERENCE` option may be unsafe when specifying an array slice for `farray`.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the `localarrayList` elements by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the `localarrayList` elements in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the `localarrayList` elements are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `localarrayList` argument.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `localarrayList`.

[name] Name of the Array object.

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

24.5.8 ESMF_ArrayCreate - Create Array object from a list of LocalArray objects w/ arbitrary seqIndices for halo

INTERFACE:

```
   ! Private name; call using ESMF_ArrayCreate()
   function ESMF_ArrayCreateLocalArrayArb(distgrid, localarrayList, &
     haloSeqIndexList, datacopyflag, distgridToArrayMap, undistLBound, &
     undistUBound, name, rc)
```

ARGUMENTS:

```
        type(ESMF_DistGrid), intent(in) :: distgrid
        type(ESMF_LocalArray), intent(in) :: localarrayList(:)
        integer, intent(in) :: haloSeqIndexList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
        integer, intent(in), optional :: distgridToArrayMap(:)
        integer, intent(in), optional :: undistLBound(:)
```

```
        integer, intent(in), optional :: undistUBound(:)
        character (len=*), intent(in), optional :: name
        integer, intent(out), optional :: rc
```

RETURN VALUE:

```
    type(ESMF_Array) :: ESMF_ArrayCreateLocalArrayArb
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object from existing `ESMF_LocalArray` objects according to distgrid. Each PET must issue this call in unison in order to create a consistent Array object. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for array dimensions that are not distributed can be chosen freely using the `undistLBound` and `undistUBound` arguments.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

localarrayList List of valid `ESMF_LocalArray` objects, i.e. memory must be associated with the actual arguments. The type/kind/rank information of all `localarrayList` elements must be identical and will be used to set `Array`'s properties accordingly. The shape of each `localarrayList` element will be checked against the information contained in the `distgrid`.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of `haloSeqIndexList` can (and typically will) be different on each PET.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by `farray` directly or will copy the data from `farray` into a new memory allocation. Valid options are `ESMF_DATACOPY_REFERENCE` (default) or `ESMF_DATACOPY_VALUE`. Depending on the specific situation the `ESMF_DATACOPY_REFERENCE` option may be unsafe when specifying an array slice for `farray`.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the `localarrayList` elements by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the `localarrayList` elements in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the `localarrayList` elements are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `localarrayList`.

[name] Name of the Array object.

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

24.5.9 ESMF_ArrayCreate - Create Array object from typekind (allocate memory)

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAllocate(distgrid, typekind, indexflag, &
  distgridToArrayMap, computationalEdgeLWidth, computationalEdgeUWidth, &
  computationalLWidth, computationalUWidth, totalLWidth, totalUWidth, &
  undistLBound, undistUBound, name, rc)
```

ARGUMENTS:

```
       type(ESMF_DistGrid), intent(in) :: distgrid
       type(ESMF_TypeKind_Flag), intent(in) :: typekind
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_Index_Flag), intent(in), optional :: indexflag
       integer, intent(in), optional :: distgridToArrayMap(:)
       integer, intent(in), optional :: computationalEdgeLWidth(:)
       integer, intent(in), optional :: computationalEdgeUWidth(:)
       integer, intent(in), optional :: computationalLWidth(:)
       integer, intent(in), optional :: computationalUWidth(:)
       integer, intent(in), optional :: totalLWidth(:)
       integer, intent(in), optional :: totalUWidth(:)
       integer, intent(in), optional :: undistLBound(:)
       integer, intent(in), optional :: undistUBound(:)
       character (len=*), intent(in), optional :: name
       integer, intent(out), optional :: rc
```

RETURN VALUE:

```
    type(ESMF_Array) :: ESMF_ArrayCreateAllocate
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to typekind and distgrid. The Array rank is indirectly determined by the incoming information. Each PET must issue this call in unison in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the `distgrid` and the optional `Width` arguments.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

typekind The typekind of the Array.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section 9.24 for a list of valid indexflag options.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`.

Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the computational region. The default is to accommodate the union of exclusive and computational region.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

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

24.5.10 ESMF_ArrayCreate - Create Array object from typekind (allocate memory) w/ arbitrary seqIndices for halo

INTERFACE:

```
  ! Private name; call using ESMF_ArrayCreate()
  function ESMF_ArrayCreateAllocateArb(distgrid, typekind, &
    haloSeqIndexList, distgridToArrayMap, &
    undistLBound, undistUBound, name, rc)
```

ARGUMENTS:

```
        type(ESMF_DistGrid), intent(in) :: distgrid
        type(ESMF_TypeKind_Flag), intent(in) :: typekind
        integer, intent(in) :: haloSeqIndexList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer, intent(in), optional :: distgridToArrayMap(:)
        integer, intent(in), optional :: undistLBound(:)
```

```
        integer, intent(in), optional :: undistUBound(:)
        character (len=*), intent(in), optional :: name
        integer, intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_Array) :: ESMF_ArrayCreateAllocateArb
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to typekind and distgrid. The Array rank is indirectly determined by the incoming information. Each PET must issue this call in unison in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the `distgrid` and `haloSeqIndexList` arguments.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

typekind The typekind of the Array.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of `haloSeqIndexList` can (and typically will) be different on each PET.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

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

24.5.11 ESMF_ArrayCreate - Create Array object from ArraySpec (allocate memory)

INTERFACE:

```
   ! Private name; call using ESMF_ArrayCreate()
   function ESMF_ArrayCreateAllocateAS(distgrid, arrayspec, indexflag, &
     distgridToArrayMap, computationalEdgeLWidth, computationalEdgeUWidth, &
     computationalLWidth, computationalUWidth, totalLWidth, totalUWidth, &
     undistLBound, undistUBound, name, rc)
```

ARGUMENTS:

```
        type(ESMF_DistGrid), intent(in) :: distgrid
        type(ESMF_ArraySpec), intent(in) :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_Index_Flag), intent(in), optional :: indexflag
        integer, intent(in), optional :: distgridToArrayMap(:)
        integer, intent(in), optional :: computationalEdgeLWidth(:)
        integer, intent(in), optional :: computationalEdgeUWidth(:)
        integer, intent(in), optional :: computationalLWidth(:)
        integer, intent(in), optional :: computationalUWidth(:)
        integer, intent(in), optional :: totalLWidth(:)
        integer, intent(in), optional :: totalUWidth(:)
        integer, intent(in), optional :: undistLBound(:)
        integer, intent(in), optional :: undistUBound(:)
        character (len=*), intent(in), optional :: name
        integer, intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_Array) :: ESMF_ArrayCreateAllocateAS
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to arrayspec and distgrid. Each PET must issue this call with identical arguments in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the arguments to this call: `distgrid` and the optional `Width` arguments. The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

arrayspec `ESMF_ArraySpec` object containing the type/kind/rank information.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section 9.24 for a list of valid indexflag options.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the computational region. The default is to accommodate the union of exclusive and computational region.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the computational region. The default is to accommodate the union of exclusive and computational region.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

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

24.5.12 ESMF_ArrayCreate - Create Array object from ArraySpec (allocate memory) w/ arbitrary seqIndices for halo

INTERFACE:

```
   ! Private name; call using ESMF_ArrayCreate()
   function ESMF_ArrayCreateAllocateASArb(distgrid, arrayspec, &
     haloSeqIndexList, distgridToArrayMap, &
     undistLBound, undistUBound, name, rc)
```

ARGUMENTS:

```
        type(ESMF_DistGrid), intent(in) :: distgrid
        type(ESMF_ArraySpec), intent(in) :: arrayspec
        integer, intent(in) :: haloSeqIndexList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer, intent(in), optional :: distgridToArrayMap(:)
        integer, intent(in), optional :: undistLBound(:)
        integer, intent(in), optional :: undistUBound(:)
        character (len=*), intent(in), optional :: name
        integer, intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_Array) :: ESMF_ArrayCreateAllocateASArb
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to arrayspec and distgrid. Each PET must issue this call in unison in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the arguments to this call: `distgrid` and `haloSeqIndexList` arguments.
The return value is the newly created `ESMF_Array` object.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

arrayspec `ESMF_ArraySpec` object containing the type/kind/rank information.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of `haloSeqIndexList` can (and typically will) be different on each PET.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

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

24.5.13 ESMF_ArrayCreate - Create Array object as copy of existing Array object

INTERFACE:

```
   ! Private name; call using ESMF_ArrayCreate()
   function ESMF_ArrayCreateCopy(array, rc)
```

ARGUMENTS:

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

RETURN VALUE:

```
      type(ESMF_Array) :: ESMF_ArrayCreateCopy
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_Array` object as the copy of an existing Array.
The return value is the newly created `ESMF_Array` object.
The arguments are:

array `ESMF_Array` object to be copied.

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

24.5.14 ESMF_ArrayDestroy - Release resources associated with an Array object

INTERFACE:

```
   subroutine ESMF_ArrayDestroy(array, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroy an `ESMF_Array`, releasing the resources associated with the object.
The arguments are:

array `ESMF_Array` object to be destroyed.

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

24.5.15 ESMF_ArrayGather - Gather a Fortran array from an ESMF_Array

INTERFACE:

```
   subroutine ESMF_ArrayGather(array, farray, rootPet, tile, vm, rc)
```

ARGUMENTS:

```
   type(ESMF_Array), intent(in) :: array
   <type>(ESMF_KIND_<kind>), intent(out), target :: farray(<rank>)
   integer, intent(in) :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer, intent(in), optional :: tile
   type(ESMF_VM), intent(in), optional :: vm
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gather the data of an ESMF_Array object into the `farray` located on `rootPET`. A single DistGrid tile of `array` must be gathered into `farray`. The optional `tile` argument allows selection of the tile. For Arrays defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Array.
If the Array contains replicating DistGrid dimensions data will be gathered from the numerically higher DEs. Replicated data elements in numericaly lower DEs will be ignored.
This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.
The arguments are:

array The `ESMF_Array` object from which data will be gathered.

{farray} The Fortran array into which to gather data. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid destination array, i.e. `farray`.

[tile] The DistGrid tile in `array` from which to gather `farray`. By default `farray` will be gathered from tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

24.5.16 ESMF_ArrayGet - Access to Array internals

INTERFACE:

```
! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetDefault(array, arrayspec, typekind, &
  rank, localarrayList, indexflag, distgridToArrayMap, &
  distgridToPackedArrayMap, arrayToDistGridMap, undistLBound, &
  undistUBound, exclusiveLBound, exclusiveUBound, computationalLBound, &
  computationalUBound, totalLBound, totalUBound, computationalLWidth, &
  computationalUWidth, totalLWidth, totalUWidth, distgrid, dimCount, &
  tileCount, minIndexPTile, maxIndexPTile, deToTileMap, indexCountPDe, &
  delayout, deCount, localDeCount, localDeList, name, rc)
```

ARGUMENTS:

```
    type(ESMF_Array), intent(in) :: array
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_ArraySpec), intent(out), optional :: arrayspec
    type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
    integer, intent(out), optional :: rank
    type(ESMF_LocalArray), target, intent(out), optional :: localarrayList(:)
    type(ESMF_Index_Flag), intent(out), optional :: indexflag
    integer, target, intent(out), optional :: distgridToArrayMap(:)
    integer, target, intent(out), optional :: distgridToPackedArrayMap(:)
    integer, target, intent(out), optional :: arrayToDistGridMap(:)
    integer, target, intent(out), optional :: undistLBound(:)
    integer, target, intent(out), optional :: undistUBound(:)
```

```
integer, target, intent(out), optional :: exclusiveLBound(:,:)
integer, target, intent(out), optional :: exclusiveUBound(:,:)
integer, target, intent(out), optional :: computationalLBound(:,:)
integer, target, intent(out), optional :: computationalUBound(:,:)
integer, target, intent(out), optional :: totalLBound(:,:)
integer, target, intent(out), optional :: totalUBound(:,:)
integer, target, intent(out), optional :: computationalLWidth(:,:)
integer, target, intent(out), optional :: computationalUWidth(:,:)
integer, target, intent(out), optional :: totalLWidth(:,:)
integer, target, intent(out), optional :: totalUWidth(:,:)
type(ESMF_DistGrid), intent(out), optional :: distgrid
integer, intent(out), optional :: dimCount
integer, intent(out), optional :: tileCount
integer, intent(out), optional :: minIndexPTile(:,:)
integer, intent(out), optional :: maxIndexPTile(:,:)
integer, intent(out), optional :: deToTileMap(:)
integer, intent(out), optional :: indexCountPDe(:,:)
type(ESMF_DELayout), intent(out), optional :: delayout
integer, intent(out), optional :: deCount
integer, intent(out), optional :: localDeCount
integer, intent(out), optional :: localDeList(:)
character(len=*), intent(out), optional :: name
integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal information.
This interface works for any number of DEs per PET.
The arguments are:

array Queried `ESMF_Array` object.

[arrayspec] `ESMF_ArraySpec` object containing the type/kind/rank information of the Array object.

[typekind] TypeKind of the Array object.

[rank] Rank of the Array object.

[localarrayList] Upon return this holds a list of the associated `ESMC_LocalArray` objects. `localarrayList` must be allocated to be of size `localDeCount`, i.e. the number of DEs associated with the calling PET.

[indexflag] Upon return this flag indicates how the DE-local indices are defined. See section 9.24 for a list of possible return values.

[distgridToArrayMap] Upon return this list holds the Array dimensions against which the DistGrid dimensions are mapped. `distgridToArrayMap` must be allocated to be of size `dimCount`. An entry of zero indicates that the respective DistGrid dimension is replicating the Array across the DEs along this direction.

[distgridToPackedArrayMap] Upon return this list holds the indices of the Array dimensions in packed format against which the DistGrid dimensions are mapped. `distgridToPackedArrayMap` must be allocated to be of size `dimCount`. An entry of zero indicates that the respective DistGrid dimension is replicating the Array across the DEs along this direction.

[arrayToDistGridMap] Upon return this list holds the DistGrid dimensions against which the Array dimensions are mapped. `arrayToDistGridMap` must be allocated to be of size `rank`. An entry of zero indicates that the respective Array dimension is not decomposed, rendering it a tensor dimension.

[undistLBound] Upon return this array holds the lower bounds of the undistributed dimensions of the Array. UndistLBound must be allocated to be of size rank-dimCount.

[undistUBound] Upon return this array holds the upper bounds of the undistributed dimensions of the Array. UndistUBound must be allocated to be of size rank-dimCount.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive regions for all PET-local DEs. exclusiveLBound must be allocated to be of size (dimCount, localDeCount).

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive regions for all PET-local DEs. exclusiveUBound must be allocated to be of size (dimCount, localDeCount).

[computationalLBound] Upon return this holds the lower bounds of the computational regions for all PET-local DEs. computationalLBound must be allocated to be of size (dimCount, localDeCount).

[computationalUBound] Upon return this holds the upper bounds of the computational regions for all PET-local DEs. computationalUBound must be allocated to be of size (dimCount, localDeCount).

[totalLBound] Upon return this holds the lower bounds of the total regions for all PET-local DEs. totalLBound must be allocated to be of size (dimCount, localDeCount).

[totalUBound] Upon return this holds the upper bounds of the total regions for all PET-local DEs. totalUBound must be allocated to be of size (dimCount, localDeCount).

[computationalLWidth] Upon return this holds the lower width of the computational regions for all PET-local DEs. computationalLWidth must be allocated to be of size (dimCount, localDeCount).

[computationalUWidth] Upon return this holds the upper width of the computational regions for all PET-local DEs. computationalUWidth must be allocated to be of size (dimCount, localDeCount).

[totalLWidth] Upon return this holds the lower width of the total memory regions for all PET-local DEs. totalLWidth must be allocated to be of size (dimCount, localDeCount).

[totalUWidth] Upon return this holds the upper width of the total memory regions for all PET-local DEs. totalUWidth must be allocated to be of size (dimCount, localDeCount).

[distgrid] Upon return this holds the associated ESMF_DistGrid object.

[dimCount] Number of dimensions (rank) of distgrid.

[tileCount] Number of tiles in distgrid.

[minIndexPTile] Lower index space corner per dim, per tile, with size(minIndexPTile) == (/dimCount, tileCount/).

[maxIndexPTile] Upper index space corner per dim, per tile, with size(maxIndexPTile) == (/dimCount, tileCount/).

[deToTileMap] List of tile id numbers, one for each DE, with size(deToTileMap) == (/deCount/)

[indexCountPDe] Array of extents per dim, per de, with size(indexCountPDe) == (/dimCount, deCount/).

[delayout] Upon return this holds the associated ESMF_DELayout object.

[deCount] Upon return this holds the total number of DEs defined in the DELayout associated with the Array object.

[localDeCount] Upon return this holds the number of PET-local DEs defined in the DELayout associated with the Array object.

[localDeList] Upon return this holds the list of DE ids for the PET-local DEs defined in the DELayout associated with the Array object. The provided argument must be of size localDeCount.

[name] Name of the Array object.

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

24.5.17 ESMF_ArrayGet - Access to Array internals per dim per local DE

INTERFACE:

```
   ! Private name; call using ESMF_ArrayGet()
   subroutine ESMF_ArrayGetPLocalDePDim(array, dim, localDe, &
     indexCount, indexList, rc)
```

ARGUMENTS:

```
     type(ESMF_Array), intent(in) :: array
     integer, intent(in) :: dim
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(in), optional :: localDe
     integer, intent(out), optional :: indexCount
     integer, intent(out), optional :: indexList(:)
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal information per local DE, per dim.
This interface works for any number of DEs per PET.
The arguments are:

array Queried `ESMF_Array` object.

dim Dimension for which information is requested. `[1,..,dimCount]`

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[indexCount] DistGrid indexCount associated with `localDe, dim`.

[indexList] List of DistGrid tile-local indices for `localDe` along dimension `dim`.

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

24.5.18 ESMF_ArrayGet - Access to PET-local Array tile via Fortran array pointer

INTERFACE:

```
   ! Private name; call using ESMF_ArrayGet()
   subroutine ESMF_ArrayGetFPtr<rank><type><kind>(array, localDe, &
   farrayPtr, rc)
```

ARGUMENTS:

```
   type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer, intent(in), optional :: localDe
   <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Access Fortran array pointer to the specified DE-local memory allocation of the Array object.
The arguments are:

array Queried `ESMF_Array` object.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

farrayPtr Upon return, `farrayPtr` points to the DE-local data allocation of `localDe` in `array`. It depends on the specific entry point of `ESMF_ArrayCreate()` used during `array` creation, which Fortran operations are supported on the returned `farrayPtr`. See 24.3 for more details.

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

24.5.19 ESMF_ArrayGet - Access to PET-local Array tile via LocalArray object.

INTERFACE:

```
   ! Private name; call using ESMF_ArrayGet()
   subroutine ESMF_ArrayGetLocalArray(array, localDe, localarray, rc)
```

ARGUMENTS:

```
     type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(in), optional :: localDe
     type(ESMF_LocalArray), intent(inout) :: localarray
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Provide access to `ESMF_LocalArray` object that holds data for the specified local DE.
The arguments are:

array Queried `ESMF_Array` object.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

localarray Upon return `localarray` refers to the DE-local data allocation of `array`.

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

24.5.20 ESMF_ArrayHalo - Execute an Array halo operation

INTERFACE:

```
   subroutine ESMF_ArrayHalo(array, routehandle, &
     routesyncflag, finishedflag, cancelledflag, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),          intent(inout)         :: array
     type(ESMF_RouteHandle),    intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_RouteSync_Flag), intent(in),  optional :: routesyncflag
     logical,                   intent(out), optional :: finishedflag
     logical,                   intent(out), optional :: cancelledflag
     logical,                   intent(in),  optional :: checkflag
     integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed Array halo operation for `array`. The `array` argument must be weakly congruent and typekind conform to the Array used during `ESMF_ArrayHaloStore()`. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
See `ESMF_ArrayHaloStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.

array `ESMF_Array` containing data to be haloed.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section 9.38 for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[cancelledflag] A value of `.true.` indicates that were cancelled communication operations. In this case the data in the `dstArray` must be considered invalid. It may have been partially modified by the call. A value of `.false.` indicates that none of the communication operations was cancelled. The data in `dstArray` is valid if `finishedflag` returns equal `.true.`.

[checkflag] If set to `.TRUE.` the input Array pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

24.5.21 ESMF_ArrayHaloRelease - Release resources associated with Array halo operation

INTERFACE:

```
   subroutine ESMF_ArrayHaloRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with an Array halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

24.5.22 ESMF_ArrayHaloStore - Precompute an Array halo operation

INTERFACE:

```
     subroutine ESMF_ArrayHaloStore(array, routehandle, &
       startregion, haloLDepth, haloUDepth, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),            intent(inout)         :: array
     type(ESMF_RouteHandle),      intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_StartRegion_Flag), intent(in),  optional ::startregion
     integer,                     intent(in),  optional :: haloLDepth(:)
     integer,                     intent(in),  optional :: haloUDepth(:)
     integer,                     intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Store an Array halo operation over the data in `array`. By default, i.e. without specifying `startregion`, `haloLDepth` and `haloUDepth`, all elements in the total Array region that lie outside the exclusive region will be considered potential destination elements for halo. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.
Specifying `startregion` allows to change the shape of the effective halo region from the inside. Setting this flag to `ESMF_STARTREGION_COMPUTATIONAL` means that only elements outside the computational region of the Array are considered for potential destination elements for halo. The default is `ESMF_STARTREGION_EXCLUSIVE`.
The `haloLDepth` and `haloUDepth` arguments allow to reduce the extent of the effective halo region. Starting at the region specified by `startregion`, the `haloLDepth` and `haloUDepth` define a halo depth in each direction. Note that the maximum halo region is limited by the total Array region, independent of the actual `haloLDepth`

and haloUDepth setting. The total Array region is local DE specific. The haloLDepth and haloUDepth are interpreted as the maximum desired extent, reducing the potentially larger region available for halo.
The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayHalo() on any Array that is weakly congruent and typekind conform to array. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same routehandle can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
This call is *collective* across the current VM.

array ESMF_Array containing data to be haloed. The data in the halo region may be destroyed by this call.

routehandle Handle to the precomputed Route.

[startregion] The start of the effective halo region on every DE. The default setting is ESMF_STARTREGION_EXCLUSIVE, rendering all non-exclusive elements potential halo destination elments. See section 9.41 for a complete list of valid settings.

[haloLDepth] This vector specifies the lower corner of the effective halo region with respect to the lower corner of startregion. The size of haloLDepth must equal the number of distributed Array dimensions.

[haloUDepth] This vector specifies the upper corner of the effective halo region with respect to the upper corner of startregion. The size of haloUDepth must equal the number of distributed Array dimensions.

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

24.5.23 ESMF_ArrayPrint - Print Array internals

INTERFACE:

```
   subroutine ESMF_ArrayPrint(array, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Print internal information of the specified ESMF_Array object.

The arguments are:

array ESMF_Array object.

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

24.5.24 ESMF_ArrayRead - Read Array data from a file

INTERFACE:

```
   subroutine ESMF_ArrayRead(array, file, variableName, &
     timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),     intent(inout)         :: array
     character(*),         intent(in)            :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),         intent(in),  optional :: variableName
     integer,              intent(in),  optional :: timeslice
     type(ESMF_IOFmtFlag), intent(in),  optional :: iofmt
     integer,              intent(out), optional :: rc
```

DESCRIPTION:

Read Array data from file and put it into an ESMF_Array object. For this API to be functional, the environment variable ESMF_PIO should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in ESMF_COMM=mpiuni mode.

The arguments are:

array The ESMF_Array object in which the read data is returned.

file The name of the file from which Array data is read.

[variableName] Variable name in the file; default is the "name" of Array. Use this argument only in the IO format (such as NetCDF) that supports variable name. If the IO format does not support this (such as binary format), ESMF will return an error code.

[timeslice] The time-slice number of the variable read from file.

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to ESMF_IOFMT_NETCDF.

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

24.5.25 ESMF_ArrayRedist - Execute an Array redistribution

INTERFACE:

```
   subroutine ESMF_ArrayRedist(srcArray, dstArray, routehandle, &
     routesyncflag, finishedflag, cancelledflag, checkflag, rc)
```

ARGUMENTS:

```
    type(ESMF_Array),          intent(in),    optional :: srcArray
    type(ESMF_Array),          intent(inout), optional :: dstArray
    type(ESMF_RouteHandle),    intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_RouteSync_Flag), intent(in),    optional  :: routesyncflag
    logical,                   intent(out),   optional :: finishedflag
    logical,                   intent(out),   optional :: cancelledflag
    logical,                   intent(in),    optional :: checkflag
    integer,                   intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed Array redistribution from `srcArray` to `dstArray`. Both `srcArray` and `dstArray` must be weakly congruent and typekind conform with the respective Arrays used during `ESMF_ArrayRedistStore()`. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.
See `ESMF_ArrayRedistStore()` on how to precompute `routehandle`.
This call is *collective* across the current VM.

[srcArray] `ESMF_Array` with source data.

[dstArray] `ESMF_Array` with destination data.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section 9.38 for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[cancelledflag] A value of `.true.` indicates that were cancelled communication operations. In this case the data in the `dstArray` must be considered invalid. It may have been partially modified by the call. A value of `.false.` indicates that none of the communication operations was cancelled. The data in `dstArray` is valid if `finishedflag` returns equal `.true.`.

[checkflag] If set to `.TRUE.` the input Array pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

24.5.26 ESMF_ArrayRedistRelease - Release resources associated with Array redistribution

INTERFACE:

```
subroutine ESMF_ArrayRedistRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with an Array redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

24.5.27 ESMF_ArrayRedistStore - Precompute Array redistribution with local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayRedistStore()
subroutine ESMF_ArrayRedistStore<type><kind>(srcArray, dstArray, &
  routehandle, factor, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),          intent(in)             :: srcArray
     type(ESMF_Array),          intent(inout)          :: dstArray
     type(ESMF_RouteHandle),    intent(inout)          :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)             :: factor
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                   intent(in),  optional :: srcToDstTransposeMap(:)
     integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

`ESMF_ArrayRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArrayRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArrayRedistStore()` method, as provided through the separate entry points shown in 24.5.27 and 24.5.28, is described in the following paragraphs as a whole.
Store an Array redistribution operation from `srcArray` to `dstArray`. Interface 24.5.27 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 24.5.28. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.
Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*.
Source Array, destination Array, and the factor may be of different <type><kind>. Further, source and destination Arrays may differ in shape, however, the number of elements must match.

If srcToDstTransposeMap is not specified the redistribution corresponds to an identity mapping of the sequentialized source Array to the sequentialized destination Array. If the srcToDstTransposeMap argument is provided it must be identical on all PETs. The srcToDstTransposeMap allows source and destination Array dimensions to be transposed during the redistribution. The number of source and destination Array dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Array object for srcArray and dstArray arguments.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayRedist() on any pair of Arrays that are weakly congruent and typekind conform with the srcArray, dstArray pair. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same routehandle can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multipy source data.

[srcToDstTransposeMap] List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

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

24.5.28 ESMF_ArrayRedistStore - Precompute Array redistribution without local factor argument

INTERFACE:

```
   ! Private name; call using ESMF_ArrayRedistStore()
   subroutine ESMF_ArrayRedistStoreNF(srcArray, dstArray, routehandle, &
     srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),       intent(in)            :: srcArray
     type(ESMF_Array),       intent(inout)         :: dstArray
     type(ESMF_RouteHandle), intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),  optional :: srcToDstTransposeMap(:)
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

ESMF_ArrayRedistStore() is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into ESMF_ArrayRedistStore() through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the ESMF_ArrayRedistStore() method, as provided through the separate entry points shown in 24.5.27 and 24.5.28, is described in the following paragraphs as a whole.

Store an Array redistribution operation from srcArray to dstArray. Interface 24.5.27 allows PETs to specify a factor argument. PETs not specifying a factor argument call into interface 24.5.28. If multiple PETs specify the factor argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a factor argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*.

Source Array, destination Array, and the factor may be of different <type><kind>. Further, source and destination Arrays may differ in shape, however, the number of elements must match.

If srcToDstTransposeMap is not specified the redistribution corresponds to an identity mapping of the sequentialized source Array to the sequentialized destination Array. If the srcToDstTransposeMap argument is provided it must be identical on all PETs. The srcToDstTransposeMap allows source and destination Array dimensions to be transposed during the redistribution. The number of source and destination Array dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Array object for srcArray and dstArray arguments.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayRedist() on any pair of Arrays that are weakly congruent and typekind conform with the srcArray, dstArray pair. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the sizes of the undistributed dimensions, that vary faster with memory than the first distributed dimension, are permitted to be different. This means that the same routehandle can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap] List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

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

24.5.29 ESMF_ArrayScatter - Scatter a Fortran array across the ESMF_Array

INTERFACE:

```
   subroutine ESMF_ArrayScatter(array, farray, rootPet, tile, vm, rc)
```

ARGUMENTS:

```
   type(ESMF_Array), intent(inout) :: array
   <type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
   integer, intent(in) :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer, intent(in), optional :: tile
   type(ESMF_VM), intent(in), optional :: vm
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Scatter the data of farray located on rootPET across an ESMF_Array object. A single farray must be scattered across a single DistGrid tile in Array. The optional tile argument allows selection of the tile. For Arrays defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of farray must match the shape of the tile in Array.
If the Array contains replicating DistGrid dimensions data will be scattered across all of the replicated pieces.
This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.
The arguments are:

array The ESMF_Array object across which data will be scattered.

{farray} The Fortran array that is to be scattered. Only root must provide a valid farray, the other PETs may treat farray as an optional argument.

rootPet PET that holds the valid data in farray.

[tile] The DistGrid tile in array into which to scatter farray. By default farray will be scattered into tile 1.

[vm] Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method's overhead.

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

24.5.30 ESMF_ArraySet - Set Array properties

INTERFACE:

```
   ! Private name; call using ESMF_ArraySet()
   subroutine ESMF_ArraySetDefault(array, computationalLWidth, &
     computationalUWidth, name, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),   intent(inout)         :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(in),  optional :: computationalLWidth(:,:)
     integer,            intent(in),  optional :: computationalUWidth(:,:)
     character(len = *), intent(in),  optional :: name
     integer,            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets adjustable settings in an ESMF_Array object. Arrays with tensor dimensions will set values for *all* tensor components.
The arguments are:

array ESMF_Array object for which to set properties.

[name] The Array name.

[computationalLWidth] This argument must have of size (dimCount, localDeCount). computationalLWidth specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for all local DEs.

[computationalUWidth] This argument must have of size (dimCount, localDeCount). computationalUWidth specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for all local DEs.

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

24.5.31 ESMF_ArraySet - Set Array properties

INTERFACE:

```
   ! Private name; call using ESMF_ArraySet()
   subroutine ESMF_ArraySetPLocalDe(array, localDe, rimSeqIndex, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),   intent(inout)         :: array
     integer,            intent(in)            :: localDe
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(in),  optional :: rimSeqIndex(:)
     integer,            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets adjustable settings in an ESMF_Array object for a specific localDe.
The arguments are:

array ESMF_Array object for which to set properties.

localDe Local DE for which to set values.

[rimSeqIndex] Sequence indices in the halo rim of localDe.

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

24.5.32 ESMF_ArraySMM - Execute an Array sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_ArraySMM(srcArray, dstArray, routehandle, &
     routesyncflag, finishedflag, cancelledflag, zeroregion, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),         intent(in),    optional :: srcArray
     type(ESMF_Array),         intent(inout), optional :: dstArray
     type(ESMF_RouteHandle),   intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
    type(ESMF_RouteSync_Flag), intent(in),    optional :: routesyncflag
    logical,                   intent(out),   optional :: finishedflag
    logical,                   intent(out),   optional :: cancelledflag
    type(ESMF_Region_Flag),    intent(in),    optional :: zeroregion
    logical,                   intent(in),    optional :: checkflag
    integer,                   intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Execute a precomputed Array sparse matrix multiplication from `srcArray` to `dstArray`. Both `srcArray` and `dstArray` must be weakly congruent and typekind conform to the respective Arrays used during `ESMF_ArraySMMStore()`.
Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the size of the undistributed dimensions, that vary faster with memory than the first distributed dimension, is permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.
See `ESMF_ArraySMMStore()` on how to precompute `routehandle`. See section 24.2.17 for details on the operation `ESMF_ArraySMM()` performs.
This call is *collective* across the current VM.

[srcArray] `ESMF_Array` with source data.

[dstArray] `ESMF_Array` with destination data.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section 9.38 for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[cancelledflag] A value of `.true.` indicates that were cancelled communication operations. In this case the data in the `dstArray` must be considered invalid. It may have been partially modified by the call. A value of `.false.` indicates that none of the communication operations was cancelled. The data in `dstArray` is valid if `finishedflag` returns equal `.true.`.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in `dstArray` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstArray` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section 9.36 for a complete list of valid settings.

[checkflag] If set to `.TRUE.` the input Array pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

24.5.33 ESMF_ArraySMMRelease - Release resources associated with Array sparse matrix multiplication

INTERFACE:

```
subroutine ESMF_ArraySMMRelease(routehandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Release resouces associated with an Array sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

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

24.5.34 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStore<type><kind>(srcArray, dstArray, &
  routehandle, factorList, factorIndexList, rc)
```

ARGUMENTS:

```
    type(ESMF_Array),                  intent(in)    :: srcArray
    type(ESMF_Array),                  intent(inout) :: dstArray
    type(ESMF_RouteHandle),            intent(inout) :: routehandle
    <type>(ESMF_KIND_<kind>), target, intent(in)    :: factorList(:)
    integer,                           intent(in)    :: factorIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

`ESMF_ArraySMMStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArraySMMStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArraySMMStore()` method, as provided through the separate entry points shown in 24.5.34 and 24.5.35, is described in the following paragraphs as a whole.
Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a

PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* factorList and factorIndexList arguments.
Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*.
Source and destination Arrays, as well as the supplied factorList argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements.
It is erroneous to specify the identical Array object for srcArray and dstArray arguments.
The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArraySMM() on any pair of Arrays that are weakly congruent and typekind conform with the srcArray, dstArray pair. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the size of the undistributed dimensions, that vary faster with memory than the first distributed dimension, is permitted to be different. This means that the same routehandle can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.
This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in factorList.

The second dimension of factorIndexList steps through the list of pairs, i.e. size(factorIndexList,2) == size(factorList). The first dimension of factorIndexList is either of size 2 or size 4.

In the *size 2 format* factorIndexList(1,:) specifies the sequence index of the source element in the srcArray while factorIndexList(2,:) specifies the sequence index of the destination element in dstArray. For this format to be a valid option source and destination Arrays must have matching number of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identiy matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the factorIndexList(1,:) specifies the sequence index while factorIndexList(2,:) specifies the tensor sequence index of the source element in the srcArray. Further factorIndexList(3,:) specifies the sequence index and factorIndexList(4,:) specifies the tensor sequence index of the destination element in the dstArray.

See section 24.2.17 for details on the definition of Array *sequence indices* and *tensor sequence indices*.

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

24.5.35 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication without local factors

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStoreNF(srcArray, dstArray, routehandle, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),       intent(in)             :: srcArray
     type(ESMF_Array),       intent(inout)          :: dstArray
     type(ESMF_RouteHandle), intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

`ESMF_ArraySMMStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArraySMMStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArraySMMStore()` method, as provided through the separate entry points shown in 24.5.34 and 24.5.35, is described in the following paragraphs as a whole.
Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.
Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 24.2.17 for details on the definition of *sequence indices*.
Source and destination Arrays, as well as the supplied `factorList` argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements.
It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.
The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArraySMM()` on any pair of Arrays that are weakly congruent and typekind conform with the `srcArray`, `dstArray` pair. Congruent Arrays possess matching DistGrids, and the shape of the local array tiles matches between the Arrays for every DE. For weakly congruent Arrays the size of the undistributed dimensions, that vary faster with memory than the first distributed dimension, is permitted to be different. This means that the same `routehandle` can be applied to a large class of similar Arrays that differ in the number of elements in the left most undistributed dimensions.

This call is *collective* across the current VM.

srcArray `ESMF_Array` with source data.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

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

24.5.36 ESMF_ArrayValidate - Validate Array internals

INTERFACE:

```
   subroutine ESMF_ArrayValidate(array, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the `Array` is internally consistent. The method returns an error code if problems are found.
The arguments are:

array Specified `ESMF_Array` object.

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

24.5.37 ESMF_ArrayWrite - Write Array data into a file

INTERFACE:

```
  subroutine ESMF_ArrayWrite(array, file, &
     variableName, append, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),     intent(in)            :: array
     character(*),         intent(in)            :: file
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),         intent(in),  optional :: variableName
     logical,              intent(in),  optional :: append
     integer,              intent(in),  optional :: timeslice
     type(ESMF_IOFmtFlag), intent(in),  optional :: iofmt
     integer,              intent(out), optional :: rc
```

DESCRIPTION:

Write Array data into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, 32.3.
Limitations:

- Only 1 DE per PET supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

array The `ESMF_Array` object that contains data to be written.

file The name of the output file to which Array data is written.

[variableName] Variable name in the output file; default is the "name" of Array. Use this argument only in the IO format (such as NetCDF) that supports variable name. If the IO format does not support this (such as binary format), ESMF will return an error code.

[append] Logical: if .true., data (with attributes) is appended to an existing file; default is .false.

[timeslice] Some IO formats (e.g. NetCDF) support the output of data in form of time slices. The `timeslice` argument provides access to this capability. Usage of this feature requires that the first slice is written with a positive `timeslice` value, and that subsequent slices are written with a `timeslice` argument that increments by one each time. By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file.

[iofmt] The IO format. Please see Section 9.25 for the list of options. If not present, defaults to `ESMF_IOFMT_NETCDF`.

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

25 LocalArray Class

25.1 Description

The `ESMF_LocalArray` class provides a language independent representation of data in array format. One of the major functions of the LocalArray class is to bridge the Fortran/C/C++ language difference that exists with respect to array representation. All ESMF Field and Array data is internally stored in ESMF LocalArray objects allowing transparent access from Fortran and C/C++.
In the ESMF Fortran API the LocalArray becomes visible in those cases where a local PET may be associated with multiple pieces of an Array, e.g. if there are multiple DEs associated with a single PET. The Fortran language standard does not provide an array of arrays construct, however arrays of derived types holding arrays are possible. ESMF calls use arguments that are of type `ESMF_LocalArray` with `dimension` attributes where necessary.

25.2 Restrictions and Future Work

- The TKR (type/kind/rank) overloaded LocalArray interfaces declare the dummy Fortran array arguments with the pointer attribute. The advantage of doing this is that it allows ESMF to inquire information about the provided Fortran array. The disadvantage of this choice is that actual Fortran arrays passed into these interfaces *must* also be defined with pointer attribute in the user code.

25.3 Class API

25.3.1 ESMF_LocalArrayAssignment(=) - LocalArray assignment

INTERFACE:

```
interface assignment(=)
localarray1 = localarray2
```

ARGUMENTS:

```
type(ESMF_LocalArray) :: localarray1
type(ESMF_LocalArray) :: localarray2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign localarray1 as an alias to the same ESMF LocalArray object in memory as localarray2. If localarray2 is invalid, then localarray1 will be equally invalid after the assignment.
The arguments are:

localarray1 The `ESMF_LocalArray` object on the left hand side of the assignment.

localarray2 The `ESMF_LocalArray` object on the right hand side of the assignment.

25.3.2 ESMF_LocalArrayOperator(==) - LocalArray equality operator

INTERFACE:

```
interface operator(==)
if (localarray1 == localarray2) then ... endif
OR
result = (localarray1 == localarray2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_LocalArray), intent(in) :: localarray1
type(ESMF_LocalArray), intent(in) :: localarray2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether localarray1 and localarray2 are valid aliases to the same ESMF LocalArray object in memory. For a more general comparison of two ESMF LocalArrays, going beyond the simple alias test, the ESMF_LocalArrayMatch() function (not yet implemented) must be used.
The arguments are:

localarray1 The `ESMF_LocalArray` object on the left hand side of the equality operation.

localarray2 The `ESMF_LocalArray` object on the right hand side of the equality operation.

25.3.3 ESMF_LocalArrayOperator(/=) - LocalArray not equal operator

INTERFACE:

```
interface operator(/=)
if (localarray1 /= localarray2) then ... endif
OR
result = (localarray1 /= localarray2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_LocalArray), intent(in) :: localarray1
type(ESMF_LocalArray), intent(in) :: localarray2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether localarray1 and localarray2 are *not* valid aliases to the same ESMF LocalArray object in memory. For a more general comparison of two ESMF LocalArrays, going beyond the simple alias test, the ESMF_LocalArrayMatch() function (not yet implemented) must be used.
The arguments are:

localarray1 The `ESMF_LocalArray` object on the left hand side of the non-equality operation.

localarray2 The `ESMF_LocalArray` object on the right hand side of the non-equality operation.

25.3.4 ESMF_LocalArrayCreate – Create a LocalArray by explicitly specifying typekind and rank arguments

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateByTKR(typekind, rank, totalCount, &
  totalLBound, totalUBound, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrayCreateByTKR
```

ARGUMENTS:

```
     type(ESMF_TypeKind_Flag), intent(in) :: typekind
     integer, intent(in) :: rank
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(in), optional :: totalCount(:)
     integer, intent(in), optional :: totalLBound(:)
     integer, intent(in), optional :: totalUBound(:)
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create a new `ESMF_LocalArray` and allocate data space, which remains uninitialized. The return value is a new LocalArray.
The arguments are:

typekind Array typekind. See section 9.45 for valid values.

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

[totalCount] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The `count` argument may be omitted if both `totalLBound` and `totalUBound` arguments are present.

[totalLBound] An integer array of length rank, with the lower index for each dimension.

[totalUBound] An integer array of length rank, with the upper index for each dimension.

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

25.3.5 ESMF_LocalArrayCreate – Create a LocalArray by specifying an ArraySpec

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateBySpec(arrayspec, totalCount, &
  totalLBound, totalUBound, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrayCreateBySpec
```

ARGUMENTS:

```
     type(ESMF_ArraySpec), intent(in) :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(in), optional :: totalCount(:)
     integer, intent(in), optional :: totalLBound(:)
     integer, intent(in), optional :: totalUBound(:)
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create a new `ESMF_LocalArray` and allocate data space, which remains uninitialized. The return value is a new LocalArray.
The arguments are:

arrayspec ArraySpec object specifying typekind and rank.

[totalCount] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The `count` argument may be omitted if both `totalLBound` and `totalUBound` arguments are present.

[totalLBound] An integer array of length rank, with the lower index for each dimension.

[totalUBound] An integer array of length rank, with the upper index for each dimension.

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

25.3.6 ESMF_LocalArrayCreate – Create a LocalArray from pre-existing LocalArray

INTERFACE:

```
   ! Private name; call using ESMF_LocalArrayCreate()
   function ESMF_LocalArrayCreateCopy(localarray, rc)
```

RETURN VALUE:

```
     type(ESMF_LocalArray) :: ESMF_LocalArrayCreateCopy
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Perform a deep copy of an existing `ESMF_LocalArray` object. The return value is a new LocalArray.
The arguments are:

localarray Existing LocalArray to be copied.

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

25.3.7 ESMF_LocalArrayCreate - Create a LocalArray from a Fortran pointer (associated or unassociated)

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrCreateByPtr<rank><type><kind>(farrayPtr, &
datacopyflag, totalCount, totalLBound, totalUBound, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrCreateByPtr<rank><type><kind>
```

ARGUMENTS:

```
   <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   integer, intent(in), optional :: totalCount(:)
   integer, intent(in), optional :: totalLBound(:)
   integer, intent(in), optional :: totalUBound(:)
   integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates an `ESMF_LocalArray` based on a Fortran array pointer. Two cases must be distinguished.
First, if `farrayPtr` is associated the optional `datacopyflag` argument may be used to indicate whether the associated data is to be copied or referenced. For associated `farrayPtr` the optional `totalCount`, `totalLBound` and `totalUBound` arguments need not be specified. However, all present arguments will be checked against `farrayPtr` for consistency.
Second, if `farrayPtr` is unassociated the optional argument `datacopyflag` must not be specified. However, in this case a complete set of totalCount and bounds information must be provided. Any combination of present `totalCount` `totalLBound` and `totalUBound` arguments that provides a complete specification is valid. All input information will be checked for consistency.
The arguments are:

farrayPtr A Fortran array pointer (associated or unassociated).

[datacopyflag] Indicate copy vs. reference behavior in case of associated `farrayPtr`. This argument must *not* be present for unassociated `farrayPtr`. Default to `ESMF_DATACOPY_REFERENCE`, makes the `ESMF_LocalArray` reference the associated data array. If set to `ESMF_DATACOPY_VALUE` this routine allocates new memory and copies the data from the pointer into the new LocalArray allocation.

[totalCount] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The `count` argument may be omitted if both `totalLBound` and `totalUBound` arguments are present.

[totalLBound] An integer array of lower index values. Must be the same length as the rank.

[totalUBound] An integer array of upper index values. Must be the same length as the rank.

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

25.3.8 ESMF_LocalArrayDestroy - Release resources associated with a LocalArray

INTERFACE:

```
   subroutine ESMF_LocalArrayDestroy(localarray, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroys an `ESMF_LocalArray`, releasing all resources associated with the object.
The arguments are:

localarray Destroy contents of this `ESMF_LocalArray`.

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

25.3.9 ESMF_LocalArrayGet - Return LocalArray information

INTERFACE:

```
   ! Private name; call using ESMF_LocalArrayGet()
   subroutine ESMF_LocalArrayGetDefault(localarray, &
     typekind, rank, totalCount, totalLBound, totalUBound, rc)
```

ARGUMENTS:

```
     type(ESMF_LocalArray), intent(in) :: localarray
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
     integer, intent(out), optional :: rank
     integer, intent(out), optional :: totalCount(:)
     integer, intent(out), optional :: totalLBound(:)
     integer, intent(out), optional :: totalUBound(:)
     integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns information about the `ESMF_LocalArray`.
The arguments are:

localarray Queried `ESMF_LocalArray` object.

[typekind] TypeKind of the LocalArray object.

[rank] Rank of the LocalArray object.

[totalCount] Count per dimension.

[totalLBound] Lower bound per dimension.

[totalUBound] Upper bound per dimension.

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

25.3.10 ESMF_LocalArrayGet - Get access to data in a LocalArray object

INTERFACE:

```
! Private name; call using ESMF_LocalArrayGet()
subroutine ESMF_LocalArrayGetData<rank><type><kind>(localarray, farrayPtr, &
datacopyflag, rc)
```

ARGUMENTS:

```
  type(ESMF_LocalArray) :: localarray
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Return a Fortran pointer to the data buffer, or return a Fortran pointer to a new copy of the data.
The arguments are:

localarray The `ESMF_LocalArray` to get the value from.

farrayPtr An unassociated or associated Fortran pointer correctly allocated.

[datacopyflag] An optional copy flag which can be specified. Can either make a new copy of the data or reference existing data. See section 9.12 for a list of possible values.

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

---

26 ArraySpec Class

26.1 Description

An ArraySpec is a very simple class that contains type, kind, and rank information about an Array. This information is stored in two parameters. **TypeKind** describes the data type of the elements in the Array and their precision. **Rank** is the number of dimensions in the Array.
The only methods that are associated with the ArraySpec class are those that allow you to set and retrieve this information.

26.2 Use and Examples

The ArraySpec is passed in as an argument at Field and FieldBundle creation in order to describe an Array that will be allocated or attached at a later time. There are any number of situations in which this approach is useful. One common example is a case in which the user wants to create a very flexible export State with many diagnostic variables predefined, but only a subset desired and consequently allocated for a particular run.

```
! !PROGRAM: ESMF_ArraySpecEx - ArraySpec manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of ArraySpec set and get usage
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF
      implicit none

      ! local variables
      type(ESMF_ArraySpec) :: arrayDS
      integer :: myrank
      type(ESMF_TypeKind_Flag) :: mytypekind


      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultlogfilename="ArraySpecEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

26.2.1 Set ArraySpec values

This example shows how to set values in an `ESMF_ArraySpec`.

```
      call ESMF_ArraySpecSet(arrayDS, rank=2, &
                             typekind=ESMF_TYPEKIND_R8, rc=rc)
```

26.2.2 Get ArraySpec values

This example shows how to query an `ESMF_ArraySpec`.

```
      call ESMF_ArraySpecGet(arrayDS, rank=myrank, &
        typekind=mytypekind, rc=rc)
      print *, "Returned values from ArraySpec:"
      print *, "rank =", myrank


      ! finalize ESMF framework
      call ESMF_Finalize(rc=rc)
```

```
end program ESMF_ArraySpecEx
```

26.3 Restrictions and Future Work

1. **Limit on rank.** The values for type, kind and rank passed into the ArraySpec class are subject to the same limitations as Arrays. The maximum array rank is 7, which is the highest rank supported by Fortran.

26.4 Design and Implementation Notes

The information contained in an `ESMF_ArraySpec` is used to create `ESMF_Array` objects.
`ESMF_ArraySpec` is a shallow class, and only set and get methods are needed. They do not need to be created or destroyed.

26.5 Class API

26.5.1 ESMF_ArraySpecAssignment(=) - Assign an ArraySpec to another ArraySpec

INTERFACE:

```
interface assignment(=)
  arrayspec1 = arrayspec2
```

ARGUMENTS:

```
type(ESMF_ArraySpec) :: arrayspec1
type(ESMF_ArraySpec) :: arrayspec2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Set `arrayspec1` equal to `arrayspec2`. This is the default Fortran assignment, which creates a complete, independent copy of `arrayspec2` as `arrayspec1`. If `arrayspec2` is an invalid `ESMF_ArraySpec` object then `arrayspec1` will be equally invalid after the assignment.
The arguments are:

arrayspec1 The `ESMF_ArraySpec` to be set.

arrayspec2 The `ESMF_ArraySpec` to be copied.

26.5.2 ESMF_ArraySpecOperator(==) - Test if ArraySpec 1 is equal to ArraySpec 2

INTERFACE:

```
interface operator(==)
  if (arrayspec1 == arrayspec2) then ... endif
               OR
  result = (arrayspec1 == arrayspec2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec1
type(ESMF_ArraySpec), intent(in) :: arrayspec2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (==) operator for the `ESMF_ArraySpec` class to return `.true.` if `arrayspec1` and `arrayspec2` specify the same type, kind and rank, and `.false.` otherwise.
The arguments are:

arrayspec1 First `ESMF_ArraySpec` in comparison.

arrayspec2 Second `ESMF_ArraySpec` in comparison.

26.5.3 ESMF_ArraySpecOperator(/=) - Test if ArraySpec 1 is not equal to ArraySpec 2

INTERFACE:

```
interface operator(/=)
  if (arrayspec1 /= arrayspec2) then ... endif
               OR
  result = (arrayspec1 /= arrayspec2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec1
type(ESMF_ArraySpec), intent(in) :: arrayspec2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_ArraySpec` class to return `.true.` if `arrayspec1` and `arrayspec2` do not specify the same type, kind or rank, and `.false.` otherwise.
The arguments are:

arrayspec1 First `ESMF_ArraySpec` in comparison.

arrayspec2 Second `ESMF_ArraySpec` in comparison.

26.5.4 ESMF_ArraySpecGet - Get values from an ArraySpec

INTERFACE:

```
subroutine ESMF_ArraySpecGet(arrayspec, rank, typekind, rc)
```

ARGUMENTS:

```
     type(ESMF_ArraySpec),     intent(in)            :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                  intent(out), optional :: rank
     type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
     integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns information about the contents of an `ESMF_ArraySpec`.
The arguments are:

arrayspec The `ESMF_ArraySpec` to query.

[rank] Array rank (dimensionality – 1D, 2D, etc). Maximum possible is 7D.

[typekind] Array typekind. See section 9.45 for valid values.

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

26.5.5 ESMF_ArraySpecPrint - Print information of ArraySpec

INTERFACE:

```
   subroutine ESMF_ArraySpecPrint(arrayspec, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Print ArraySpec internals.

The arguments are:

arrayspec Specified `ESMF_ArraySpec` object.

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

26.5.6 ESMF_ArraySpecSet - Set values for an ArraySpec

INTERFACE:

```
   subroutine ESMF_ArraySpecSet(arrayspec, rank, typekind, rc)
```

ARGUMENTS:

```
     type(ESMF_ArraySpec),     intent(out)           :: arrayspec
     integer,                  intent(in)            :: rank
     type(ESMF_TypeKind_Flag), intent(in)            :: typekind
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates a description of the data – the typekind, the rank, and the dimensionality.
The arguments are:

arrayspec The `ESMF_ArraySpec` to set.

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

typekind Array typekind. See section 9.45 for valid values.

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

26.5.7 ESMF_ArraySpecValidate - Validate ArraySpec internals

INTERFACE:

```
   subroutine ESMF_ArraySpecValidate(arrayspec, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the `arrayspec` is internally consistent. The method returns an error code if problems are found.
The arguments are:

arrayspec Specified `ESMF_ArraySpec` object.

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

27 Grid Class

27.1 Description

The ESMF Grid class is used to describe the geometry and discretization of logically rectangular physical grids. It also contains the description of the grid's underlying topology and the decomposition of the physical grid across the available computational resources. The most frequent use of the Grid class is to describe physical grids in user code so that sufficient information is available to perform ESMF methods such as regridding.

Key Features
Representation of grids formed by logically rectangular regions, including uniform and rectilinear grids (e.g. lat-lon grids), curvilinear grids (e.g. displaced pole grids), and grids formed by connected logically rectangular regions (e.g. cubed sphere grids).
Support for 1D, 2D, 3D, and higher dimension grids.
Distribution of grids across computational resources for parallel operations - users set which grid dimensions are distributed.
Grids can be created already distributed, so that no single resource needs global information during the creation process.
Options to define periodicity and other edge connectivities either explicitly or implicitly via shape shortcuts.
Options for users to define grid coordinates themselves or call prefabricated coordinate generation routines for standard grids [NO GENERATION ROUTINES YET].
Options for incremental construction of grids.
Options for using a set of pre-defined stagger locations or for setting custom stagger locations.

27.1.1 Grid Representation in ESMF

ESMF Grids are based on the concepts described in *A Standard Description of Grids Used in Earth System Models* [Balaji 2006]. In this document Balaji introduces the mosaic concept as a means of describing a wide variety of Earth system model grids. A **mosaic** is composed of grid tiles connected at their edges. Mosaic grids includes simple, single tile grids as a special case.
The ESMF Grid class is a representation of a mosaic grid. Each ESMF Grid is constructed of one or more logically rectangular **Tiles**. A Tile will usually have some physical significance (e.g. the region of the world covered by one face of a cubed sphere grid).
The piece of a Tile that resides on one DE (for simple cases, a DE can be thought of as a processor - see section on the DELayout) is called a **LocalTile**. For example, the six faces of a cubed sphere grid are each Tiles, and each Tile can be divided into many LocalTiles.
Every ESMF Grid contains a DistGrid object, which defines the Grid's index space, topology, distribution, and connectivities. It enables the user to define the complex edge relationships of tripole and other grids. The DistGrid can be created explicitly and passed into a Grid creation routine, or it can be created implicitly if the user takes a Grid creation shortcut. The DistGrid used in Grid creation describes the properties of the Grid cells. In addition to this one, the Grid internally creates DistGrids for each stagger location. These stagger DistGrids are related to the original DistGrid, but may contain extra padding to represent the extent of the index space of the stagger. These DistGrids are what are used when a Field is created on a Grid.

27.1.2 Supported Grids

The range of supported grids in ESMF can be defined by:

- Types of topologies and shapes supported. ESMF supports one or more logically rectangular grid Tiles with connectivities specified between cells. For more details see section 27.1.3.
- Types of distributions supported. ESMF supports regular, irregular, or arbitrary distributions of data. For more details see section 27.1.4.
- Types of coordinates supported. ESMF supports uniform, rectilinear, and curvilinear coordinates. For more details see section 27.1.5.

27.1.3 Grid Topologies and Periodicity

ESMF has shortcuts for the creation of standard Grid topologies or **shapes** up to 3D. In many cases, these enable the user to bypass the step of creating a DistGrid before creating the Grid. There are two sets of methods which allow the user to do this. These two sets of methods cover the same set of topologies, but allow the user to specify them in different ways.

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

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

ESMF_GridCreate1PeriDim() Allows the user to create a Grid with 1 periodic dimension and supports a range of options for what to do at the pole (see Section 27.2.6. Some examples of Grids which can be created here are tripole spheres, bipole spheres, cylinders with open poles.

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

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

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

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

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

27.1.4 Grid Distribution

ESMF Grids have several options for data distribution (also referred to as decomposition). As ESMF Grids are cell based, these options are all specified in terms of how the cells in the Grid are broken up between DEs.
The main distribution options are regular, irregular, and arbitrary. A **regular** distribution is one in which the same number of contiguous grid cells are assigned to each DE in the distributed dimension. A **irregular** distribution is one in which unequal numbers of contiguous grid cells are assigned to each DE in the distributed dimension. An **arbitrary** distribution is one in which any grid cell can be assigned to any DE. Any of these distribution options can be applied to any of the grid shapes (i.e., rectangle) or types (i.e., rectilinear). Support for arbitrary distribution is limited in v5.2.0, See section 27.3.7 for more detail descriptions.
Figure 13 illustrates options for distribution.
A distribution can also be specified using the DistGrid, by passing object into a Grid create call.

27.1.5 Grid Coordinates

Grid Tiles can have uniform, rectilinear, or curvilinear coordinates. The coordinates of **uniform** grids are equally spaced along their axes, and can be fully specified by the coordinates of the two opposing points that define the grid's physical span. The coordinates of **rectilinear** grids are unequally spaced along their axes, and can be fully specified by giving the spacing of grid points along each axis. The coordinates of **curvilinear grids** must be specified by giving the explicit set of coordinates for each grid point. Curvilinear grids are often uniform or rectilinear grids that have been warped; for example, to place a pole over a land mass so that it does not affect the computations performed on an ocean model grid. Figure 14 shows examples of each type of grid.
Each of these coordinate types can be set for each of the standard grid shapes described in section 27.1.3.
The table below shows how examples of common single Tile grids fall into this shape and coordinate taxonomy. Note that any of the grids in the table can have a regular or arbitrary distribution.

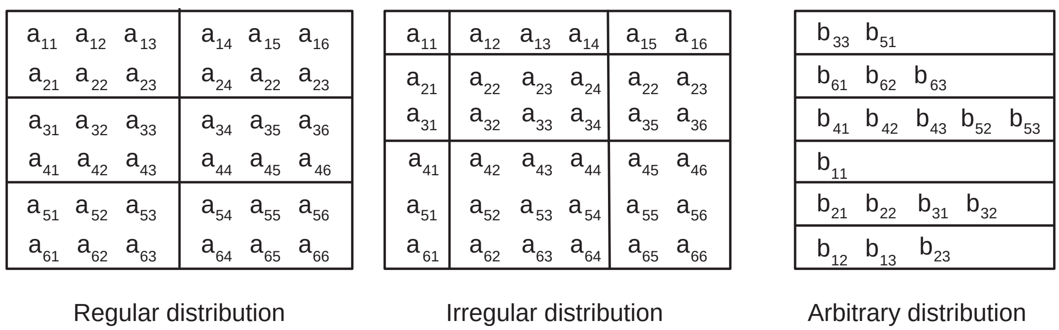


Figure 13: Examples of regular and irregular decomposition of a grid **a** that is 6x6, and an arbitrary decomposition of a grid **b** that is 6x3.

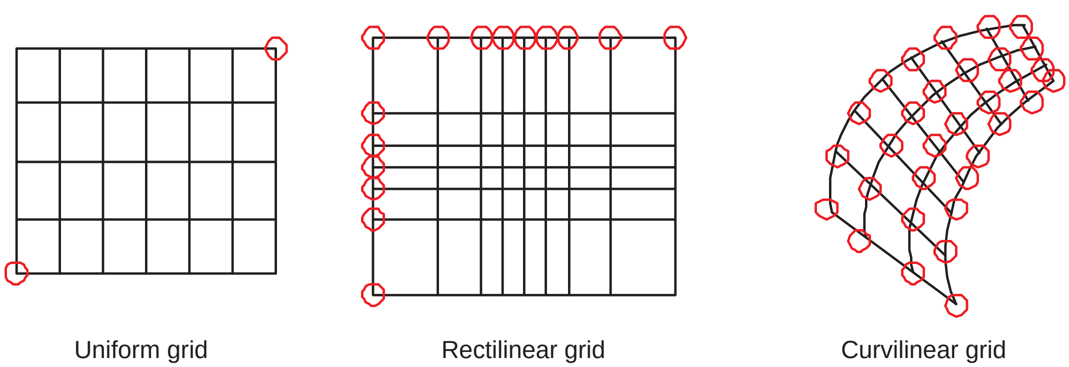


Figure 14: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

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

27.1.6 Coordinate Specification and Generation

There are two ways of specifying coordinates in ESMF. The first way is for the user to **set** the coordinates. The second way is to take a shortcut and have the framework **generate** the coordinates.
No ESMF generation routines are currently available.
See Section 27.3.10 for more description and examples of setting coordinates.

27.1.7 Staggering

Staggering is a finite difference technique in which the values of different physical quantities are placed at different locations within a grid cell.
The ESMF Grid class supports a variety of stagger locations, including cell centers, corners, and edge centers. The default stagger location in ESMF is the cell center, and cell counts in Grid are based on this assumption. Combinations of the 2D ESMF stagger locations are sufficient to specify any of the Arakawa staggers. ESMF also supports staggering in 3D and higher dimensions. There are shortcuts for standard staggers, and interfaces through which users can create custom staggers.
As a default the ESMF Grid class provides symmetric staggering, so that cell centers are enclosed by cell perimeter (e.g. corner) stagger locations. This means the coordinate arrays for stagger locations other than the center will have an additional element of padding in order to enclose the cell center locations. However, to achieve other types of staggering, the user may alter or eliminate this padding by using the appropriate options when adding coordinates to a Grid.
In v5.2.0, only the cell center stagger location is supported for an arbitrarily distributed grid. For examples and a full description of the stagger interface see Section 27.3.10.

27.2 Constants

27.2.1 ESMF_COORDSYS

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

ESMF_COORDSYS_CART Cartesian coordinate system.

ESMF_COORDSYS_SPH_DEG Spherical coordinates in degrees.

ESMF_COORDSYS_SPH_RAD Spherical coordinates in radians.

27.2.2 ESMF_GRIDCONN

DESCRIPTION:

The `ESMF_GridCreateShapeTile` command has three specific arguments `connflagDim1`, `connflagDim2`, and `connflagDim3`. These can be used to setup different types of connections at the ends of each dimension of a Tile. Each of these parameters is a two element array. The first element is the connection type at the minimum end of the dimension and the second is the connection type at the maximum end. The default value for all the connections is ESMF_GRIDCONN_NONE, specifying no connection.

The type of this flag is:
`type(ESMF_GridConn_Flag)`
The valid values are:

ESMF_GRIDCONN_NONE No connection.

ESMF_GRIDCONN_PERIODIC Periodic connection.

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

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

27.2.3 ESMF_GRIDITEM

DESCRIPTION:
The ESMF Grid can contain other kinds of data besides coordinates. This data is referred to as Grid "items". Some items may be used by ESMF for calculations involving the Grid. The following are the valid values of ESMF_GridItem_Flag.
The type of this flag is:
`type(ESMF_GridItem_Flag)`
The valid values are:

Item Label	**Type Restriction**	**Type Default**	**ESMF Uses**	**Controls**
ESMF_GRIDITEM_MASK	ESMF_TYPEKIND_I4	ESMF_TYPEKIND_I4	YES	Masking in Regrid
ESMF_GRIDITEM_AREA	NONE	ESMF_TYPEKIND_R8	NO	N/A

27.2.4 ESMF_GRIDMATCH

DESCRIPTION:
This type is used to indicate the level to which two grids match.
The type of this flag is:
`type(ESMF_GridMatch_Flag)`
The valid values are:

ESMF_GRIDMATCH_INVALID: Indicates a non-valid matching level. Returned if an error occurs in the matching function. If a higher matching level is returned then no error occurred.

ESMF_GRIDMATCH_NONE: The lowest level of grid matching. This indicates that the Grid's don't match at any of the higher levels.

ESMF_GRIDMATCH_EXACT: All the pieces of the Grid (e.g. distgrids, coordinates, etc.) except the name, match between the two Grids.

ESMF_GRIDMATCH_ALIAS: Both Grid variables are aliases to the exact same Grid object in memory.

27.2.5 ESMF_GRIDSTATUS

DESCRIPTION:
The ESMF Grid class can exist in two states. These states are present so that the library code can detect if a Grid has been appropriately setup for the task at hand. The following are the valid values of ESMF_GRIDSTATUS.
The type of this flag is:
`type(ESMF_GridStatus_Flag)`
The valid values are:

ESMF_GRIDSTATUS_EMPTY: Status after a Grid has been created with `ESMF_GridEmptyCreate`. A Grid object container is allocated but space for internal objects is not. Topology information and coordinate information is incomplete. This object can be used in `ESMF_GridEmptyComplete()` methods in which additional information is added to the Grid.

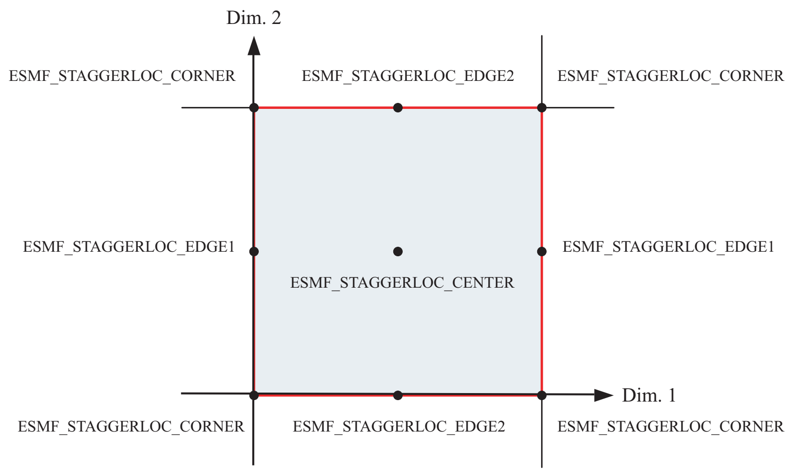


Figure 15: 2D Predefined Stagger Locations

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

27.2.6 ESMF_POLEKIND

DESCRIPTION:
This type describes the type of connection that occurs at the pole when a Grid is created with `ESMF_GridCreate1PeriodicDim()`.
The type of this flag is:
`type(ESMF_PoleKind_Flag)`
The valid values are:

ESMF_POLEKIND_NONE No connection at pole.

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

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

27.2.7 ESMF_STAGGERLOC

DESCRIPTION:
In the ESMF Grid class, data can be located at different positions in a Grid cell. When setting or retrieving coordinate data the stagger location is specified to tell the Grid method from where in the cell to get the data. Although the user may define their own custom stagger locations, ESMF provides a set of predefined locations for ease of use. The following are the valid predefined stagger locations.

The 2D predefined stagger locations (illustrated in figure 15) are:

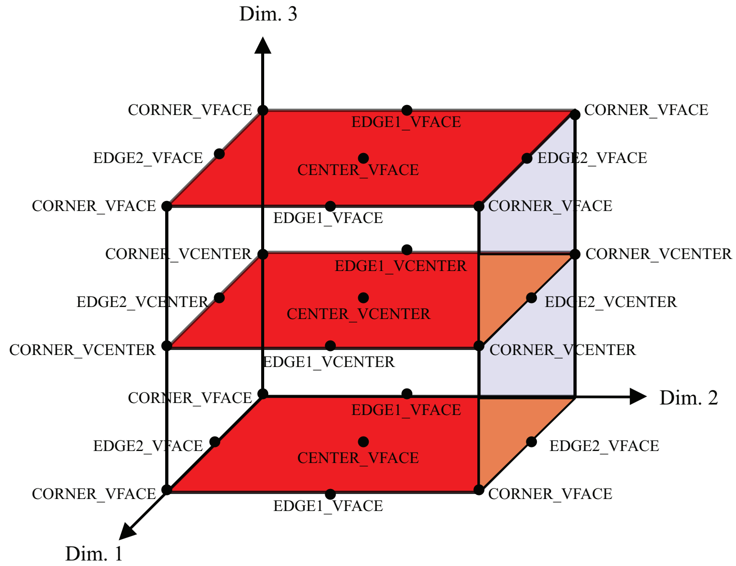


Figure 16: 3D Predefined Stagger Locations

ESMF_STAGGERLOC_CENTER: The center of the cell.

ESMF_STAGGERLOC_CORNER: The corners of the cell.

ESMF_STAGGERLOC_EDGE1: The edges offset from the center in the 1st dimension.

ESMF_STAGGERLOC_EDGE2: The edges offset from the center in the 2nd dimension.

The 3D predefined stagger locations (illustrated in figure 16) are:

ESMF_STAGGERLOC_CENTER_VCENTER: The center of the 3D cell.

ESMF_STAGGERLOC_CORNER_VCENTER: Half way up the vertical edges of the cell.

ESMF_STAGGERLOC_EDGE1_VCENTER: The center of the face bounded by edge 1 and the vertical dimension.

ESMF_STAGGERLOC_EDGE2_VCENTER: The center of the face bounded by edge 2 and the vertical dimension.

ESMF_STAGGERLOC_CORNER_VFACE: The corners of the 3D cell.

ESMF_STAGGERLOC_EDGE1_VFACE: The center of the edges of the 3D cell parallel offset from the center in the 1st dimension.

ESMF_STAGGERLOC_EDGE2_VFACE: The center of the edges of the 3D cell parallel offset from the center in the 2nd dimension.

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

27.3 Use and Examples

This section describes the use of the ESMF Grid class. It first discusses the more user friendly shape specific interface to the Grid. During this discussion it covers creation and options, adding stagger locations, coordinate data access, and other grid functionality. After this initial phase the document discusses the more advanced options which the user can employ should they need more customized interaction with the Grid class.

27.3.1 Create single-tile Grid shortcut method

The set of methods `ESMF_GridCreateNoPeriDim()`, `ESMF_GridCreate1PeriDim()`, `ESMF_GridCreate2PeriDim()`, `ESMF_GridCreate()` are shortcuts for building single tile logically rectangular Grids up to three dimensions.
In v5.1.0, these methods support all three types of distributions described in Section 27.1.4: regular, irregular and arbitrary.
The ESMF Grid is cell based and so for all distribution options the methods take as input the number of cells to describe the total index space and the number of cells to specify distribution.
To create a Grid with a regular distribution the user specifies the global maximum and minimum ranges of the Grid cell index space (`maxIndex` and `minIndex`), and the number of pieces in which to partition each dimension (via a `regDecomp` argument). ESMF then divides the index space as evenly as possible into the specified number of pieces. If there are cells left over then they are distributed one per DE starting from the first DE until they are gone.
If `minIndex` is not specified, then the bottom of the Grid cell index range is assumed to be (1,1,...,1). If `regDecomp` is not specified, then by default ESMF creates a distribution that partitions the grid cells in the first dimension (e.g. NPx1x1...1) as evenly as possible by the number of processors NP. The remaining dimensions are not partitioned. The dimension of the Grid is the size of `maxIndex`. The following is an example of creating a 10x20x30 3D grid where the first dimensions is broken into 2 pieces, the second is broken into 4 pieces, and the third is "distributed" across only one processor.

```
grid3D=ESMF_GridCreateNoPeriDim(regDecomp=(/2,4,1/), maxIndex=(/10,20,30/), &
         rc=rc)
```

Irregular distribution requires the user to specify the exact number of Grid cells per DE in each dimension. In the `ESMF_GridCreateNoPeriDim()` call the `countsPerDEDim1`, `countsPerDim2`, and `countsPerDim3` arguments are used to specify a rectangular distribution containing size(countsPerDEDim1) by size(countsPerDEDim2) by size(countsPerDEDim3) DEs. The entries in each of these arrays specify the number of grid cells per DE in that dimension. The dimension of the grid is determined by the presence of `countsPerDEDim3`. If it's present the Grid will be 3D. If just `countsPerDEDim1` and `countsPerDEDim2` are specified the Grid will be 2D.
The following call illustrates the creation of a 10x20 two dimensional rectangular Grid distributed across six DEs that are arranged 2x3. In the first dimension there are 3 grid cells on the first DE and 7 cells on the second DE. The second dimension has 3 DEs with 11,2, and 7 cells, respectively.

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/), &
       countsPerDEDim2=(/11,2,7/), rc=rc)
```

To add a distributed third dimension of size 30, broken up into two groups of 15, the above call would be altered as follows.

```
grid3d=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/), &
       countsPerDEDim2=(/11,2,7/), countsPerDEDim3=(/15,15/), rc=rc)
```

To make a third dimension distributed across only 1 DE, then `countsPerDEDim3` in the call should only have a single term.

```
grid3D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/),  &
       countsPerDEDim2=(/11,2,7/), countsPerDEDim3=(/30/), rc=rc)
```

The `petMap` parameter may be used to specify on to which specific PETs the DEs in the Grid are assigned. Note that this parameter is only available for the regular and irregular distribution types. The `petMap` array is a 3D array, for a 3D Grid each of its dimensions correspond to a Grid dimension. If the Grid is 2D, then the first two dimensions correspond to Grid dimensions and the last dimension should be of size 1. The size of each `petMap` dimension is the number of DE's along that dimension in the Grid. For a regular Grid, the size is equal to the number in regDecomp (i.e. `size(petMap,d)=regDecomp(d)` for all dimensions `d` in the Grid). For an irregular Grid the size is equal to the number of items in the corresponding `countsPerDEDim` variable (i.e. `size(petMap,d)=size(countsPerDEDimd)` for all dimensions `d` in the Grid).
The `petMap` parameter may be used to specify on to which specific PETs Each entry in `petMap` specifies to which PET the corresponding DE should be assigned. For example, `petMap(3,2)=4` tells the Grid create call to put the DE located at column 3 row 2 on PET 4.
The `petMap` parameter may be used to specify on to which specific PETs The following example demonstrates how to specify the PET to DE association for an `ESMF_GridCreateNoPeriDim()` call.

```
! allocate memory for petMap
allocate( petMap(2,2,1) )

! Set petMap
petMap(:,1,1) = (/3,2/) ! DE (1,1,1) on PET 3 and DE (2,1,1) on PET 2
petMap(:,2,1) = (/1,0/) ! DE (1,2,1) on PET 1 and DE (2,2,1) on PET 0


! Let the 3D grid be be distributed only in the first two dimensions.
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/), &
        countsPerDEDim2=(/7,6/), petMap=petMap, rc=rc)
```

To create an grid with arbitrary distribution, the user specifies the global minimum and maximum ranges of the index space with the arguments `minIndex` and `maxIndex`, the total number of cells and their index space locations residing on the local PET through a `localArbIndexCount` and a `localArbIndex` argument. `localArbIndex` is a 2D array with size `(localArbIndexCount, n)` where n is the total number dimensions distributed arbitrarily. Again, if `minIndex` is not specified, then the bottom of the index range is assumed to be (1,1,...). The dimension of the Grid is equal to the size of `maxIndex`. If n (number of arbitrarily distributed dimension) is less than the grid dimension, an optional argument `distDim` is used to specify which of the grid dimension is arbitrarily distributed. If not given, the first n dimensions are assumed to be distributed.
The following example creates a 2D Grid of dimensions 5x5, and places the diagonal elements (i.e. indices (i,i) where i goes from 1 to 5) on the local PET. The remaining PETs would individually declare the remainder of the Grid locations.

```
! allocate memory for localArbIndex
allocate( localArbIndex(5,2) )
! Set local indices
localArbIndex(1,:)=(/1,1/)
localArbIndex(2,:)=(/2,2/)
localArbIndex(3,:)=(/3,3/)
localArbIndex(4,:)=(/4,4/)
localArbIndex(5,:)=(/5,5/)

! Create a 2D Arbitrarily distributed Grid
grid2D=ESMF_GridCreateNoPeriDim(maxIndex=(/5,5/), &
      arbIndexList=localArbIndex, arbIndexCount=5, rc=rc)
```

To create a 3D Grid of dimensions 5x6x5 with the first and the third dimensions distributed arbitrarily, `distDim` is used.

```
! Create a 3D Grid with the 1st and 3rd dimension arbitrarily distributed
grid3D=ESMF_GridCreateNoPeriDim(maxIndex=(/5,6,5/), &
      arbIndexList=localArbIndex, arbIndexCount=5, &
      distDim=(/1,3/), rc=rc)
```

27.3.2 Create a 2D regularly distributed rectilinear Grid with uniformly spaced coordinates

The following is an example of creating a simple rectilinear grid and loading in a set of coordinates. It illustrates a straightforward use of the `ESMF_GridCreateNoPeriDim()` call described in the previous section. This code creates a 10x20 2D grid with uniformly spaced coordinates varying from (10,10) to (100,200). The grid is partitioned using a regular distribution. The first dimension is divided into two pieces, and the second dimension is divided into 3. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!-------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! topology and distribution of the Grid, and specify that it
! will have global indices.  Note that here aperiodic bounds are
! specified by the argument name. In this call the minIndex hasn't
! been set, so it defaults to (1,1,...). The default is to
! divide the index range as equally as possible among the DEs
! specified in regDecomp. This behavior can be changed by
! specifying decompFlag.
!-------------------------------------------------------------------
grid2D=ESMF_GridCreateNoPeriDim(          &
      ! Define a regular distribution
      maxIndex=(/10,20/), & ! define index space
      regDecomp=(/2,3/),  & ! define how to divide among DEs
      coordSys=ESMF_COORDSYS_CART, &
      ! Specify mapping of coords dim to Grid dim
      coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
      coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
      indexflag=ESMF_INDEX_GLOBAL, &
      rc=rc)

!-------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!-------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)

!-------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
```

```
       farrayPtr=coordX, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!-------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
     coordX(i) = i*10.0
enddo

!-------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!-------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
     coordY(j) = j*10.0
enddo
```

27.3.3 Create a periodic 2D regularly distributed rectilinear Grid

The following is an example of creating a simple rectilinear grid with a periodic dimension and loading in a set of coordinates. It illustrates a straightforward use of the `ESMF_GridCreate1PeriDim()` call described in the previous section. This code creates a 10x20 2D grid with uniformly spaced coordinates varying from (1,1) to (360,180). The grid is partitioned using a regular distribution. The first dimension is divided into two pieces, and the second dimension is divided into 3. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!-------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! topology and distribution of the Grid, and specify that it
! will have global indices.  Note that here a single periodic connection
! is specified by the argument name. In this call the minIndex hasn't
! been set, so it defaults to (1,1,...). The default is to
! divide the index range as equally as possible among the DEs
! specified in regDecomp. This behavior can be changed by
! specifying decompFlag. Since the coordinate system is
! not specified, it defaults to ESMF_COORDSYS_SPH_DEG.
!-------------------------------------------------------------------
grid2D=ESMF_GridCreate1PeriDim(          &
      ! Define a regular distribution
      maxIndex=(/360,180/), & ! define index space
      regDecomp=(/2,3/),  & ! define how to divide among DEs
      ! Specify mapping of coords dim to Grid dim
      coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
      coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
      indexflag=ESMF_INDEX_GLOBAL, &
```

```
      rc=rc)

!-------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!-------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)

!-------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!-------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
     coordX(i) = i*1.0
enddo

!-------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!-------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
     coordY(j) = j*1.0
enddo
```

The remaining examples in this section will use the irregular distribution because of its greater generality. To create code similar to these, but using a regular distribution, replace the `countsPerDEDim` arguments in the Grid create with the appropriate `maxIndex` and `regDecomp` arguments.

27.3.4 Create a 2D irregularly distributed rectilinear Grid with uniformly spaced coordinates

This example serves as an illustration of the difference between using a regular and irregular distribution. It repeats the previous example except using an irregular distribution to give the user more control over how the cells are divided between the DEs. As before, this code creates a 10x20 2D Grid with uniformly spaced coordinates varying from (10,10) to (100,200). In this example, the Grid is partitioned using an irregular distribution. The first dimension is divided into two pieces, the first with 3 Grid cells per DE and the second with 7 Grid cells per DE. In the second dimension, the Grid is divided into 3 pieces, with 11, 2, and 7 cells per DE respectively. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each

PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!-------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! topology and distribution of the Grid, and specify that it
! will have global coordinates.  Note that aperiodic bounds are
! indicated by the method name. In this call the minIndex hasn't
! been set, so it defaults to (1,1,...).
!-------------------------------------------------------------------
grid2D=ESMF_GridCreateNoPeriDim(          &
         ! Define an irregular distribution
         countsPerDEDim1=(/3,7/),    &
         countsPerDEDim2=(/11,2,7/), &
         ! Specify mapping of coords dim to Grid dim
         coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
         coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
         indexflag=ESMF_INDEX_GLOBAL, &
         rc=rc)

!-------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!-------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)

!-------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!-------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
     coordX(i) = i*10.0
enddo

!-------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
```

```
!------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
     coordY(j) = j*10.0
enddo
```

27.3.5 Create a 2D irregularly distributed Grid with curvilinear coordinates

The following is an example of creating a simple curvilinear Grid and loading in a set of coordinates. It creates a 10x20 2D Grid where the coordinates vary along every dimension. The Grid is partitioned using an irregular distribution. The first dimension is divided into two pieces, the first with 3 Grid cells per DE and the second with 7 Grid cells per DE. In the second dimension, the Grid is divided into 3 pieces, with 11, 2, and 7 cells per DE respectively. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! distribution of the Grid, and specify that it
! will have global indices.  Note that aperiodic bounds are
! indicated by the method name. If periodic bounds were desired they
! could be specified by using the ESMF_GridCreate1PeriDim() call.
! In this call the minIndex hasn't been set, so it defaults to (1,1,...).
!------------------------------------------------------------------
grid2D=ESMF_GridCreateNoPeriDim(      &
     ! Define an irregular distribution
     countsPerDEDim1=(/3,7/),     &
     countsPerDEDim2=(/11,2,7/),   &
     ! Specify mapping of coords dim to Grid dim
     coordDep1=(/1,2/), & ! 1st coord is 2D and depends on both Grid dim
     coordDep2=(/1,2/), & ! 2nd coord is 1D and depends on both Grid dim
     indexflag=ESMF_INDEX_GLOBAL, &
     rc=rc)

!------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)

!------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX2D, rc=rc)

!------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!------------------------------------------------------------------
do j=lbnd(2),ubnd(2)
```

```
do i=lbnd(1),ubnd(1)
     coordX2D(i,j) = i+j
enddo
enddo

!-------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY2D, rc=rc)

!-------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!-------------------------------------------------------------------
do j=lbnd(2),ubnd(2)
do i=lbnd(1),ubnd(1)
     coordY2D(i,j) = j-i/100.0
enddo
enddo
```

27.3.6 Create an irregularly distributed rectilinear Grid with a non-distributed vertical dimension

This example demonstrates how a user can build a rectilinear horizontal Grid with a non-distributed vertical dimension. The Grid contains both the center and corner stagger locations (i.e. Arakawa B-Grid). In contrast to the previous examples, this example doesn't assume that the code is being run with a 1-1 mapping between PETs and DEs. It should work when run on any number of PETs.

```
!-------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object.  The
! Grid is defined to be 180 Grid cells in the first dimension
! (e.g. longitude), 90 Grid cells in the second dimension
! (e.g. latitude), and 40 Grid cells in the third dimension
! (e.g. height).  The first dimension is decomposed over 4 DEs,
! the second over 3 DEs, and the third is not distributed.
! The connectivities in each dimension are set to aperiodic
! by this method. In this call the minIndex hasn't been set,
! so it defaults to (1,1,...).
!-------------------------------------------------------------------
grid3D=ESMF_GridCreateNoPeriDim( &
         ! Define an irregular distribution
         countsPerDEDim1=(/45,75,40,20/), &
         countsPerDEDim2=(/30,40,20/),    &
         countsPerDEDim3=(/40/),          &
         ! Specify mapping of coords dim to Grid dim
         coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
         coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
         coordDep3=(/3/), & ! 3rd coord is 1D and depends on 3rd Grid dim
         indexflag=ESMF_INDEX_GLOBAL,     & ! Use global indices
         rc=rc)

!-------------------------------------------------------------------
! Allocate coordinate storage for both center and corner stagger
```

```
! locations.  Since no coordinate values are specified in this
! call no coordinate values are set yet.
!-------------------------------------------------------------------
call ESMF_GridAddCoord(grid3D, &
       staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER, rc=rc)
call ESMF_GridAddCoord(grid3D, &
       staggerloc=ESMF_STAGGERLOC_CORNER_VCENTER, rc=rc)


!-------------------------------------------------------------------
! Get the number of DEs on this PET, so that the program
! can loop over them when accessing data.
!-------------------------------------------------------------------
call ESMF_GridGet(grid3D, localDECount=localDECount, rc=rc)

!-------------------------------------------------------------------
! Loop over each localDE when accessing data
!-------------------------------------------------------------------
do lDE=0,localDECount-1

 !------------------------------------------------------------------
 ! Fill in the coordinates for the corner stagger location first.
 !------------------------------------------------------------------
   !----------------------------------------------------------------
   ! Get the local bounds of the global indexing for the first
   ! coordinate array on the local DE. If the number of PETs
   ! is less than the total number of DEs then the rest of this
   ! example would be in a loop over the local DEs.  Also get the
   ! pointer to the first coordinate array.
   !----------------------------------------------------------------
   call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=lDE, &
          staggerLoc=ESMF_STAGGERLOC_CORNER_VCENTER,       &
          computationalLBound=lbnd_corner,                 &
          computationalUBound=ubnd_corner,                 &
          farrayPtr=cornerX, rc=rc)

   !----------------------------------------------------------------
   ! Calculate and set coordinates in the first dimension.
   !----------------------------------------------------------------
   do i=lbnd_corner(1),ubnd_corner(1)
      cornerX(i) = (i-1)*(360.0/180.0)
   enddo

   !----------------------------------------------------------------
   ! Get the local bounds of the global indexing for the second
   ! coordinate array on the local DE.  Also get the pointer to the
   ! second coordinate array.
   !----------------------------------------------------------------
   call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=lDE,   &
          staggerLoc=ESMF_STAGGERLOC_CORNER_VCENTER,         &
          computationalLBound=lbnd_corner,                   &
          computationalUBound=ubnd_corner,                   &
          farrayPtr=cornerY, rc=rc)

   !----------------------------------------------------------------
```

```
  ! Calculate and set coordinates in the second dimension.
  !-------------------------------------------------------------------
  do j=lbnd_corner(1),ubnd_corner(1)
     cornerY(j) = (j-1)*(180.0/90.0)
  enddo

  !-------------------------------------------------------------------
  ! Get the local bounds of the global indexing for the third
  ! coordinate array on the local DE, and the pointer to the array.
  !-------------------------------------------------------------------
  call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=lDE,   &
         staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,         &
         computationalLBound=lbnd, computationalUBound=ubnd,&
         farrayPtr=cornerZ, rc=rc)

  !-------------------------------------------------------------------
  ! Calculate and set the vertical coordinates
  !-------------------------------------------------------------------
  do k=lbnd(1),ubnd(1)
     cornerZ(k) = 4000.0*( (1./39.)*(k-1)  )**2
  enddo

!--------------------------------------------------------------------
! Now fill the coordinates for the center stagger location with
! the average of the corner coordinate location values.
!--------------------------------------------------------------------
  !-------------------------------------------------------------------
  ! Get the local bounds of the global indexing for the first
  ! coordinate array on the local DE, and the pointer to the array.
  !-------------------------------------------------------------------
  call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=lDE,    &
         staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,          &
         computationalLBound=lbnd, computationalUBound=ubnd, &
         farrayPtr=centerX, rc=rc)

  !-------------------------------------------------------------------
  ! Calculate and set coordinates in the first dimension.
  !-------------------------------------------------------------------
  do i=lbnd(1),ubnd(1)
     centerX(i) = 0.5*(i-1 + i)*(360.0/180.0)
  enddo

  !-------------------------------------------------------------------
  ! Get the local bounds of the global indexing for the second
  ! coordinate array on the local DE, and the pointer to the array.
  !-------------------------------------------------------------------
   call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=lDE,    &
          staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,          &
          computationalLBound=lbnd, computationalUBound=ubnd, &
          farrayPtr=centerY, rc=rc)

  !-------------------------------------------------------------------
  ! Calculate and set coordinates in the second dimension.
  !-------------------------------------------------------------------
  do j=lbnd(1),ubnd(1)
```

```
      centerY(j) = 0.5*(j-1 + j)*(180.0/90.0)
   enddo

   !------------------------------------------------------------------
   ! Get the local bounds of the global indexing for the third
   ! coordinate array on the local DE, and the pointer to the array.
   !------------------------------------------------------------------
   call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=lDE,   &
          staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,         &
          computationalLBound=lbnd, computationalUBound=ubnd,&
          farrayPtr=centerZ, rc=rc)

   !------------------------------------------------------------------
   ! Calculate and set the vertical coordinates
   !------------------------------------------------------------------
   do k=lbnd(1),ubnd(1)
      centerZ(k) = 4000.0*( (1./39.)*(k-1)  )**2
   enddo

!---------------------------------------------------------------------
! End of loop over DEs
!---------------------------------------------------------------------
enddo
```

27.3.7 Create an arbitrarily distributed rectilinear Grid with a non-distributed vertical dimension

There are more restrictions in defining an arbitrarily distributed grid. First, there is always one DE per PET. Secondly, only local index (ESMF_INDEX_LOCAL) is supported. Third, only one stagger location, i.e. ESMF_STAGGERLOC_CENTER is allowed and last there is no extra paddings on the edge of the grid.
This example demonstrates how a user can build a 3D grid with its rectilinear horizontal Grid distributed arbitrarily and a non-distributed vertical dimension.

```
!---------------------------------------------------------------------
! Set up the local index array:  Assuming the grid is 360x180x10.  First
! calculate the localArbIndexCount and localArbIndex array for each PET
! based on the total number of PETS. The cells are evenly distributed in
! all the PETs. If the total number of cells are not divisible by the
! total PETs, the remaining cells are assigned to the last PET.  The
! cells are card dealed to each PET in y dimension first,
! i.e. (1,1) -> PET 0, (1,2)->  PET 1, (1,3)-> PET 2, and so forth.
!---------------------------------------------------------------------
xdim = 360
ydim = 180
zdim = 10
localArbIndexCount = (xdim*ydim)/petCount
remain = (xdim*ydim)-localArbIndexCount*petCount
if (localPet == petCount-1) localArbIndexCount = localArbIndexCount+remain

allocate(localArbIndex(localArbIndexCount,2))
ind = localPet
do i=1, localArbIndexCount
   localArbIndex(i,1)=mod(ind,ydim)+1
   localArbIndex(i,2)=ind/ydim + 1
```

```
   ind = ind + petCount
enddo
if (localPet == petCount-1) then
   ind = xdim*ydim-remain+1
   do i=localArbIndexCount-remain+1,localArbIndexCount
      localArbIndex(i,1)=mod(ind,ydim)+1
      localArbIndex(i,2)=ind/ydim+1
      ind = ind + 1
   enddo
endif

!-------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object.
! the minIndex hasn't been set, so it defaults to (1,1,...). The
! default coordDep1 and coordDep2 are (/ESMF_DIM_ARB/) where
! ESMF_DIM_ARB represents the collapsed dimension for the
! arbitrarily distributed grid dimensions.  For the undistributed
! grid dimension, the default value for coordDep3 is (/3/).  The
! default values for coordDepX in the arbitrary distribution are
! different from the non-arbitrary distributions.
!-------------------------------------------------------------------
grid3D=ESMF_GridCreateNoPeriDim( &
 maxIndex = (/xdim, ydim, zdim/), &
         arbIndexList = localArbIndex, &
         arbIndexCount = localArbIndexCount, &
         rc=rc)

!-------------------------------------------------------------------
! Allocate coordinate storage for the center stagger location, the
! only stagger location supported for the arbitrary distribution.
!-------------------------------------------------------------------
call ESMF_GridAddCoord(grid3D, &
       staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER, rc=rc)

!------------------------------------------------------------------
! Fill in the coordinates for the center stagger location. There is
! always one DE per PET, so localDE is always 0
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=0, &
       staggerLoc=ESMF_STAGGERLOC_CENTER,       &
       computationalLBound=lbnd,                 &
       computationalUBound=ubnd,                 &
       farrayPtr=centerX, rc=rc)


!----------------------------------------------------------------
! Calculate and set coordinates in the first dimension.
!----------------------------------------------------------------
do i=lbnd(1),ubnd(1)
   centerX(i) = (localArbIndex(i,1)-0.5)*(360.0/xdim)
enddo


!----------------------------------------------------------------
! Get the local bounds of the global indexing for the second
```

```
! coordinate array on the local DE, and the pointer to the array.
!----------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=0,    &
       staggerloc=ESMF_STAGGERLOC_CENTER,                &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=centerY, rc=rc)

!----------------------------------------------------------------
! Calculate and set coordinates in the second dimension.
!----------------------------------------------------------------
do j=lbnd(1),ubnd(1)
   centerY(j) = (localArbIndex(j,2)-0.5)*(180.0/ydim)-90.0
enddo

!----------------------------------------------------------------
! Get the local bounds of the global indexing for the third
! coordinate array on the local DE, and the pointer to the array.
!----------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=0,   &
       staggerloc=ESMF_STAGGERLOC_CENTER,             &
       computationalLBound=lbnd, computationalUBound=ubnd,&
       farrayPtr=centerZ, rc=rc)

!----------------------------------------------------------------
! Calculate and set the vertical coordinates
!----------------------------------------------------------------
do k=lbnd(1),ubnd(1)
   centerZ(k) = 4000.0*( (1./zdim)*(k-1))**2
enddo
```

27.3.8 Create a curvilinear Grid using the coordinates defined in a SCRIP file

ESMF supports the creation of a 2D curvilinear Grid using the coordinates defined in a SCRIP format Grid file [13]. The grid contained in the file must be a 2D logically rectangular grid with `grid_rank` in the file set to 2. The center coordinates variables `grid_center_lat` and `grid_center_lon` in the file are placed in the ESMF_STAGGERLOC_CENTER location. If the parameter `addCornerStagger` in the `ESMF_GridCreate` call is set to .true., then the variables `grid_corner_lat` and `grid_corner_lon` in the file are used to set the ESMF_STAGGERLOC_CORNER coordinates, otherwise they are ignored. The values in the `grid_imask` variable in the file are used to set the ESMF_GRIDITEM_MASK in the Grid.
The following example code shows you how to create a 2D Grid with both center and corner coordinates using a SCRIP file and a row only regular distribution:

```
grid2D = ESMF_GridCreate(filename="data/T42_grid.nc", &
           regDecomp=(/PetCount,1/), addCornerStagger=.true., rc=rc)
```

Where T42_grid.nc is a 2D global grid of size (128x64) and the resulting Grid is distributed by partitioning the rows evenly over all the PETs.

27.3.9 Create an empty Grid in a parent Component for completion in a child Component

ESMF Grids can be created incrementally. To do this, the user first calls `ESMF_GridEmptyCreate()` to allocate the shell of a Grid. Next, we use the `ESMF_GridEmptyComplete()` call that fills in the Grid and does an internal commit to make it usable. For consistency's sake the `ESMF_GridSetCommitShapeTile()` call must occur on the same or a subset of the PETs as the `ESMF_GridEmptyCreate()` call. The `ESMF_GridEmptyComplete()` call uses the VM for the context in which it's executed and the "empty" Grid contains no information about the VM in which its create was run. This means that if the `ESMF_GridEmptyComplete()` call occurs in a subset of the PETs in which the `ESMF_GridEmptyCreate()` was executed that the Grid is created only in that subset. Inside the subset the Grid will be fine, but outside the subset the Grid objects will still be "empty" and not usable. The following example uses the incremental technique to create a rectangular 10x20 Grid with coordinates at the center and corner stagger locations.

```
!---------------------------------------------------------------------------
! IN THE PARENT COMPONENT:
! Create an empty Grid in the parent component for use in a child component.
! The parent may be defined on more PETs than the child component.
! The child's [vm or pet list] is passed into the create call so that
! the Grid is defined on the appropriate subset of the parent's PETs.
!---------------------------------------------------------------------------
   grid2D=ESMF_GridEmptyCreate(rc=rc)

!---------------------------------------------------------------------------
! IN THE CHILD COMPONENT:
! Set the Grid topology.  Here we define an irregularly distributed
! rectangular Grid.
!---------------------------------------------------------------------------
   call ESMF_GridEmptyComplete(grid2D,              &
                          countsPerDEDim1=(/6,4/),      &
                          countsPerDEDim2=(/10,3,7/), rc=rc)

!---------------------------------------------------------------------------
! Add Grid coordinates at the cell center location.
!---------------------------------------------------------------------------
   call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CENTER, rc=rc)

!---------------------------------------------------------------------------
! Add Grid coordinates at the corner stagger location.
!---------------------------------------------------------------------------
   call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, rc=rc)
```

27.3.10 Grid stagger locations

A useful finite difference technique is to place different physical quantities at different locations within a grid cell. This *staggering* of the physical variables on the mesh is introduced so that the difference of a field is naturally defined at the location of another variable. This method was first formalized by Mesinger and Arakawa (1976).
To support the staggering of variables, the Grid provides the idea of *stagger locations*. Stagger locations refer to the places in a Grid cell that can contain coordinates or other data and once a Grid is associated with a Field object, field data. Typically Grid data can be located at the cell center, at the cell corners, or at the cell faces, in 2D, 3D, and higher dimensions. (Note that any Arakawa stagger can be constructed of a set of Grid stagger locations.) There are predefined stagger locations (see Section 27.2.7), or, should the user wish to specify their own, there is also a set of methods for generating custom locations (See Section 27.3.22). Users can put Grid data (e.g. coordinates) at multiple stagger locations in a Grid. In addition, the user can create a Field at any of the stagger locations in a Grid.
By default the Grid data array at the center stagger location starts at the bottom index of the Grid (default (1,1..,1)) and extends up to the maximum cell index in the Grid (e.g. given by the `maxIndex` argument). Other stagger locations also start at the bottom index of the Grid, however, they can extend to +1 element beyond the center in some dimensions to allow for the extra space to surround the center elements. See Section 27.3.22 for a description of this extra space and how to adjust if it necessary. There are `ESMF_GridGet` subroutines (e.g. `ESMF_GridGetCoord()` or `ESMF_GridGetItem()`) which can be used to retrieve the stagger bounds for the piece of Grid data on a particular

DE.

27.3.11 Associate coordinates with stagger locations

The primary type of data the Grid is resposible for storing is coordinates. The coordinate values in a Grid can be employed by the user in calculations or to describe the geometry of a Field. The Grid coordinate values are also used by `ESMF_FieldRegridStore()` when calculating the interpolation matrix between two Fields. The user can allocate coordinate arrays without setting coordinate values using the `ESMF_GridAddCoord()` call. (See Section 27.3.13 for a discussion of setting/getting coordinate values.) When adding or accessing coordinate data, the stagger location is specified to tell the Grid method where in the cell to get the data. The different stagger locations may also have slightly different index ranges and sizes. Please see Section 27.3.10 for a discussion of Grid stagger locations.
The following example adds coordinate storage to the corner stagger location in a Grid using one of the predefined stagger locations.

```
call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, rc=rc)
```

Note only the center stagger location `ESMF_STAGGERLOC_CENTER` is supported in an arbitrarily distributed Grid.

27.3.12 Specify the relationship of coordinate Arrays to index space dimensions

To specify how the coordinate arrays are mapped to the index dimensions the arguments `coordDep1`, `coordDep2`, and `coordDep3` are used, each of which is a Fortran array. The values of the elements in a `coordDep` array specify which index dimension the corresponding coordinate dimension maps to. For example, `coordDep1=(/1,2/)` means that the first dimension of coordinate 1 maps to index dimension 1 and the second maps to index dimension 2. For a grid with non-arbitrary distribution, the default values for `coordDep1`, `coordDep2` and `coordDep3` are `/1,2..,gridDimCount/`. This default thus specifies a curvilinear grid.
The following call demonstrates the creation of a 10x20 2D rectilinear grid where the first coordinate component is mapped to the second index dimension (i.e. is of size 20) and the second coordinate component is mapped to the first index dimension (i.e. is of size 10).

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/5,5/), &
       countsPerDEDim2=(/7,7,6/),                        &
       coordDep1=(/2/),                                  &
       coordDep2=(/1/), rc=rc)
```

The following call demonstrates the creation of a 10x20x30 2D plus 1 curvilinear grid where coordinate component 1 and 2 are still 10x20, but coordinate component 3 is mapped just to the third index dimension.

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/6,4/), &
       countsPerDEDim2=(/10,7,3/), countsPerDEDim3=(/30/), &
       coordDep1=(/1,2/), coordDep2=(/1,2/), &
       coordDep3=(/3/), rc=rc)
```

By default the local piece of the array on each PET starts at (1,1,..), however, the indexing for each grid coordinate array on each DE may be shifted to the global indices by using the `indexflag`. For example, the following call switches the grid to use global indices.

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/6,4/), &
        countsPerDEDim2=(/10,7,3/), indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

For an arbitrarily distributed grid, the default value of a coordinate array dimension is `ESMF_DIM_ARB` if the index dimension is arbitrarily distributed and is `n` where `n` is the index dimension itself when it is not distributed. The following call is equivalent to the example in Section 27.3.7

```
grid3D=ESMF_GridCreateNoPeriDim( &
 maxIndex = (/xdim, ydim, zdim/), &
         arbIndexList = localArbIndex, &
         arbIndexCount = localArbIndexCount, &
   coordDep1 = (/ESMF_DIM_ARB/), &
 coordDep2 = (/ESMF_DIM_ARB/), &
 coordDep3 = (/3/), &
         rc=rc)
```

The following call uses non-default `coordDep1`, `coordDep2`, and `coordDep3` to create a 3D curvilinear grid with its horizontal dimensions arbitrarily distributed.

```
grid3D=ESMF_GridCreateNoPeriDim( &
 maxIndex = (/xdim, ydim, zdim/), &
         arbIndexList = localArbIndex, &
         arbIndexCount = localArbIndexCount, &
   coordDep1 = (/ESMF_DIM_ARB, 3/), &
 coordDep2 = (/ESMF_DIM_ARB, 3/), &
 coordDep3 = (/ESMF_DIM_ARB, 3/), &
         rc=rc)
```

27.3.13 Access coordinates

Once a Grid has been created, the user has several options to access the Grid coordinate data. The first of these, `ESMF_GridSetCoord()`, enables the user to use ESMF Arrays to set data for one stagger location across the whole Grid. For example, the following sets the coordinates in the first dimension (e.g. x) for the corner stagger location to those in the ESMF Array `arrayCoordX`.

```
call ESMF_GridSetCoord(grid2D, &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       coordDim=1, array=arrayCoordX, rc=rc)
```

The method `ESMF_GridGetCoord()` allows the user to obtain a reference to an ESMF Array which contains the coordinate data for a stagger location in a Grid. The user can then employ any of the standard `ESMF_Array` tools to operate on the data. The following copies the coordinates from the second component of the corner and puts it into the ESMF Array `arrayCoordY`.

```
call ESMF_GridGetCoord(grid2D,    &
       staggerLoc=ESMF_STAGGERLOC_CORNER,    &
       coordDim=2,                           &
       array=arrayCoordY, rc=rc)
```

Alternatively, the call `ESMF_GridGetCoord()` gets a Fortran pointer to the coordinate data. The user can then operate on this array in the usual manner. The following call gets a reference to the Fortran array which holds the data for the second coordinate (e.g. y).

```
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CORNER, farrayPtr=coordY2D, rc=rc)
```

27.3.14 Associate items with stagger locations

The ESMF Grids contain the ability to store other kinds of data beyond coordinates. These kinds of data are referred to as "items". Although the user is free to use this data as they see fit, the user should be aware that this data may

also be used by other parts of ESMF (e.g. the ESMF_GRIDITEM_MASK item is used in regridding). Please see Section 27.2.3 for a list of valid items.
Like coordinates items are also created on stagger locations. When adding or accessing item data, the stagger location is specified to tell the Grid method where in the cell to get the data. The different stagger locations may also have slightly different index ranges and sizes. Please see Section 27.3.10 for a discussion of Grid stagger locations. The user can allocate item arrays without setting item values using the `ESMF_GridAddItem()` call. (See Section 27.3.15 for a discussion of setting/getting item values.)
The following example adds mask item storage to the corner stagger location in a grid.

```
call ESMF_GridAddItem(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, &
       itemflag=ESMF_GRIDITEM_MASK, rc=rc)
```

27.3.15 Access items

Once an item has been added to a Grid, the user has several options to access the data. The first of these, `ESMF_GridSetItem()`, enables the user to use ESMF Arrays to set data for one stagger location across the whole Grid. For example, the following sets the mask item in the corner stagger location to those in the ESMF Array `arrayMask`.

```
call ESMF_GridSetItem(grid2D,             &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       itemflag=ESMF_GRIDITEM_MASK, array=arrayMask, rc=rc)
```

The method `ESMF_GridGetItem()` allows the user to get a reference to the Array which contains item data for a stagger location on a Grid. The user can then employ any of the standard `ESMF_Array` tools to operate on the data. The following gets the mask data from the corner and puts it into the ESMF Array `arrayMask`.

```
call ESMF_GridGetItem(grid2D,             &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       itemflag=ESMF_GRIDITEM_MASK,           &
       array=arrayMask, rc=rc)
```

Alternatively, the call `ESMF_GridGetItem()` gets a Fortran pointer to the item data. The user can then operate on this array in the usual manner. The following call gets a reference to the Fortran array which holds the data for the mask data.

```
call ESMF_GridGetItem(grid2D, localDE=0,   &
       staggerloc=ESMF_STAGGERLOC_CORNER,  &
       itemflag=ESMF_GRIDITEM_MASK, farrayPtr=mask2D, rc=rc)
```

27.3.16 Grid regions and bounds

Like an Array or a Field, the index space of each stagger location in the Grid contains an exclusive region, a computational region and a total region. Please see Section 24.2.6 for an in depth description of these regions.
The exclusive region is the index space defined by the distgrid of each stagger location of the Grid. This region is the region which is owned by the DE and is the region operated on by communication methods such as `ESMF_FieldRegrid()`. The exclusive region for a stagger location is based on the exclusive region defined by the DistGrid used to create the Grid. The size of the stagger exclusive region is the index space for the Grid cells, plus the stagger padding.
The default stagger padding depends on the topology of the Grid. For an unconnected dimension the stagger padding is a width of 1 on the upper side (i.e. `gridEdgeUWidth=(1,1,1,1...)`). For a periodic dimension there is no stagger padding. By adjusting `gridEdgeLWidth` and `gridEdgeUWidth`, the user can set the stagger padding

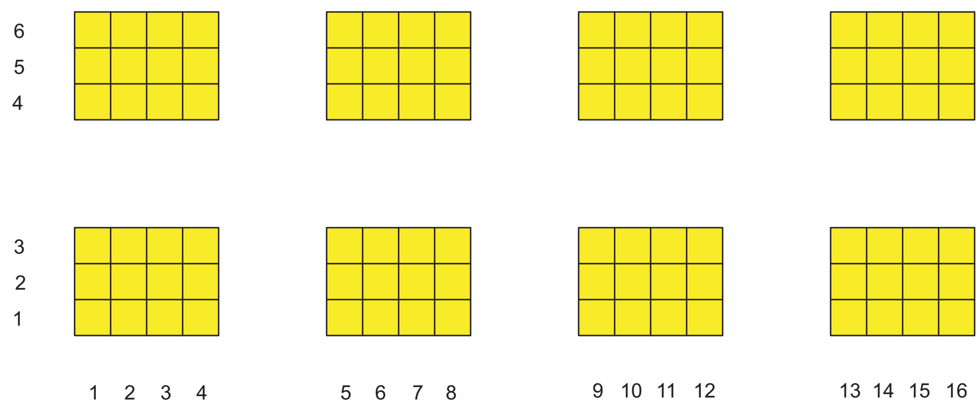


Figure 17: An example of a Grid's exclusive region for the corner stagger

for the whole Grid and thus the exclusive region can be adjusted at will around the index space corresponding to the cells. The user can also use `staggerEdgeLWidth` and `staggerEdgeUWidth` to adjust individual stagger location padding within the Grid's padding (Please see Section 27.3.23 for further discussion of customizing the stagger padding).

Figure 17 shows an example of a Grid exclusive region for the `ESMF_STAGGERLOC_CORNER` stagger with default stagger padding. This exclusive region would be for a Grid generated by either of the following calls:

```
grid2D=ESMF_GridCreateNoPeriDim(regDecomp=(/2,4/), maxIndex=(/5,15/), &
         indexflag=ESMF_INDEX_GLOBAL, rc=rc)


grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/4,4,4,3/), &
         countsPerDEDim2=(/3,2/), indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

Each rectangle in this diagram represents a DE and the numbers along the sides are the index values of the locations in the DE. Note that the exclusive region has one extra index location in each dimension than the number of cells because of the padding for the larger corner stagger location.
The computational region is a user setable region which can be used to distinguish a particular area for computation. The Grid doesn't currently contain functionality to let the user set the computational region so it defaults to the exclusive region, however, if the user sets an Array holding different computational bounds into the Grid then that Array's computational bounds will be used.
The total region is the outermost boundary of the memory allocated on each DE to hold the data for the stagger location on that DE. This region can be as small as the exclusive region, but may be larger to include space for halos, memory padding, etc. The total region is what is enlarged to include space for halos, and the total region must be large enough to contain the maximum halo operation on the Grid. The Grid doesn't currently contain functionality to let the user set the total region so it defaults to the exclusive region, however, if the user sets an Array holding different total bounds into the Grid then that Array's total bounds will be used.
The user can retrieve a set of bounds for each index space region described above: exclusive bounds, computational bounds, and total bounds. Note that although some of these are similar to bounds provided by ESMF_Array subroutines (see Section 24.2.6) the format here is different. The Array bounds are only for distributed dimensions and are ordered to correspond to the dimension order in the associated DistGrid. The bounds provided by the Grid are ordered

according to the order of dimensions of the data in question. This means that the bounds provided should be usable "as is" to access the data.
Each of the three types of bounds refers to the maximum and minimum per dimension of the index ranges of a particular region. The paramters referring to the maximums contain a 'U' for upper. The parameters referring to the minimums contain an 'L' for lower. The bounds and associated quantities are almost always given on a per DE basis. The three types of bounds `exclusiveBounds`, `computationalBounds`, and `totalBounds` refer to the ranges of the exlusive region, the computational region, and the total region. Each of these bounds also has a corresponding count parameter which gives the number of items across that region (on a DE) in each dimension. (e.g. `totalCount(d)=totallUBound(i)-totalLBound(i)+1`). Width parameters give the spacing between two different types of region. The `computationalWidth` argument gives the spacing between the exclusive region and the computational region. The `totalWidth` argument gives the spacing between the total region and the computational region. Like the other bound information these are typically on a per DE basis, for example specifying `totalLWidth=(1,1)` makes the bottom of the total region one lower in each dimension than the computational region on each DE. The exceptions to the per DE rule are `staggerEdgeWidth`, and `gridEdgeWidth` which give the spacing only on the DEs along the boundary of the Grid.
All the above bound discussions only apply to the grid with non-arbitrary distributions, i.e., regular or irregular distributions. For an arbitrarily distributed grid, only center stagger location is supported and there is no padding around the grid. Thus, the exclusive bounds, the total bounds and the computational bounds are identical and `staggerEdgeWidth`, and `gridEdgeWidth` are all zeros.

27.3.17 Get Grid coordinate bounds

When operating on coordinates the user may often wish to retrieve the bounds of the piece of coordinate data on a particular local DE. This is useful for iterating through the data to set coordinates, retrieve coordinates, or do calculations. The method `ESMF_GridGetCoord` allows the user to retrieve bound information for a particular coordinate array.
As described in the previous section there are three types of bounds the user can get: exclusive bounds, computational bounds, and total bounds. The bounds provided by `ESMF_GridGetCoordBounds` are for both distributed and undistributed dimensions and are ordered according to the order of dimensions in the coordinate. This means that the bounds provided should be usable "as is" to access data in the coordinate array. In the case of factorized coordinate Arrays where a coordinate may have a smaller dimension than its associated Grid, then the dimension of the coordinate's bounds are the dimension of the coordinate, not the Grid.
The following is an example of retrieving the bounds for localDE 0 for the first coordinate array from the corner stagger location.

```
call ESMF_GridGetCoordBounds(grid2D, coordDim=1, localDE=0,  &
       staggerLoc=ESMF_STAGGERLOC_CORNER,                          &
       exclusiveLBound=elbnd, exclusiveUBound=eubnd,               &
       computationalLBound=clbnd, computationalUBound=cubnd,       &
       totalLBound=tlbnd, totalUBound=tubnd, rc=rc)
```

27.3.18 Get Grid stagger location bounds

When operating on data stored at a particular stagger in a Grid the user may find it useful to be able to retrieve the bounds of the data on a particular local DE. This is useful for iterating through the data for computations or allocating arrays to hold the data. The method `ESMF_GridGet` allows the user to retrieve bound information for a particular stagger location.
As described in Section 27.3.16 there are three types of bounds the user can typically get, however, the Grid doesn't hold data at a stagger location (that is the job of the Field), and so no Array is contained there and so no total region exists, so the user may only retrieve exclusive and computational bounds from a stagger location. The bounds provided by `ESMF_GridGet` are ordered according to the order of dimensions in the Grid.
The following is an example of retrieving the bounds for localDE 0 from the corner stagger location.

```
call ESMF_GridGet(grid2D, localDE=0,                         &
       staggerLoc=ESMF_STAGGERLOC_CORNER,                     &
       exclusiveLBound=elbnd, exclusiveUBound=eubnd,          &
```

```
        computationalLBound=clbnd, computationalUBound=cubnd, rc=rc)
```

27.3.19 Get Grid stagger location information

In addition to the per DE information that can be accessed about a stagger location there is some global information that can accessed by using `ESMF_GridGet` without specifying a localDE. One of the uses of this information is to create an ESMF Array to hold data for a stagger location.
The information currently available from a stagger location is the `distgrid`. The `distgrid` gives the distgrid which describes the size and distribution of the elements in the stagger location.
The following is an example of retrieving information for localDE 0 from the corner stagger location.

```
! Get info about staggerloc
call ESMF_GridGet(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER,  &
       distgrid=staggerDistgrid, &
       rc=rc)
```

27.3.20 Create an Array at a stagger location

In order to create an Array to correspond to a Grid stagger location several pieces of information need to be obtained from both the Grid and the stagger location in the Grid.
The information that needs to be obtained from the Grid is the `distgridToGridMap` to ensure that the new Array has its dimensions are mapped correctly to the Grid. These are obtained using the `ESMF_GridGet` method.
The information that needs to be obtained from the stagger location is the distgrid that describes the size and distribution of the elements in the stagger location. This information can be obtained using the stagger location specific `ESMF_GridGet` method.
The following is an example of using information from a 2D Grid with non-arbitrary distribution to create an Array corresponding to a stagger location.

```
! Get info from Grid
call ESMF_GridGet(grid2D, distgridToGridMap=distgridToGridMap, rc=rc)

! Get info about staggerloc
call ESMF_GridGet(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, &
       distgrid=staggerDistgrid, &
       rc=rc)

! construct ArraySpec
call ESMF_ArraySpecSet(arrayspec, rank=2, typekind=ESMF_TYPEKIND_R8, rc=rc)

! Create an Array based on info from grid
array=ESMF_ArrayCreate(arrayspec=arrayspec, &
        distgrid=staggerDistgrid, distgridToArrayMap=distgridToGridMap, &
        rc=rc)
```

Creating an Array for a Grid with arbitrary distribution is different. For a 2D Grid with both dimension arbitrarily distributed, the Array dimension is 1. For a 3D Grid with two arbitrarily distributed dimensions and one undistributed dimension, the Array dimension is 2. In general, if the Array does not have any ungridded dimension, the Array dimension should be 1 plus the number of undistributed dimensions of the Grid.
The following is an example of creating an Array for a 3D Grid with 2 arbitrarily distributed dimensions such as the one defined in Section 27.3.7.

```
! Get distGrid from Grid
call ESMF_GridGet(grid3D, distgrid=distgrid, rc=rc)

! construct ArraySpec
call ESMF_ArraySpecSet(arrayspec, rank=2, typekind=ESMF_TYPEKIND_R8, rc=rc)

! Create an Array based on the presence of distributed dimensions
array=ESMF_ArrayCreate(arrayspec=arrayspec,distgrid=distgrid, rc=rc)
```

27.3.21 Create more complex Grids using DistGrid

Besides the shortcut methods for creating a Grid object such as `ESMF_GridCreateNoPeriDim()`, there is a set of methods which give the user more control over the specifics of the grid. The following describes the more general interface, using DistGrid. The basic idea is to first create an ESMF DistGrid object describing the distribution and shape of the Grid, and then to employ that to either directly create the Grid or first create Arrays and then create the Grid from those. This method gives the user maximum control over the topology and distribution of the Grid. See the DistGrid documentation in Section 31.1 for an in-depth description of its interface and use.
As an example, the following call constructs a 10x20 Grid with a lower bound of (1,2).

```
! Create DistGrid
distgrid2D = ESMF_DistGridCreate(minIndex=(/1,2/), maxIndex=(/11,22/), &
        rc=rc)

! Create Grid
grid3D=ESMF_GridCreate(distGrid=distgrid2D, rc=rc)
```

To alter which dimensions are distributed, the `distgridToGridMap` argument can be used. The `distgridToGridMap` is used to set which dimensions of the Grid are mapped to the dimensions described by `maxIndex`. In other words, it describes how the dimensions of the underlying default DistGrid are mapped to the Grid. Each entry in `distgridToGridMap` contains the Grid dimension to which the cooresponding DistGrid dimension should be mapped. The following example illustrates the creation of a Grid where the largest dimension is first. To accomplish this the two dimensions are swapped.

```
! Create DistGrid
distgrid2D = ESMF_DistGridCreate(minIndex=(/1,2/), maxIndex=(/11,22/), &
     rc=rc)

! Create Grid
grid2D=ESMF_GridCreate(distGrid=distgrid2D, distgridToGridMap=(/2,1/), &
     rc=rc)
```

27.3.22 Specify custom stagger locations

Although ESMF provides a set of predefined stagger locations (See Section 27.2.7), the user may need one outside this set. This section describes the construction of custom stagger locations.
To completely specify stagger for an arbitrary number of dimensions, we define the stagger location in terms of a set of cartesian coordinates. The cell is represented by a n-dimensional cube with sides of length 2, and the coordinate origin located at the center of the cell. The geometry of the cell is for reference purposes only, and does not literally represent the actual shape of the cell. Think of this method instead as an easy way to specify a part (e.g. center, corner, face) of a higher dimensional cell which is extensible to any number of dimensions.
To illustrate this approach, consider a 2D cell. In 2 dimensions the cell is represented by a square. An xy axis is placed at its center, with the positive x-axis oriented *East* and the positive y-axis oriented *North*. The resulting coordinate for

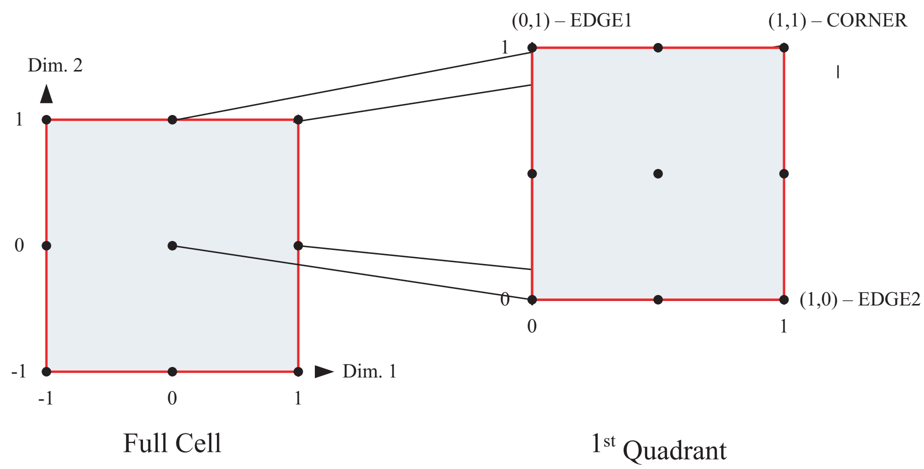


Figure 18: An example of specifying 2D stagger locations using coordinates.

the lower left corner is at $(-1, -1)$, and upper right corner at $(1, 1)$. However, because our staggers are symmetric they don't need to distinguish between the -1, and the 1, so we only need concern ourselves with the first quadrant of this cell. We only need to use the 1, and the 0, and many of the cell locations collapse together (e.g. we only need to represent one corner). See figure 18 for an illustration of these concepts.

The cell center is represented by the coordinate pair $(0, 0)$ indicating the origin. The cell corner is $+1$ in each direction, giving a coordinate pair of $(1, 1)$. The edges are each $+1$ in one dimension and 0 in the other indicating that they're even with the center in one dimension and offset in the other.
For three dimensions, the vertical component of the stagger location can be added by simply adding an additional coordinate. The three dimensional generalization of the cell center becomes $(0, 0, 0)$ and the cell corner becomes $(1, 1, 1)$. The rest of the 3D stagger locations are combinations of $+1$ offsets from the center.
To generalize this to d dimensions, to represent a d dimensional stagger location. A set of d 0 and 1 is used to specify for each dimension whether a stagger location is aligned with the cell center in that dimension (0), or offset by $+1$ in that dimension (1). Using this scheme we can represent any symmetric stagger location.
To construct a custom stagger location in ESMF the subroutine `ESMF_StaggerLocSet()` is used to specify, for each dimension, whether the stagger is located at the interior (0) or on the boundary (1) of the cell. This method allows users to construct stagger locations for which there is no predefined value. In this example, it's used to set the 4D center and 4D corner locations.

```
! Set Center
call ESMF_StaggerLocSet(staggerLoc,loc=(/0,0,0,0/),rc=rc)
call ESMF_GridAddCoord(grid4D, staggerLoc=staggerLoc, rc=rc)

! Set Corner
call ESMF_StaggerLocSet(staggerLoc,loc=(/1,1,1,1/),rc=rc)
call ESMF_GridAddCoord(grid4D, staggerLoc=staggerLoc, rc=rc)
```

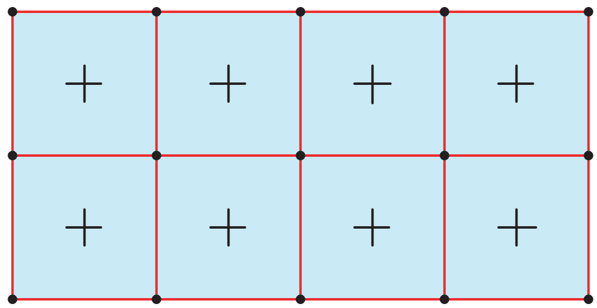


Figure 19: An example 2D Grid with cell centers and corners.

27.3.23 Specify custom stagger padding

There is an added complication with the data (e.g. coordinates) stored at stagger locations in that they can require different amounts of storage depending on the underlying Grid type.

Consider the example 2D grid in figure 19, where the dots represent the cell corners and the "+" represents the cell centers. For the corners to completely enclose the cell centers (symmetric stagger), the number of corners in each dimension needs to be one greater then the number of cell centers. In the above figure, there are two rows and three columns of cell centers. To enclose the cell centers, there must be three rows and four columns of cell corners. This is true in general for Grids without periodicity or other connections. In fact, for a symmetric stagger, given that the center location requires n x m storage, the corresponding corner location requires n+1 x m+1, and the edges, depending on the side, require n+1 x m or m+1 x n. In order to add the extra storage, a new DistGrid is created at each stagger location. This Distgrid is similar to the DistGrid used to create the Grid, but has an extra set of elements added to hold the index locations for the stagger padding. By default, when the coordinate arrays are created, one extra layer of padding is added to the index space to create symmetric staggers (i.e. the center location is surrounded). The default is to add this padding on the positive side, and to only add this padding where needed (e.g. no padding for the center, padding on both dimensions for the corner, in only one dimension for the edge in 2D.) There are two ways for the user to change these defaults.
One way is to use the `GridEdgeWidth` or `GridAlign` arguments when creating a Grid. These arguments can be used to change the default padding around the Grid cell index space. This extra padding is used by default when setting the padding for a stagger location.
The `gridEdgeLWidth` and `gridEdgeUWidth` arguments are both 1D arrays of the same size as the Grid dimension. The entries in the arrays give the extra offset from the outer boundary of the grid cell index space. The following example shows the creation of a Grid with all the extra space to hold stagger padding on the negative side of a Grid. This is the reverse of the default behavior. The resulting Grid will have an exclusive region which extends from $(-1,-1)$ to $(10,10)$, however, the cell center stagger location will still extend from $(1,1)$ to $(10,10)$.

```
grid2D=ESMF_GridCreateNoPeriDim(minIndex=(/1,1/),maxIndex=(/10,10/), &
          gridEdgeLWidth=(/1,1/), gridEdgeUWidth=(/0,0/), rc=rc)
```

To indicate how the data in a Grid's stagger locations are aligned with the cell centers, the optional `gridAlign` parameter may be used. This parameter indicates which stagger elements in a cell share the same index values as the cell center. For example, in a 2D cell, it would indicate which of the four corners has the same index value as the center. To set `gridAlign`, the values -1,+1 are used to indicate the alignment in each dimension. This parameter is mostly informational, however, if the `gridEdgeWidth` parameters are not set then its value determines where the default padding is placed. If not specified, then the default is to align all staggers to the most negative, so the padding is on the positive side. The following code illustrates creating a Grid aligned to the reverse of default (with everything to the positive side). This creates a Grid identical to that created in the previous example.

```
grid2D=ESMF_GridCreateNoPeriDim(minIndex=(/1,1/),maxIndex=(/10,10/), &
         gridAlign=(/1,1/), rc=rc)
```

The `gridEdgeWidth` and `gridAlign` arguments both allow the user to set the default padding to be used by stagger locations in a Grid. By default, stagger locations allocated in a Grid set their stagger padding based on these values. A stagger location's padding in each dimension is equal to the value of `gridEdgeWidth` (or the value implied by `gridAlign`), unless the stagger location is centered in a dimension in which case the stagger padding is 0. For example, the cell center stagger location has 0 stagger padding in all dimensions, whereas the edge stagger location lower padding is equal to `gridEdgeLWidth` and the upper padding is equal to `gridEdgeUWidth` in one dimension, but both are 0 in the other, centered, dimension. If the user wishes to set the stagger padding individually for each stagger location they may use the `staggerEdgeWidth` and `staggerAlign` arguments.
The `staggerEdgeLWidth` and `staggerEdgeUWidth` arguments are both 1D arrays of the same size as the Grid dimension. The entries in the arrays give the extra offset from the Grid cell index space for a stagger location. The following example shows the addition of two stagger locations. The corner location has no extra boundary and the center has a single layer of extra padding on the negative side and none on the positive. This is the reverse of the default behavior.

```
grid2D=ESMF_GridCreate(distgrid=distgrid2D, &
         gridEdgeLWidth=(/1,1/), gridEdgeUWidth=(/0,0/), rc=rc)

call ESMF_GridAddCoord(grid2D, &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       staggerEdgeLWidth=(/0,0/), staggerEdgeUWidth=(/0,0/), rc=rc)

call ESMF_GridAddCoord(grid2D, &
       staggerLoc=ESMF_STAGGERLOC_CENTER, &
       staggerEdgeLWidth=(/1,1/), staggerEdgeUWidth=(/0,0/), rc=rc)
```

To indicate how the data at a particular stagger location is aligned with the cell center, the optional `staggerAlign` parameter may be used. This parameter indicates which stagger elements in a cell share the same index values as the cell center. For example, in a 2D cell, it would indicate which of the four corners has the same index value as the center. To set `staggerAlign`, the values -1,+1 are used to indicate the alignment in each dimension. If a stagger location is centered in a dimension (e.g. an edge in 2D), then that dimension is ignored in the alignment. This parameter is mostly informational, however, if the `staggerEdgeWidth` parameters are not set then its value determines where the default padding is placed. If not specified, then the default is to align all staggers to the most negative, so the padding is on the positive side. The following code illustrates aligning the positive (northeast in 2D) corner with the center.

```
call ESMF_GridAddCoord(grid2D, &
       staggerLoc=ESMF_STAGGERLOC_CORNER, staggerAlign=(/1,1/), rc=rc)
```

27.3.24 Create a 2D regularly distributed rectilinear Grid from file

This example shows how to read an ESMF GridSpec Attribute Package from an XML file and use it to create a grid. The XML file contains Attribute values filled-in by the user. The standard GridSpec Attribute Package is supplied with ESMF and is defined in an XSD file, which is used to validate the XML file. See

ESMF_DIR/src/Infrastructure/Grid/etc/esmf_grid_shape_tile.xml (Attribute Package values) and

ESMF_DIR/src/Infrastructure/Grid/etc/esmf_grid.xsd (Attribute Package definition).

The following XML attributes, from the file mentioned above, specifies a two dimensional, 10x20 single-tile rectilinear grid that is regularly distributed into 2 DEs in the first dimension and 3 DEs in the second dimension, for a total of 6 DEs (2x3):

```
<?xml version="1.0"?>
<GridSpec>
  <Mosaic>
    <attribute_package convention="ESMF" purpose="General">
      <NX>10</NX>
      <NY>20</NY>
    </attribute_package>
    <RegDecompX>2</RegDecompX>
    <RegDecompY>3</RegDecompY>
  </Mosaic>
</GridSpec>
```

Read the file and create the grid,

```
! Read an XML file containing user-filled-in values for a GridSpec
! Attribute package and use it to create a grid.  The file is
! validated against an internal, ESMF-supplied XSD file defining
! the standard GridSpec Attribute package (see file pathnames above).

grid2D=ESMF_GridCreate("esmf_grid_shape_tile.xml", rc=rc)
```

then show that the minimum and maximum global indices of the Grid are (1,1) $\sim$ (11,21) (one extra default stagger pad in each dimension):

```
call ESMF_GridGet(grid2D, tile=1, minIndex=minIndex, maxIndex=maxIndex, &
  staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)
print *, "minIndex(1), minIndex(2) = ", minIndex(1), minIndex(2)
print *, "maxIndex(1), maxIndex(2) = ", maxIndex(1), maxIndex(2)
```

Get the resulting computational bounds for each local DE within the local PET, for center stagger locations:

```
call ESMF_VMGet(vm, localPet=localPet, petCount=petCount, rc=rc)
print *, "localPet = ", localPet, "petCount = ", petCount

call ESMF_GridGet(grid2D, localDECount=localDECount, rc=rc)
print *, "localDECount = ", localDECount


do i=0,localDECount-1
  call ESMF_GridGet(grid2D, localDE=i,  &
                    staggerLoc=ESMF_STAGGERLOC_CENTER, &
                    computationalLBound=clbnd, computationalUBound=cubnd, &
                    rc=rc)
    print *, "clbnd,cubnd = ", clbnd(1), ", ", clbnd(2), " ", &
                               cubnd(1), ", ", cubnd(2)
    print *, " "
enddo
```

For a 4 PET run, this will show the following (lower) $\sim$ (upper) computational bounds per DE, 6 DEs total (2x3):

```
PET 0:
  local DE 0 - (1,1)  ~ (5,7)
  local DE 1 - (1,15) ~ (5,20)
PET 1:
  local DE 0 - (6,1)  ~ (10,7)
  local DE 1 - (6,15) ~ (10,20)
PET 2:
  local DE 0 - (1,8)  ~ (5,14)
PET 3:
  local DE 0 - (6,8)  ~ (10,14)
```

For a 1 PET run, the distribution will be

```
  local DE 0 - (1,1)  ~ (5,7)
  local DE 1 - (6,1)  ~ (10,7)
  local DE 2 - (1,8)  ~ (5,14)
  local DE 3 - (6,8)  ~ (10,14)
  local DE 4 - (1,15) ~ (5,20)
  local DE 5 - (6,15) ~ (10,20)
```

The Grid and its distribution, represented graphically:

```
  -------------------------------------------------------> 2nd dim
  |
  |  (1,1)-------(1,7)(1,8)------(1,14)(1,15)-----(1,20)
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  (5,1)-------(5,7)(5,8)------(5,14)(5,15)-----(5,20)
  |  (6,1)-------(6,7)(6,8)------(6,14)(6,15)-----(6,20)
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  |               ||              ||                |
  |  (10,1)-----(10,7)(10,8)----(10,14)(10,15)---(10,20)
  |
  |
  v
 1st dim
```

Write the attributes back out to an xml file.

```
! Write an XML file
call ESMF_AttributeWrite(grid2D, 'ESMF', 'General', &
                         attwriteflag=ESMF_ATTWRITE_XML, rc=rc)
```

27.4 Restrictions and Future Work

- **7D limit.** Only grids up to 7D will be supported.
- **During the first development phase only single tile grids are supported.** In the near future, support for mosaic grids will be added. The initial implementation will be to create mosaics that contain tiles of the same grid type, e.g. rectilinear.
- **Future adaptation.** Currently Grids are created and then remain unchanged. In the future, it would be useful to provide support for the various forms of grid adaptation. This would allow the grids to dynamically change their resolution to more closely match what is needed at a particular time and postion during a computation for front tracking or adaptive meshes.
- **Future Grid generation.** This class for now only contains the basic functionality for operating on the grid. In the future methods will be added to enable the automatic generation of various types of grids.

27.5 Design and Implementation Notes

27.5.1 Grid Topology

The `ESMF_Grid` class depends upon the `ESMF_DistGrid` class for the specification of its topology. That is, when creating a Grid, first an `ESMF_DistGrid` is created to describe the appropriate index space topology. This decision was made because it seemed redundant to have a system for doing this in both classes. It also seems most appropriate for the machinary for topology creation to be located at the lowest level possible so that it can be used by other classes (e.g. the `ESMF_Array` class). Because of this, however, the authors recommend that as a natural part of the implementation of subroutines to generate standard grid shapes (e.g. `ESMF_GridGenSphere`) a set of standard topology generation subroutines be implemented (e.g. `ESMF_DistGridGenSphere`) for users who want to create a standard topology, but a custom geometry.

27.6 Class API: General Grid Methods

27.6.1 ESMF_GridAssignment(=) - Grid assignment

INTERFACE:

```
interface assignment(=)
grid1 = grid2
```

ARGUMENTS:

```
type(ESMF_Grid) :: grid1
type(ESMF_Grid) :: grid2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign grid1 as an alias to the same ESMF Grid object in memory as grid2. If grid2 is invalid, then grid1 will be equally invalid after the assignment.
The arguments are:

grid1 The `ESMF_Grid` object on the left hand side of the assignment.

grid2 The `ESMF_Grid` object on the right hand side of the assignment.

27.6.2 ESMF_GridOperator(==) - Grid equality operator

INTERFACE:

```
   interface operator(==)
     if (grid1 == grid2) then ... endif
               OR
     result = (grid1 == grid2)
```

RETURN VALUE:

```
     logical :: result
```

ARGUMENTS:

```
     type(ESMF_Grid), intent(in) :: grid1
     type(ESMF_Grid), intent(in) :: grid2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether grid1 and grid2 are valid aliases to the same ESMF Grid object in memory. For a more general comparison of two ESMF Grids, going beyond the simple alias test, the ESMF_GridMatch() function (not yet fully implemented) must be used.
The arguments are:

grid1 The `ESMF_Grid` object on the left hand side of the equality operation.

grid2 The `ESMF_Grid` object on the right hand side of the equality operation.

27.6.3 ESMF_GridOperator(/=) - Grid not equal operator

INTERFACE:

```
   interface operator(/=)
     if (grid1 /= grid2) then ... endif
               OR
     result = (grid1 /= grid2)
```

RETURN VALUE:

```
     logical :: result
```

ARGUMENTS:

```
     type(ESMF_Grid), intent(in) :: grid1
     type(ESMF_Grid), intent(in) :: grid2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether grid1 and grid2 are *not* valid aliases to the same ESMF Grid object in memory. For a more general comparison of two ESMF Grids, going beyond the simple alias test, the ESMF_GridMatch() function (not yet fully implemented) must be used.
The arguments are:

grid1 The `ESMF_Grid` object on the left hand side of the non-equality operation.

grid2 The `ESMF_Grid` object on the right hand side of the non-equality operation.

27.6.4 ESMF_GridAddCoord - Allocate coordinate arrays but don't set their values

INTERFACE:

```
   ! Private name; call using ESMF_GridAddCoord()
      subroutine ESMF_GridAddCoordNoValues(grid, staggerloc,  &
        staggerEdgeLWidth, staggerEdgeUWidth, staggerAlign, &
        staggerLBound,rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)            :: grid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
       integer,                intent(in),  optional :: staggerEdgeLWidth(:)
       integer,                intent(in),  optional :: staggerEdgeUWidth(:)
       integer,                intent(in),  optional :: staggerAlign(:)
       integer,                intent(in),  optional :: staggerLBound(:)
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

When a Grid is created all of its potential stagger locations can hold coordinate data, but none of them have storage allocated. This call allocates coordinate storage (creates internal ESMF_Arrays and associated memory) for a particular stagger location. Note that this call doesn't assign any values to the storage, it only allocates it. The remaining options `staggerEdgeLWidth`, etc. allow the user to adjust the padding on the coordinate arrays.
The arguments are:

grid Grid to allocate coordinate storage in.

[staggerloc] The stagger location to add. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

[staggerEdgeLWidth] This array should be the same dimCount as the grid. It specifies the lower corner of the stagger region with respect to the lower corner of the exclusive region.

[staggerEdgeUWidth] This array should be the same dimCount as the grid. It specifies the upper corner of the stagger region with respect to the upper corner of the exclusive region.

[staggerAlign] This array is of size grid dimCount. For this stagger location, it specifies which element has the same index value as the center. For example, for a 2D cell with corner stagger it specifies which of the 4 corners has the same index as the center. If this is set and either staggerEdgeUWidth or staggerEdgeLWidth is not, this determines the default array padding for a stagger. If not set, then this defaults to all negative. (e.g. The most negative part of the stagger in a cell is aligned with the center and the padding is all on the postive side.)

[staggerLBound] Specifies the lower index range of the memory of every DE in this staggerloc in this Grid. Only used when Grid indexflag is `ESMF_INDEX_USER`.

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

27.6.5 ESMF_GridAddItem - Allocate item array but don't set their values

INTERFACE:

```
   ! Private name; call using ESMF_GridAddItem()
      subroutine ESMF_GridAddItemNoValues(grid, itemflag,  &
        staggerloc, itemTypeKind, staggerEdgeLWidth, staggerEdgeUWidth, &
        staggerAlign, staggerLBound,rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),            intent(in)             :: grid
       type (ESMF_GridItem_Flag), intent(in)             :: itemflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type (ESMF_StaggerLoc)  ,  intent(in),  optional  :: staggerloc
       type (ESMF_TypeKind_Flag), intent(in),  optional  :: itemTypeKind
       integer,                   intent(in),  optional  :: staggerEdgeLWidth(:)
       integer,                   intent(in),  optional  :: staggerEdgeUWidth(:)
       integer,                   intent(in),  optional  :: staggerAlign(:)
       integer,                   intent(in),  optional  :: staggerLBound(:)
       integer,                   intent(out), optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

When a Grid is created all of its potential stagger locations can hold item data, but none of them have storage allocated. This call allocates item storage (creates an internal ESMF_Array and associated memory) for a particular stagger location. Note that this call doesn't assign any values to the storage, it only allocates it. The remaining options `staggerEdgeLWidth`, etc. allow the user to adjust the padding on the item array.
The arguments are:

grid Grid to allocate coordinate storage in.

itemflag The grid item to add. Please see Section 27.2.3 for a list of valid items.

[staggerloc] The stagger location to add. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

[itemTypeKind] The typekind of the item to add.

[staggerEdgeLWidth] This array should be the same dimCount as the grid. It specifies the lower corner of the stagger region with respect to the lower corner of the exclusive region.

[staggerEdgeUWidth] This array should be the same dimCount as the grid. It specifies the upper corner of the stagger region with respect to the upper corner of the exclusive region.

[staggerAlign] This array is of size grid dimCount. For this stagger location, it specifies which element has the same index value as the center. For example, for a 2D cell with corner stagger it specifies which of the 4 corners has the same index as the center. If this is set and either staggerEdgeUWidth or staggerEdgeLWidth is not, this determines the default array padding for a stagger. If not set, then this defaults to all negative. (e.g. The most negative part of the stagger in a cell is aligned with the center and the padding is all on the postive side.)

[staggerLBound] Specifies the lower index range of the memory of every DE in this staggerloc in this Grid. Only used when Grid indexflag is `ESMF_INDEX_USER`.

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

27.6.6 ESMF_GridCreate - Create a copy of a Grid with a new DistGrid

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate()
       function ESMF_GridCreateCopyFromNewDG(grid, distgrid, &
         name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreateCopyFromNewDG
```

ARGUMENTS:

```
        type(ESMF_Grid),       intent(in)              :: grid
        type(ESMF_DistGrid),   intent(in)              :: distgrid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        character (len=*),     intent(in),   optional  :: name
        integer,               intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This call allows the user to copy of an existing ESMF Grid, but with a new distribution. All internal data from the old Grid (coords, items) is redistributed to the new Grid.
The arguments are:

grid `ESMF_Grid` to copy.

distgrid `ESMF_DistGrid` object which describes how the Grid is decomposed and distributed over DEs.

[name] Name of the new Grid. If not specified, a new unique name will be created for the Grid.

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

27.6.7 ESMF_GridCreate - Create a copy of a Grid with a different regular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate()
       function ESMF_GridCreateCopyFromReg(grid, &
         regDecomp, decompFlag, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreateCopyFromReg
```

ARGUMENTS:

```
        type(ESMF_Grid),        intent(in)              :: grid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,                intent(in),   optional  :: regDecomp(:)
        type(ESMF_Decomp_Flag), intent(in),   optional  :: decompflag(:)
        character (len=*),      intent(in),   optional  :: name
        integer,                intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method creates a copy of an existing Grid, the new Grid is regularly distributed (see Figure 13). To specify the new distribution, the user passes in an array (regDecomp) specifying the number of DEs to divide each dimension into. The array decompFlag indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.
The arguments are:

grid ESMF_Grid to copy.

[regDecomp] List that has the same number of elements as maxIndex. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is ESMF_DECOMP_BALANCED in all dimensions. Please see Section 9.13 for a full description of the possible options.

[name] Name of the new Grid. If not specified, a new unique name will be created for the Grid.

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

27.6.8 ESMF_GridCreate - Create a Grid with user set edge connections and an irregular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate()
       function ESMF_GridCreateEdgeConnI(minIndex,         &
         countsPerDEDim1,countsPerDeDim2,                  &
         countsPerDEDim3,                                  &
         connflagDim1, connflagDim2, connflagDim3,                     &
         coordSys, coordTypeKind,                          &
         coordDep1, coordDep2, coordDep3,                  &
         gridEdgeLWidth, gridEdgeUWidth, gridAlign,        &
         gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreateEdgeConnI
```

ARGUMENTS:

```
        integer,                    intent(in),  optional :: minIndex(:)
        integer,                    intent(in)            :: countsPerDEDim1(:)
        integer,                    intent(in)            :: countsPerDEDim2(:)
  -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,                    intent(in),  optional :: countsPerDEDim3(:)
        type(ESMF_GridConn_Flag),   intent(in),  optional :: connflagDim1(:)
        type(ESMF_GridConn_Flag),   intent(in),  optional :: connflagDim2(:)
        type(ESMF_GridConn_Flag),   intent(in),  optional :: connflagDim3(:)
        type(ESMF_CoordSys_Flag),   intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag),   intent(in),  optional :: coordTypeKind
        integer,                    intent(in),  optional :: coordDep1(:)
        integer,                    intent(in),  optional :: coordDep2(:)
        integer,                    intent(in),  optional :: coordDep3(:)
```

```
integer,                    intent(in),  optional :: gridEdgeLWidth(:)
integer,                    intent(in),  optional :: gridEdgeUWidth(:)
integer,                    intent(in),  optional :: gridAlign(:)
integer,                    intent(in),  optional :: gridMemLBound(:)
type(ESMF_Index_Flag),      intent(in),  optional :: indexflag
integer,                    intent(in),  optional :: petMap(:,:,:)
character (len=*),          intent(in),  optional :: name
integer,                    intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, irregularly distributed grid (see Figure 13) without a periodic dimension. To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim arrays that are specified.
Section 27.3.4 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

countsPerDEDim1 This arrays specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).

countsPerDEDim2 This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).

[countsPerDEDim3] This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.

[connflagDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connflagDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connflagDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size size(countsPerDEDim1) x size(countsPerDEDim2) x size(countsPerDEDim3). If countsPerDEDim3 isn't present, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.9 ESMF_GridCreate - Create a Grid with user set edge connections and a regular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate()
       function ESMF_GridCreateEdgeConnR(regDecomp, decompFlag, &
         minIndex, maxIndex,                                    &
         connflagDim1, connflagDim2, connflagDim3,                       &
         coordSys, coordTypeKind,                            &
         coordDep1, coordDep2, coordDep3,                    &
         gridEdgeLWidth, gridEdgeUWidth, gridAlign,          &
         gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreateEdgeConnR
```

ARGUMENTS:

```
        integer,                  intent(in),  optional :: regDecomp(:)
        type(ESMF_Decomp_Flag),   intent(in),  optional :: decompflag(:)
        integer,                  intent(in),  optional :: minIndex(:)
        integer,                  intent(in)            :: maxIndex(:)
```

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_GridConn_Flag), intent(in),  optional :: connflagDim1(:)
       type(ESMF_GridConn_Flag), intent(in),  optional :: connflagDim2(:)
       type(ESMF_GridConn_Flag), intent(in),  optional :: connflagDim3(:)
       type(ESMF_CoordSys_Flag), intent(in),  optional :: coordSys
       type(ESMF_TypeKind_Flag), intent(in),  optional :: coordTypeKind
       integer,                  intent(in),  optional :: coordDep1(:)
       integer,                  intent(in),  optional :: coordDep2(:)
       integer,                  intent(in),  optional :: coordDep3(:)
       integer,                  intent(in),  optional :: gridEdgeLWidth(:)
       integer,                  intent(in),  optional :: gridEdgeUWidth(:)
       integer,                  intent(in),  optional :: gridAlign(:)
       integer,                  intent(in),  optional :: gridMemLBound(:)
       type(ESMF_Index_Flag),    intent(in),  optional :: indexflag
       integer,                  intent(in),  optional :: petMap(:,:,:)
       character (len=*),        intent(in),  optional :: name
       integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, regularly distributed grid (see Figure 13). To specify the distribution, the user passes in an array (`regDecomp`) specifying the number of DEs to divide each dimension into. The array `decompFlag` indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.
The arguments are:

[regDecomp] List that has the same number of elements as `maxIndex`. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_BALANCED` in all dimensions. Please see Section 9.13 for a full description of the possible options.

[minIndex] The bottom extent of the grid array. If not given then the value defaults to /1,1,1,.../.

maxIndex The upper extent of the grid array.

[connflagDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connflagDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connflagDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.10 ESMF_GridCreate - Create a Grid with user set edge connections and an arbitrary distribution

INTERFACE:

```
  ! Private name; call using ESMF_GridCreate()
      function ESMF_GridCreateEdgeConnA(minIndex, maxIndex,  &
        arbIndexCount, arbIndexList,                         &
        connflagDim1, connflagDim2, connflagDim3,                     &
        coordSys, coordTypeKind,                          &
        coordDep1, coordDep2, coordDep3,                  &
        distDim, name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreateEdgeConnA
```

ARGUMENTS:

```
       integer,                  intent(in),  optional :: minIndex(:)
       integer,                  intent(in)            :: maxIndex(:)
       integer,                  intent(in)            :: arbIndexCount
       integer,                  intent(in)            :: arbIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_GridConn_Flag), intent(in),  optional :: connflagDim1(:)
       type(ESMF_GridConn_Flag), intent(in),  optional :: connflagDim2(:)
       type(ESMF_GridConn_Flag), intent(in),  optional :: connflagDim3(:)
       type(ESMF_CoordSys_Flag), intent(in),  optional :: coordSys
       type(ESMF_TypeKind_Flag), intent(in),  optional :: coordTypeKind
       integer,                  intent(in),  optional :: coordDep1(:)
       integer,                  intent(in),  optional :: coordDep2(:)
       integer,                  intent(in),  optional :: coordDep3(:)
       integer,                  intent(in),  optional :: distDim(:)
       character (len=*),        intent(in),  optional :: name
       integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, arbitrarily distributed grid (see Figure 13). To specify the arbitrary distribution, the user passes in an 2D array of local indices, where the first dimension is the number of local grid cells specified by `localArbIndexCount` and the second dimension is the number of distributed dimensions.
`distDim` specifies which grid dimensions are arbitrarily distributed. The size of `distDim` has to agree with the size of the second dimension of `localArbIndex`.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

[maxIndex] The upper extend of the grid index ranges.

arbIndexCount The number of grid cells in the local DE. It is okay to have 0 grid cell in a local DE.

[arbIndexList] This 2D array specifies the indices of the PET LOCAL grid cells. The dimensions should be arbIndexCount * number of Distributed grid dimensions where arbIndexCount is the input argument specified below

[connflagDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connflagDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connflagDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is

/ESMF_DIM_ARB/ if the first dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=1) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep2] The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=2) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep3] The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=3) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[distDim] This array specifies which dimensions are arbitrarily distributed. The size of the array specifies the total distributed dimensions. if not specified, defaults is all dimensions will be arbitrarily distributed. The size has to agree with the size of the second dimension of `localArbIndex`.

[name] `ESMF_Grid` name.

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

27.6.11 ESMF_GridCreate - Create a Grid from a DistGrid

INTERFACE:

```
! Private name; call using ESMF_GridCreate()
    function ESMF_GridCreateFrmDistGrid(distgrid, &
      distgridToGridMap, &
      coordSys, coordTypeKind, coordDimCount, coordDimMap, &
      gridEdgeLWidth, gridEdgeUWidth, gridAlign, &
      gridMemLBound, indexflag, name, rc)
```

RETURN VALUE:

```
    type(ESMF_Grid) :: ESMF_GridCreateFrmDistGrid
```

ARGUMENTS:

```
     type(ESMF_DistGrid),      intent(in)             :: distgrid
     integer,                  intent(in),  optional  :: distgridToGridMap(:)
     type(ESMF_CoordSys_Flag), intent(in),  optional  :: coordSys
     type(ESMF_TypeKind_Flag), intent(in),  optional  :: coordTypeKind
     integer,                  intent(in),  optional  :: coordDimCount(:)
     integer,                  intent(in),  optional  :: coordDimMap(:,:)
     integer,                  intent(in),  optional  :: gridEdgeLWidth(:)
     integer,                  intent(in),  optional  :: gridEdgeUWidth(:)
     integer,                  intent(in),  optional  :: gridAlign(:)
     integer,                  intent(in),  optional  :: gridMemLBound(:)
     type(ESMF_Index_Flag),    intent(in),  optional  :: indexflag
     character (len=*),        intent(in),  optional  :: name
     integer,                  intent(out), optional  :: rc
```

DESCRIPTION:

This is the most general form of creation for an ESMF_Grid object. It allows the user to fully specify the topology and index space using the DistGrid methods and then build a grid out of the resulting DistGrid. Note that since the Grid created by this call uses distgrid as a description of its index space, the resulting Grid will have exactly the same number of dimensions (i.e. the same dimCount) as distgrid. The distgridToGridMap argument specifies how the Grid dimensions are mapped to the distgrid. The coordDimCount and coordDimMap arguments allow the user to specify how the coordinate arrays should map to the grid dimensions. (Note, though, that creating a grid does not allocate coordinate storage. A method such as ESMF_GridAddCoord() must be called before adding coordinate values.)
The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs.

[distgridToGridMap] List that has dimCount elements. The elements map each dimension of distgrid to a dimension in the grid. (i.e. the values should range from 1 to dimCount). If not specified, the default is to map all of distgrid's dimensions against the dimensions of the grid in sequence.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_CART.

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

[coordDimCount] List that has dimCount elements. Gives the dimension of each component (e.g. x) array. This is to allow factorization of the coordinate arrays. If not specified all arrays are the same size as the grid.

[coordDimMap] 2D list of size dimCount x dimCount. This array describes the map of each component array's dimensions onto the grids dimensions. Each entry coordDimMap(i,j) tells which grid dimension component i's, jth dimension maps to. Note that if j is bigger than coordDimCount(i) it is ignored. The default for each row i is coordDimMap(i,:)=(1,2,3,4,...).

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is ESMF_INDEX_USER. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[name] ESMF_Grid name.

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

27.6.12 ESMF_GridCreate - Create a Arbitrary Grid from a DistGrid

INTERFACE:

```
  ! Private name; call using ESMF_GridCreate()
      function ESMF_GridCreateFrmDistGridArb(distgrid, &
        indexArray, distDim, &
        coordSys, coordTypeKind, coordDimCount, coordDimMap, &
        name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreateFrmDistGridArb
```

ARGUMENTS:

```
       type(ESMF_DistGrid),      intent(in)              :: distgrid
       integer,                  intent(in)              :: indexArray(:,:)
       integer,                  intent(in),   optional  :: distDim(:)
       type(ESMF_CoordSys_Flag), intent(in),   optional  :: coordSys
       type(ESMF_TypeKind_Flag), intent(in),   optional  :: coordTypeKind
       integer,                  intent(in),   optional  :: coordDimCount(:)
       integer,                  intent(in),   optional  :: coordDimMap(:,:)
       character (len=*),        intent(in),   optional  :: name
       integer,                  intent(out),  optional  :: rc
```

DESCRIPTION:

This is the lower level function to create an arbitrailiy distributed `ESMF_Grid` object. It allows the user to fully specify the topology and index space (of the distributed dimensions) using the DistGrid methods and then build a grid out of the resulting `distgrid`. The `indexArray(2,dimCount)`, argument is required to specifies the topology of the grid.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs.

[indexArray] The minIndex and maxIndex array of size `2` x `dimCount` `indexArray(1,:)` is the minIndex and `indexArray(2,:)` is the maxIndex

[distDim] This array specifies which dimensions are arbitrarily distributed. The size of the array specifies the total distributed dimensions. if not specified, the default is that all dimensions will be arbitrarily distributed.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_CART.

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

[coordDimCount] List that has dimCount elements. Gives the dimension of each component (e.g. x) array. This is to allow factorization of the coordinate arrays. If not specified each component is assumed to be size 1. Note, the default value is different from the same argument for a non-arbitrarily distributed grid.

[coordDimMap] 2D list of size dimCount x dimCount. This array describes the map of each coordinate array's dimensions onto the grids dimensions. `coordDimMap(i,j)` is the grid dimension of the jth dimension of the i'th coordinate array. If not specified, the default value of `coordDimMap(i,1)` is /ESMF_DIM_ARB/ if the ith dimension of the grid is arbitrarily distributed, or `i` if the ith dimension is not distributed. Note that if j is bigger than `coordDimCount(i)` then it's ignored. Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[name] `ESMF_Grid` name.

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

27.6.13 ESMF_GridCreate - Create a Grid from a file

INTERFACE:

```
! Private name; call using ESMF_GridCreate()
   function ESMF_GridCreateFrmFile(fileName, &
     convention, purpose, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreateFrmFile
```

ARGUMENTS:

```
        character (len=*), intent(in)            :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        character (len=*), intent(in),  optional :: convention
        character (len=*), intent(in),  optional :: purpose
        integer,           intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Grid` object from specifications in a file containing an ESMF GridSpec Attribute package in XML format. Currently limited to creating a 2D regularly distributed rectilinear Grid; in the future more dimensions, grid types and distributions will be supported. See Section 27.3.24 for an example, as well as the accompanying file ESMF_DIR/src/Infrastructure/Grid/etc/esmf_grid_shape_tile.xml.
Requires the third party Xerces C++ XML Parser library to be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, Xerces" and the website http://xerces.apache.org/xerces-c.
The arguments are:

fileName The name of the XML file to be read, containing ESMF GridSpec Attributes.

[convention] The convention of a grid Attribute package. [CURRENTLY NOT IMPLEMENTED]

[purpose] The purpose of a grid Attribute package. [CURRENTLY NOT IMPLEMENTED]

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors. Equals `ESMF_RC_LIB_NOT_PRESENT` if Xerces is not present.

27.6.14 ESMF_GridCreate - Create a Grid from a SCRIP grid file and a DistGrid

INTERFACE:

```
! Private name; call using ESMF_GridCreate()
function ESMF_GridCreateFrmScripDistGrd(distgrid, filename, &
  rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreateFrmScripDistGrd
```

ARGUMENTS:

```
     type(ESMF_DistGrid), intent(in)             :: distgrid
     character(len=*),    intent(in)             :: filename
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(out), optional  :: rc
```

DESCRIPTION:

This function creates a `ESMF_Grid` object using the grid definition from a SCRIP grid file. The grid distribution is defined by a DistGrid object. The distrgrid has to match the grid defined in the file. This means the distgrid should consist of one 2D tile with the same size in each dimension as the grid in the file. The grid defined in the file has to be a 2D logically rectangular grid (i.e. `grid_rank` in the file needs to be 2).
This call is *collective* across the current VM.
The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs.

[filename] The SCRIP Grid filename.

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

27.6.15 ESMF_GridCreate - Create a Grid from a SCRIP grid file with a regular distribution

INTERFACE:

```
! Private name; call using ESMF_GridCreate()
function ESMF_GridCreateFrmScripReg(filename, regDecomp, &
  decompflag, isSphere, addCornerStagger, rc)
```

RETURN VALUE:

```
type(ESMF_Grid) :: ESMF_GridCreateFrmScripReg
```

ARGUMENTS:

```
    character(len=*),       intent(in)             :: filename
    integer,                intent(in)             :: regDecomp(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Decomp_Flag), intent(in),  optional  :: decompflag(:)
    logical,                intent(in),  optional  :: isSphere
    logical,                intent(in),  optional  :: addCornerStagger
    integer,                intent(out), optional  :: rc
```

DESCRIPTION:

This function creates a `ESMF_Grid` object using the grid definition from a SCRIP grid file. To specify the distribution, the user passes in an array (`regDecomp`) specifying the number of DEs to divide each dimension into. The array `decompflag` indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible. The grid defined in the file has to be a 2D logically rectangular grid (i.e. `grid_rank` in the file needs to be 2).
This call is *collective* across the current VM.
The arguments are:

[filename] The SCRIP Grid filename.

[regDecomp] A 2 element array specifying how the grid is decomposed. Each entry is the number of decounts for that dimension.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_BALANCED` in all dimensions. Please see Section 9.13 for a full description of the possible options.

[isSphere] If .true. is a spherical grid, if .false. is regional. Defaults to .true.

[addCornerStagger] Uses the information in the SCRIP file to add the Corner stagger to the Grid. If not specified, defaults to false.

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

27.6.16 ESMF_GridCreate1PeriDim - Create a Grid with one periodic dim and an irregular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate1PeriDim()
       function ESMF_GridCreate1PeriDimI(minIndex,         &
         countsPerDEDim1,countsPerDeDim2,                  &
         countsPerDEDim3,                                  &
         polekindflag, periodicDim, poleDim,                   &
         coordSys, coordTypeKind,                          &
         coordDep1, coordDep2, coordDep3,                  &
         gridEdgeLWidth, gridEdgeUWidth, gridAlign,        &
         gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreate1PeriDimI
```

ARGUMENTS:

```
        integer,                   intent(in),  optional :: minIndex(:)
        integer,                   intent(in)            :: countsPerDEDim1(:)
        integer,                   intent(in)            :: countsPerDEDim2(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,                   intent(in),  optional :: countsPerDEDim3(:)
        type(ESMF_PoleKind_Flag),  intent(in),  optional :: polekindflag(2)
        integer,                   intent(in),  optional :: periodicDim
        integer,                   intent(in),  optional :: poleDim
        type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
        integer,                   intent(in),  optional :: coordDep1(:)
        integer,                   intent(in),  optional :: coordDep2(:)
        integer,                   intent(in),  optional :: coordDep3(:)
        integer,                   intent(in),  optional :: gridEdgeLWidth(:)
        integer,                   intent(in),  optional :: gridEdgeUWidth(:)
        integer,                   intent(in),  optional :: gridAlign(:)
        integer,                   intent(in),  optional :: gridMemLBound(:)
        type(ESMF_Index_Flag),     intent(in),  optional :: indexflag
        integer,                   intent(in),  optional :: petMap(:,:,:)
        character (len=*),         intent(in),  optional :: name
        integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, irregularly distributed grid (see Figure 13) without a periodic dimension. To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the

index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim arrays that are specified.
Section 27.3.4 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

countsPerDEDim1 This arrays specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).

countsPerDEDim2 This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).

[countsPerDEDim3] This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.

[polekindflag] Two item array which specifies the type of connection which occurs at the pole. polekindflag(1) the connection that occurs at the minimum end of the index dimension. polekindflag(2) the connection that occurs at the maximum end of the index dimension. Please see Section 27.2.6 for a full list of options. If not specified, the default is `ESMF_POLETYPE_MONOPOLE` for both.

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

[poleDim] The dimension at who's ends the poles are located. If not specified defaults to 2.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size size(countsPerDEDim1) x size(countsPerDEDim2) x size(countsPerDEDim3). If countsPerDEDim3 isn't present, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.17 ESMF_GridCreate1PeriDim - Create a Grid with one periodic dim and a regular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate1PeriDim()
       function ESMF_GridCreate1PeriDimR(regDecomp, decompFlag, &
         minIndex, maxIndex,                                    &
         polekindflag, periodicDim, poleDim,                        &
         coordSys, coordTypeKind,                               &
         coordDep1, coordDep2, coordDep3,                       &
         gridEdgeLWidth, gridEdgeUWidth, gridAlign,             &
         gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreate1PeriDimR
```

ARGUMENTS:

```
        integer,                   intent(in),  optional :: regDecomp(:)
        type(ESMF_Decomp_Flag),    intent(in),  optional :: decompflag(:)
        integer,                   intent(in),  optional :: minIndex(:)
        integer,                   intent(in)            :: maxIndex(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_PoleKind_Flag),  intent(in),  optional :: polekindflag(2)
        integer,                   intent(in),  optional :: periodicDim
        integer,                   intent(in),  optional :: poleDim
        type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
        integer,                   intent(in),  optional :: coordDep1(:)
        integer,                   intent(in),  optional :: coordDep2(:)
        integer,                   intent(in),  optional :: coordDep3(:)
        integer,                   intent(in),  optional :: gridEdgeLWidth(:)
        integer,                   intent(in),  optional :: gridEdgeUWidth(:)
        integer,                   intent(in),  optional :: gridAlign(:)
        integer,                   intent(in),  optional :: gridMemLBound(:)
        type(ESMF_Index_Flag),     intent(in),  optional :: indexflag
        integer,                   intent(in),  optional :: petMap(:,:,:)
        character (len=*),         intent(in),  optional :: name
        integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, regularly distributed grid (see Figure 13). To specify the distribution, the user passes in an array (regDecomp) specifying the number of DEs to divide each dimension into. The array decompFlag indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.
The arguments are:

[regDecomp] List that has the same number of elements as maxIndex. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is ESMF_DECOMP_BALANCED in all dimensions. Please see Section 9.13 for a full description of the possible options.

[minIndex] The bottom extent of the grid array. If not given then the value defaults to /1,1,1,.../.

maxIndex The upper extent of the grid array.

[polekindflag] Two item array which specifies the type of connection which occurs at the pole. polekindflag(1) the connection that occurs at the minimum end of the index dimension. polekindflag(2) the connection that occurs at the maximum end of the index dimension. Please see Section 27.2.6 for a full list of options. If not specified, the default is ESMF_POLETYPE_MONOPOLE for both.

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

[poleDim] The dimension at who's ends the poles are located. If not specified defaults to 2.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by coordsPerDEDim1,2,3. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the gridEdgeWidths are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.18 ESMF_GridCreate1PeriDim - Create a Grid with one periodic dim and an arbitrary distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreate1PeriDim()
       function ESMF_GridCreate1PeriDimA(minIndex, maxIndex,  &
         arbIndexCount, arbIndexList,                         &
         polekindflag, periodicDim, poleDim,                      &
         coordSys, coordTypeKind,                             &
         coordDep1, coordDep2, coordDep3,                     &
         distDim, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreate1PeriDimA
```

ARGUMENTS:

```
        integer,                   intent(in),  optional :: minIndex(:)
        integer,                   intent(in)            :: maxIndex(:)
        integer,                   intent(in)            :: arbIndexCount
        integer,                   intent(in)            :: arbIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_PoleKind_Flag),  intent(in),  optional :: polekindflag(2)
        integer,                   intent(in),  optional :: periodicDim
        integer,                   intent(in),  optional :: poleDim
        type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
        integer,                   intent(in),  optional :: coordDep1(:)
        integer,                   intent(in),  optional :: coordDep2(:)
        integer,                   intent(in),  optional :: coordDep3(:)
        integer,                   intent(in),  optional :: distDim(:)
        character (len=*),         intent(in),  optional :: name
        integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, arbitrarily distributed grid (see Figure 13). To specify the arbitrary distribution, the user passes in an 2D array of local indices, where the first dimension is the number of local grid cells specified by `localArbIndexCount` and the second dimension is the number of distributed dimensions.
`distDim` specifies which grid dimensions are arbitrarily distributed. The size of `distDim` has to agree with the size of the second dimension of `localArbIndex`.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

[maxIndex] The upper extend of the grid index ranges.

arbIndexCount The number of grid cells in the local DE. It is okay to have 0 grid cell in a local DE.

[arbIndexList] This 2D array specifies the indices of the PET LOCAL grid cells. The dimensions should be arbIndexCount * number of Distributed grid dimensions where arbIndexCount is the input argument specified below

[polekindflag] Two item array which specifies the type of connection which occurs at the pole. polekindflag(1) the connection that occurs at the minimum end of the index dimension. polekindflag(2) the connection that occurs at the maximum end of the index dimension. Please see Section 27.2.6 for a full list of options. If not specified, the default is `ESMF_POLETYPE_MONOPOLE` for both.

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

[poleDim] The dimension at who's ends the poles are located. If not specified defaults to 2.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if the first dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=1) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep2] The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=2) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep3] The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=3) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[distDim] This array specifies which dimensions are arbitrarily distributed. The size of the array specifies the total distributed dimensions. if not specified, defaults is all dimensions will be arbitrarily distributed. The size has to agree with the size of the second dimension of `localArbIndex`.

[name] `ESMF_Grid` name.

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

27.6.19 ESMF_GridCreate2PeriDim - Create a Grid with two periodic dims and an irregular distribution

INTERFACE:

```
  ! Private name; call using ESMF_GridCreate2PeriDim()
      function ESMF_GridCreate2PeriDimI(minIndex,         &
        countsPerDEDim1,countsPerDeDim2,                  &
        countsPerDEDim3,                                  &
        coordSys, coordTypeKind,                          &
        coordDep1, coordDep2, coordDep3,                  &
        gridEdgeLWidth, gridEdgeUWidth, gridAlign,        &
        gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreate2PeriDimI
```

ARGUMENTS:

```
       integer,                   intent(in),  optional :: minIndex(:)
       integer,                   intent(in)            :: countsPerDEDim1(:)
       integer,                   intent(in)            :: countsPerDEDim2(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,                   intent(in),  optional :: countsPerDEDim3(:)
       type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
       type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
       integer,                   intent(in),  optional :: coordDep1(:)
       integer,                   intent(in),  optional :: coordDep2(:)
       integer,                   intent(in),  optional :: coordDep3(:)
       integer,                   intent(in),  optional :: gridEdgeLWidth(:)
       integer,                   intent(in),  optional :: gridEdgeUWidth(:)
       integer,                   intent(in),  optional :: gridAlign(:)
       integer,                   intent(in),  optional :: gridMemLBound(:)
       type(ESMF_Index_Flag),     intent(in),  optional :: indexflag
       integer,                   intent(in),  optional :: petMap(:,:,:)
       character (len=*),         intent(in),  optional :: name
       integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, irregularly distributed grid (see Figure 13) without a periodic dimension. To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim arrays that are specified.
Section 27.3.4 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

countsPerDEDim1 This arrays specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).

countsPerDEDim2 This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).

[countsPerDEDim3] This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size size(countsPerDEDim1) x size(countsPerDEDim2) x size(countsPerDEDim3). If countsPerDEDim3 isn't present, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.20 ESMF_GridCreate2PeriDim - Create a Grid with two periodic dims and a regular distribution

INTERFACE:

```
! Private name; call using ESMF_GridCreate2PeriDim()
    function ESMF_GridCreate2PeriDimR(regDecomp, decompFlag, &
      minIndex, maxIndex,                                    &
      coordSys, coordTypeKind,                               &
      coordDep1, coordDep2, coordDep3,                       &
      gridEdgeLWidth, gridEdgeUWidth, gridAlign,             &
      gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
    type(ESMF_Grid) :: ESMF_GridCreate2PeriDimR
```

ARGUMENTS:

```
      integer,                   intent(in),  optional :: regDecomp(:)
      type(ESMF_Decomp_Flag),    intent(in),  optional :: decompflag(:)
      integer,                   intent(in),  optional :: minIndex(:)
      integer,                   intent(in)            :: maxIndex(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
      type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
      integer,                   intent(in),  optional :: coordDep1(:)
      integer,                   intent(in),  optional :: coordDep2(:)
      integer,                   intent(in),  optional :: coordDep3(:)
      integer,                   intent(in),  optional :: gridEdgeLWidth(:)
      integer,                   intent(in),  optional :: gridEdgeUWidth(:)
      integer,                   intent(in),  optional :: gridAlign(:)
      integer,                   intent(in),  optional :: gridMemLBound(:)
      type(ESMF_Index_Flag),     intent(in),  optional :: indexflag
      integer,                   intent(in),  optional :: petMap(:,:,:)
      character (len=*),         intent(in),  optional :: name
      integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, regularly distributed grid (see Figure 13). To specify the distribution, the user passes in an array (`regDecomp`) specifying the number of DEs to divide each dimension into. The array `decompFlag` indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.
The arguments are:

[regDecomp] List that has the same number of elements as `maxIndex`. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_BALANCED` in all dimensions. Please see Section 9.13 for a full description of the possible options.

[minIndex] The bottom extent of the grid array. If not given then the value defaults to /1,1,1,.../.

maxIndex The upper extent of the grid array.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.21 ESMF_GridCreate2PeriDim - Create a Grid with two periodic dims and an arbitrary distribution

INTERFACE:

```
  ! Private name; call using ESMF_GridCreate2PeriDim()
      function ESMF_GridCreate2PeriDimA(minIndex, maxIndex, &
        arbIndexCount, arbIndexList,                     &
        coordSys, coordTypeKind,                         &
        coordDep1, coordDep2, coordDep3,                 &
        distDim, name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreate2PeriDimA
```

ARGUMENTS:

```
       integer,                  intent(in),  optional :: minIndex(:)
       integer,                  intent(in)            :: maxIndex(:)
       integer,                  intent(in)            :: arbIndexCount
       integer,                  intent(in)            :: arbIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
       type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
       integer,                  intent(in),  optional :: coordDep1(:)
       integer,                  intent(in),  optional :: coordDep2(:)
       integer,                  intent(in),  optional :: coordDep3(:)
       integer,                  intent(in),  optional :: distDim(:)
       character (len=*),        intent(in),  optional :: name
       integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, arbitrarily distributed grid (see Figure 13). To specify the arbitrary distribution, the user passes in an 2D array of local indices, where the first dimension is the number of local grid cells specified by `localArbIndexCount` and the second dimension is the number of distributed dimensions.
`distDim` specifies which grid dimensions are arbitrarily distributed. The size of `distDim` has to agree with the size of the second dimension of `localArbIndex`.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

[maxIndex] The upper extend of the grid index ranges.

arbIndexCount The number of grid cells in the local DE. It is okay to have 0 grid cell in a local DE.

[arbIndexList] This 2D array specifies the indices of the PET LOCAL grid cells. The dimensions should be arbIndexCount * number of Distributed grid dimensions where arbIndexCount is the input argument specified below

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if the first dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=1) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep2] The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=2) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep3] The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=3) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[distDim] This array specifies which dimensions are arbitrarily distributed. The size of the array specifies the total distributed dimensions. if not specified, defaults is all dimensions will be arbitrarily distributed. The size has to agree with the size of the second dimension of `localArbIndex`.

[name] `ESMF_Grid` name.

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

27.6.22 ESMF_GridCreateNoPeriDim - Create a Grid with no periodic dim and an irregular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridCreateNoPeriDim()
       function ESMF_GridCreateNoPeriDimI(minIndex,         &
         countsPerDEDim1,countsPerDeDim2,                   &
         countsPerDEDim3,                                   &
         coordSys, coordTypeKind,                           &
         coordDep1, coordDep2, coordDep3,                   &
         gridEdgeLWidth, gridEdgeUWidth, gridAlign,         &
         gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
       type(ESMF_Grid) :: ESMF_GridCreateNoPeriDimI
```

ARGUMENTS:

```
        integer,                   intent(in),  optional :: minIndex(:)
        integer,                   intent(in)            :: countsPerDEDim1(:)
        integer,                   intent(in)            :: countsPerDEDim2(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,                   intent(in),  optional :: countsPerDEDim3(:)
        type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
        integer,                   intent(in),  optional :: coordDep1(:)
        integer,                   intent(in),  optional :: coordDep2(:)
        integer,                   intent(in),  optional :: coordDep3(:)
        integer,                   intent(in),  optional :: gridEdgeLWidth(:)
        integer,                   intent(in),  optional :: gridEdgeUWidth(:)
        integer,                   intent(in),  optional :: gridAlign(:)
        integer,                   intent(in),  optional :: gridMemLBound(:)
        type(ESMF_Index_Flag),     intent(in),  optional :: indexflag
        integer,                   intent(in),  optional :: petMap(:,:,:)
        character (len=*),         intent(in),  optional :: name
        integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, irregularly distributed grid (see Figure 13) without a periodic dimension. To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim arrays that are specified.
Section 27.3.4 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

countsPerDEDim1 This arrays specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).

countsPerDEDim2 This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).

[countsPerDEDim3] This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size size(countsPerDEDim1) x size(countsPerDEDim2) x size(countsPerDEDim3). If countsPerDEDim3 isn't present, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.23 ESMF_GridCreateNoPeriDim - Create a Grid with no periodic dim and a regular distribution

INTERFACE:

```
  ! Private name; call using ESMF_GridCreateNoPeriDim()
      function ESMF_GridCreateNoPeriDimR(regDecomp, decompFlag, &
        minIndex, maxIndex,                                     &
        coordSys, coordTypeKind,                                &
        coordDep1, coordDep2, coordDep3,                        &
        gridEdgeLWidth, gridEdgeUWidth, gridAlign,              &
        gridMemLBound, indexflag, petMap, name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreateNoPeriDimR
```

ARGUMENTS:

```
       integer,                    intent(in),  optional :: regDecomp(:)
       type(ESMF_Decomp_Flag),     intent(in),  optional :: decompflag(:)
       integer,                    intent(in),  optional :: minIndex(:)
       integer,                    intent(in)            :: maxIndex(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
       type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
       integer,                    intent(in),  optional :: coordDep1(:)
       integer,                    intent(in),  optional :: coordDep2(:)
       integer,                    intent(in),  optional :: coordDep3(:)
       integer,                    intent(in),  optional :: gridEdgeLWidth(:)
       integer,                    intent(in),  optional :: gridEdgeUWidth(:)
       integer,                    intent(in),  optional :: gridAlign(:)
       integer,                    intent(in),  optional :: gridMemLBound(:)
       type(ESMF_Index_Flag),      intent(in),  optional :: indexflag
       integer,                    intent(in),  optional :: petMap(:,:,:)
       character (len=*),          intent(in),  optional :: name
       integer,                    intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, regularly distributed grid (see Figure 13). To specify the distribution, the user passes in an array (`regDecomp`) specifying the number of DEs to divide each dimension into. The array `decompFlag` indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.
The arguments are:

[regDecomp] List that has the same number of elements as `maxIndex`. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_BALANCED` in all dimensions. Please see Section 9.13 for a full description of the possible options.

[minIndex] The bottom extent of the grid array. If not given then the value defaults to /1,1,1,.../.

maxIndex The upper extent of the grid array.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.24 ESMF_GridCreateNoPeriDim - Create a Grid with no periodic dim and an arbitrary distribution

INTERFACE:

```
  ! Private name; call using ESMF_GridCreateNoPeriodic()
      function ESMF_GridCreateNoPeriDimA(minIndex, maxIndex, &
        arbIndexCount, arbIndexList,                         &
        coordSys, coordTypeKind,                             &
        coordDep1, coordDep2, coordDep3,                     &
        distDim, name, rc)
```

RETURN VALUE:

```
      type(ESMF_Grid) :: ESMF_GridCreateNoPeriDimA
```

ARGUMENTS:

```
        integer,                   intent(in),  optional :: minIndex(:)
        integer,                   intent(in)            :: maxIndex(:)
        integer,                   intent(in)            :: arbIndexCount
        integer,                   intent(in)            :: arbIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_CoordSys_Flag),  intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag),  intent(in),  optional :: coordTypeKind
        integer,                   intent(in),  optional :: coordDep1(:)
        integer,                   intent(in),  optional :: coordDep2(:)
        integer,                   intent(in),  optional :: coordDep3(:)
        integer,                   intent(in),  optional :: distDim(:)
        character (len=*),         intent(in),  optional :: name
        integer,                   intent(out), optional :: rc
```

DESCRIPTION:

This method creates a single tile, arbitrarily distributed grid (see Figure 13). To specify the arbitrary distribution, the user passes in an 2D array of local indices, where the first dimension is the number of local grid cells specified by `localArbIndexCount` and the second dimension is the number of distributed dimensions.
`distDim` specifies which grid dimensions are arbitrarily distributed. The size of `distDim` has to agree with the size of the second dimension of `localArbIndex`.
The arguments are:

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

[maxIndex] The upper extend of the grid index ranges.

arbIndexCount The number of grid cells in the local DE. It is okay to have 0 grid cell in a local DE.

[arbIndexList] This 2D array specifies the indices of the PET LOCAL grid cells. The dimensions should be arbIndexCount * number of Distributed grid dimensions where arbIndexCount is the input argument specified below

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if the first dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=1) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep2] The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=2) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[coordDep3] The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_DIM_ARB/ where /ESMF_DIM_ARB/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_DIM_ARB/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=3) Please see Section 9.2 for a definition of ESMF_DIM_ARB.

[distDim] This array specifies which dimensions are arbitrarily distributed. The size of the array specifies the total distributed dimensions. if not specified, defaults is all dimensions will be arbitrarily distributed. The size has to agree with the size of the second dimension of `localArbIndex`.

[name] `ESMF_Grid` name.

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

27.6.25 ESMF_GridDestroy - Release resources associated with a Grid

INTERFACE:

```
      subroutine ESMF_GridDestroy(grid, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroys an `ESMF_Grid` object and related internal structures. This call does destroy internally created DistGrid and DELayout classes, for example those created by `ESMF_GridCreateShapeTile()`. It also destroys internally created coordinate/item Arrays, for example those created by `ESMF_GridAddCoord()`. However, if the user uses an externally created class, for example creating an Array and setting it using `ESMF_GridSetCoord()`, then that class is not destroyed by this method.
The arguments are:

grid `ESMF_Grid` to be destroyed.

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

27.6.26 ESMF_GridEmptyComplete - Complete a Grid with user set edge connections and an irregular distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridEmptyComplete()
      subroutine ESMF_GridEmptyCompleteEConnI(grid, minIndex,         &
        countsPerDEDim1,countsPerDeDim2,                        &
        countsPerDEDim3,                                        &
        connDim1, connDim2, connDim3,                           &
        coordSys, coordTypeKind,                                &
        coordDep1, coordDep2, coordDep3,                        &
        gridEdgeLWidth, gridEdgeUWidth, gridAlign,              &
        gridMemLBound, indexflag, petMap, name, rc)
```

ARGUMENTS:

```
        type (ESMF_Grid)                                :: grid
        integer,                  intent(in),  optional :: minIndex(:)
        integer,                  intent(in)            :: countsPerDEDim1(:)
        integer,                  intent(in)            :: countsPerDEDim2(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,                  intent(in),  optional :: countsPerDEDim3(:)
        type(ESMF_GridConn_Flag), intent(in),  optional :: connDim1(:)
        type(ESMF_GridConn_Flag), intent(in),  optional :: connDim2(:)
        type(ESMF_GridConn_Flag), intent(in),  optional :: connDim3(:)
        type(ESMF_CoordSys_Flag), intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag), intent(in),  optional :: coordTypeKind
        integer,                  intent(in),  optional :: coordDep1(:)
        integer,                  intent(in),  optional :: coordDep2(:)
        integer,                  intent(in),  optional :: coordDep3(:)
        integer,                  intent(in),  optional :: gridEdgeLWidth(:)
        integer,                  intent(in),  optional :: gridEdgeUWidth(:)
        integer,                  intent(in),  optional :: gridAlign(:)
        integer,                  intent(in),  optional :: gridMemLBound(:)
        type(ESMF_Index_Flag),    intent(in),  optional :: indexflag
        integer,                  intent(in),  optional :: petMap(:,:,:)
        character (len=*),        intent(in),  optional :: name
        integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method takes in an empty Grid created by `ESMF_GridEmptyCreate()`. It then completes the grid to form a single tile, irregularly distributed grid (see Figure 13). To specify the irregular distribution, the user passes in an array for each grid dimension, where the length of the array is the number of DEs in the dimension. Up to three dimensions can be specified, using the countsPerDEDim1, countsPerDEDim2, countsPerDEDim3 arguments. The index of each array element corresponds to a DE number. The array value at the index is the number of grid cells on the DE in that dimension. The dimCount of the grid is equal to the number of countsPerDEDim arrays that are specified.
Section 27.3.4 shows an example of using this method to create a 2D Grid with uniformly spaced coordinates. This creation method can also be used as the basis for grids with rectilinear coordinates or curvilinear coordinates.
The arguments are:

grid The empty `ESMF_Grid` to set information into and then commit.

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

countsPerDEDim1 This arrays specifies the number of cells per DE for index dimension 1 for the exclusive region (the center stagger location).

countsPerDEDim2 This array specifies the number of cells per DE for index dimension 2 for the exclusive region (center stagger location).

[countsPerDEDim3] This array specifies the number of cells per DE for index dimension 3 for the exclusive region (center stagger location). If not specified then grid is 2D.

[connDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size size(countsPerDEDim1) x size(countsPerDEDim2) x size(countsPerDEDim3). If countsPerDEDim3 isn't present, then the last dimension is of size 1.

[name] `ESMF_Grid` name.

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

27.6.27 ESMF_GridEmptyComplete - Complete a Grid with user set edge connections and a regular distribution

INTERFACE:

```
! Private name; call using ESMF_GridEmptyComplete()
   subroutine ESMF_GridEmptyCompleteEConnR(grid, regDecomp, decompFlag, &
      minIndex, maxIndex,                                     &
      connDim1, connDim2, connDim3,                        &
      coordSys, coordTypeKind,                             &
      coordDep1, coordDep2, coordDep3,                     &
      gridEdgeLWidth, gridEdgeUWidth, gridAlign,           &
      gridMemLBound, indexflag, petMap, name, rc)

!
```

ARGUMENTS:

```
      type (ESMF_Grid)                                 :: grid
      integer,                  intent(in),  optional :: regDecomp(:)
      type(ESMF_Decomp_Flag),   intent(in),  optional :: decompflag(:)
      integer,                  intent(in),  optional :: minIndex(:)
      integer,                  intent(in)            :: maxIndex(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_GridConn_Flag), intent(in),  optional :: connDim1(:)
      type(ESMF_GridConn_Flag), intent(in),  optional :: connDim2(:)
      type(ESMF_GridConn_Flag), intent(in),  optional :: connDim3(:)
      type(ESMF_CoordSys_Flag), intent(in),  optional :: coordSys
      type(ESMF_TypeKind_Flag), intent(in),  optional :: coordTypeKind
      integer,                  intent(in),  optional :: coordDep1(:)
      integer,                  intent(in),  optional :: coordDep2(:)
      integer,                  intent(in),  optional :: coordDep3(:)
      integer,                  intent(in),  optional :: gridEdgeLWidth(:)
      integer,                  intent(in),  optional :: gridEdgeUWidth(:)
      integer,                  intent(in),  optional :: gridAlign(:)
      integer,                  intent(in),  optional :: gridMemLBound(:)
      type(ESMF_Index_Flag),    intent(in),  optional :: indexflag
      integer,                  intent(in),  optional :: petMap(:,:,:)
      character (len=*),        intent(in),  optional :: name
      integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method takes in an empty Grid created by `ESMF_GridEmptyCreate()`. It then completes the grid to form a single tile, regularly distributed grid (see Figure 13). To specify the distribution, the user passes in an array (`regDecomp`) specifying the number of DEs to divide each dimension into. The array `decompFlag` indicates how the division into DEs is to occur. The default is to divide the range as evenly as possible.
The arguments are:

grid The empty `ESMF_Grid` to set information into and then commit.

[regDecomp] List that has the same number of elements as `maxIndex`. Each entry is the number of decounts for that dimension. If not specified, the default decomposition will be petCountx1x1..x1.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_HOMOGEN` in all dimensions. Please see Section 9.13 for a full description of the possible options.

[minIndex] The bottom extent of the grid array. If not given then the value defaults to /1,1,1,.../.

maxIndex The upper extent of the grid array.

[connDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] This array specifies the dependence of the first coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep2] This array specifies the dependence of the second coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[coordDep3] This array specifies the dependence of the third coordinate component on the three index dimensions described by `coordsPerDEDim1,2,3`. The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. If not present the default is 1,2,...,grid rank.

[gridEdgeLWidth] The padding around the lower edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridEdgeUWidth] The padding around the upper edges of the grid. This padding is between the index space corresponding to the cells and the boundary of the the exclusive region. This extra space is to contain the extra padding for non-center stagger locations, and should be big enough to hold any stagger in the grid. It is an error for this to be non-zero for a periodic dimension.

[gridAlign] Specification of how the stagger locations should align with the cell index space (can be overridden by the individual staggerAligns). If the `gridEdgeWidths` are not specified than this parameter implies the EdgeWidths.

[gridMemLBound] Specifies the lower index range of the memory of every DE in this Grid. Only used when indexflag is `ESMF_INDEX_USER`. May be overridden by staggerMemLBound.

[indexflag] Indicates the indexing scheme to be used in the new Grid. Please see Section 9.24 for the list of options. If not present, defaults to ESMF_INDEX_DELOCAL.

[petMap] Sets the mapping of pets to the created DEs. This 3D should be of size regDecomp(1) x regDecomp(2) x regDecomp(3) If the Grid is 2D, then the last dimension is of size 1.

[name] ESMF_Grid name.

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

27.6.28 ESMF_GridEmptyComplete - Complete a Grid with user set edge connections and an arbitrary distribution

INTERFACE:

```
   ! Private name; call using ESMF_GridEmptyComplete()
       subroutine ESMF_GridEmptyCompleteEConnA(grid, minIndex, maxIndex,  &
         arbIndexCount, arbIndexList,                               &
         connDim1, connDim2, connDim3,                           &
         coordSys, coordTypeKind,                                &
         coordDep1, coordDep2, coordDep3,                        &
         distDim, name, rc)
  !
```

ARGUMENTS:

```
        type (ESMF_Grid)                                :: grid
        integer,                  intent(in),  optional :: minIndex(:)
        integer,                  intent(in)            :: maxIndex(:)
        integer,                  intent(in)            :: arbIndexCount
        integer,                  intent(in)            :: arbIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_GridConn_Flag), intent(in),  optional :: connDim1(:)
        type(ESMF_GridConn_Flag), intent(in),  optional :: connDim2(:)
        type(ESMF_GridConn_Flag), intent(in),  optional :: connDim3(:)
        type(ESMF_CoordSys_Flag), intent(in),  optional :: coordSys
        type(ESMF_TypeKind_Flag), intent(in),  optional :: coordTypeKind
        integer,                  intent(in),  optional :: coordDep1(:)
        integer,                  intent(in),  optional :: coordDep2(:)
        integer,                  intent(in),  optional :: coordDep3(:)
        integer,                  intent(in),  optional :: distDim(:)
        character (len=*),        intent(in),  optional :: name
        integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method takes in an empty Grid created by ESMF_GridEmptyCreate(). It then completes the grid to form a single tile, arbitrarily distributed grid (see Figure 13). To specify the arbitrary distribution, the user passes in an 2D array of local indices, where the first dimension is the number of local grid cells specified by localArbIndexCount and the second dimension is the number of distributed dimensions.
distDim specifies which grid dimensions are arbitrarily distributed. The size of distDim has to agree with the size of the second dimension of localArbIndex.
The arguments are:

grid The empty ESMF_Grid to set information into and then commit.

[minIndex] Tuple to start the index ranges at. If not present, defaults to /1,1,1,.../.

[maxIndex] The upper extend of the grid index ranges.

arbIndexCount The number of grid cells in the local DE. It is okay to have 0 grid cell in a local DE.

[arbIndexList] This 2D array specifies the indices of the PET LOCAL grid cells. The dimensions should be arbIndexCount * number of Distributed grid dimensions where arbIndexCount is the input argument specified below

[connDim1] Fortran array describing the index dimension 1 connections. The first element represents the minimum end of dimension 1. The second element represents the maximum end of dimension 1. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connDim2] Fortran array describing the index dimension 2 connections. The first element represents the minimum end of dimension 2. The second element represents the maximum end of dimension 2. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[connDim3] Fortran array describing the index dimension 3 connections. The first element represents the minimum end of dimension 3. The second element represents the maximum end of dimension 3. If array is only one element long, then that element is used for both the minimum and maximum end. Please see Section 27.2.2 for a list of valid options. If not present, defaults to ESMF_GRIDCONN_NONE.

[coordSys] The coordinate system of the grid coordinate data. For a full list of options, please see Section 27.2.1. If not specified then defaults to ESMF_COORDSYS_SPH_DEG.

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

[coordDep1] The size of the array specifies the number of dimensions of the first coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_GRID_ARBDIM/ where /ESMF_GRID_ARBDIM/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_GRID_ARBDIM/ if the first dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=1) Please see Section 9.2 for a definition of ESMF_GRID_ARBDIM.

[coordDep2] The size of the array specifies the number of dimensions of the second coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_GRID_ARBDIM/ where /ESMF_GRID_ARBDIM/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_GRID_ARBDIM/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=2) Please see Section 9.2 for a definition of ESMF_GRID_ARBDIM.

[coordDep3] The size of the array specifies the number of dimensions of the third coordinate component array. The values specify which of the index dimensions the corresponding coordinate arrays map to. The format should be /ESMF_GRID_ARBDIM/ where /ESMF_GRID_ARBDIM/ is mapped to the collapsed 1D dimension from all the arbitrarily distributed dimensions. n is the dimension that is not distributed (if exists). If not present the default is /ESMF_GRID_ARBDIM/ if this dimension is arbitararily distributed, or /n/ if not distributed (i.e. n=3) Please see Section 9.2 for a definition of ESMF_GRID_ARBDIM.

[distDim] This array specifies which dimensions are arbitrarily distributed. The size of the array specifies the total distributed dimensions. if not specified, defaults is all dimensions will be arbitrarily distributed. The size has to agree with the size of the second dimension of `localArbIndex`.

[name] `ESMF_Grid` name.

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

27.6.29 ESMF_GridEmptyCreate - Create a Grid that has no contents

INTERFACE:

```
function ESMF_GridEmptyCreate(rc)
```

RETURN VALUE:

```
type(ESMF_Grid) :: ESMF_GridEmptyCreate
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Partially create an `ESMF_Grid` object. This function allocates an `ESMF_Grid` object, but doesn't allocate any coordinate storage or other internal structures. The `ESMF_GridEmptyComplete()` calls can be used to set the values in the grid object and to construct the internal structure.
The arguments are:

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

27.6.30 ESMF_GridGet - Get information about a Grid

INTERFACE:

```
   ! Private name; call using ESMF_GridGet()
       subroutine ESMF_GridGetDefault(grid, coordTypeKind, &
         dimCount, tileCount, staggerlocCount, localDECount, distgrid, &
         distgridToGridMap, coordDimCount, coordDimMap, arbDim, &
         rank, arbDimCount, gridEdgeLWidth, gridEdgeUWidth, gridAlign,  &
         indexFlag, status, name, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),            intent(in)            :: grid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_TypeKind_Flag),   intent(out), optional :: coordTypeKind
       integer,                    intent(out), optional :: dimCount
       integer,                    intent(out), optional :: tileCount
       integer,                    intent(out), optional :: staggerlocCount
       integer,                    intent(out), optional :: localDECount
       type(ESMF_DistGrid),        intent(out), optional :: distgrid
       integer,       target,      intent(out), optional :: distgridToGridMap(:)
       integer,       target,      intent(out), optional :: coordDimCount(:)
       integer,       target,      intent(out), optional :: coordDimMap(:,:)
       integer,                    intent(out), optional :: arbDim
       integer,                    intent(out), optional :: rank
       integer,                    intent(out), optional :: arbDimCount
       integer,       target,      intent(out), optional :: gridEdgeLWidth(:)
       integer,       target,      intent(out), optional :: gridEdgeUWidth(:)
       integer,       target,      intent(out), optional :: gridAlign(:)
       type(ESMF_Index_Flag),      intent(out), optional :: indexflag
       type(ESMF_GridStatus_Flag), intent(out), optional :: status
       character (len=*),          intent(out), optional :: name
       integer,                    intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets various types of information about a grid.
The arguments are:

grid Grid to get the information from.

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

[dimCount] DimCount of the Grid object.

[tileCount] The number of logically rectangular tiles in the grid.

[staggerlocCount] The number of stagger locations.

[localDECount] The number of DEs in this grid on this PET.

[distgrid] The structure describing the distribution of the grid.

[distgridToGridMap] List that has as many elements as the distgrid dimCount. This array describes mapping between the grids dimensions and the distgrid.

[coordDimCount] List that has as many elements as the grid dimCount (from arrayspec). Gives the dimension of each component (e.g. x) array. This is to allow factorization of the coordinate arrays. If not specified all arrays are the same size as the grid.

[coordDimMap] 2D list of size grid dimCount x grid dimCount. This array describes the map of each component array's dimensions onto the grids dimensions.

[arbDim] The distgrid dimension that is mapped by the arbitrarily distributed grid dimensions.

[rank] The count of the memory dimensions, it is the same as dimCount for a non-arbitrarily distributed grid, and equal or less for a arbitrarily distributed grid.

[arbDimCount] The number of dimensions distributed arbitrarily for an arbitrary grid, 0 if the grid is non-arbitrary.

[gridEdgeLWidth] The padding around the lower edges of the grid. The array should be of size greater or equal to the Grid dimCount.

[gridEdgeUWidth] The padding around the upper edges of the grid. The array should be of size greater or equal to the Grid dimCount.

[gridAlign] Specification of how the stagger locations should align with the cell index space. The array should be of size greater or equal to the Grid dimCount.

[indexflag] Flag indicating the indexing scheme being used in the Grid. Please see Section 9.24 for the list of options.

[status] Flag indicating the status of the Grid. Please see Section 27.2.5 for the list of options.

[name] `ESMF_Grid` name.

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

27.6.31 ESMF_GridGet - Get information about a particular DE in a Grid

INTERFACE:

```
   ! Private name; call using ESMF_GridGet()
       subroutine ESMF_GridGetPLocalDe(grid, localDe, &
         isLBound,isUBound, arbIndexCount, arbIndexList, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)            :: grid
       integer,                intent(in)            :: localDe
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       logical,                intent(out), optional :: isLBound(:)
       logical,                intent(out), optional :: isUBound(:)
       integer,                intent(out), optional :: arbIndexCount
       integer,        target, intent(out), optional :: arbIndexList(:,:)
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This call gets information about a particular local DE in a Grid.
The arguments are:

grid Grid to get the information from.

[localDe] The local DE from which to get the information. `[0,..,localDeCount-1]`

[isLBound] Upon return, for each dimension this indicates if the DE is a lower bound of the Grid. `isLBound` must be allocated to be of size equal to the Grid dimCount.

[isUBound] Upon return, for each dimension this indicates if the DE is an upper bound of the Grid. `isUBound` must be allocated to be of size equal to the Grid dimCount.

[arbIndexCount] The number of local cells for an arbitrarily distributed grid

[arbIndexList] The 2D array storing the local cell indices for an arbitrarily distributed grid. The size of the array is arbIndexCount * arbDimCount

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

27.6.32 ESMF_GridGet - Get information about a particular DE in a stagger location in a Grid

INTERFACE:

```
   ! Private name; call using ESMF_GridGet()
       subroutine ESMF_GridGetPLocalDePSloc(grid, staggerloc, localDE, &
         exclusiveLBound, exclusiveUBound, exclusiveCount,  &
         computationalLBound, computationalUBound, computationalCount, rc)
```

ARGUMENTS:

```
      type(ESMF_Grid),        intent(in)            :: grid
      integer,                intent(in)            :: localDe
      type (ESMF_StaggerLoc), intent(in)            :: staggerloc
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,        target, intent(out), optional :: exclusiveLBound(:)
      integer,        target, intent(out), optional :: exclusiveUBound(:)
      integer,        target, intent(out), optional :: exclusiveCount(:)
      integer,        target, intent(out), optional :: computationalLBound(:)
      integer,        target, intent(out), optional :: computationalUBound(:)
      integer,        target, intent(out), optional :: computationalCount(:)
      integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets information about the range of index space which a particular stagger location occupies. This call differs from the coordinate bound calls (e.g. `ESMF_GridGetCoord`) in that a given coordinate array may only occupy a subset of the Grid's dimensions, and so these calls may not give all the bounds of the stagger location. The bounds from this call are the full bounds, and so for example, give the appropriate bounds for allocating a Fortran array to hold data residing on the stagger location. Note that unlike the output from the Array, these values also include the undistributed dimensions and are ordered to reflect the order of the indices in the Grid. This call will still give correct values even if the stagger location does not contain coordinate arrays (e.g. if `ESMF_GridAddCoord` hasn't yet been called on the stagger location).
The arguments are:

grid Grid to get the information from.

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations.

[localDe] The local DE from which to get the information. `[0,..,localDeCount-1]`

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to the Grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to the Grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items,`exclusiveUBound-exclusiveLBound+1`, in the exclusive region per dimension. `exclusiveCount` must be allocated to be of size equal to the Grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the computational region. `computationalLBound` must be allocated to be of size equal to the Grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the computational region. `computationalUBound` must be allocated to be of size equal to the Grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension. (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to the Grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

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

27.6.33 ESMF_GridGet - Get information about a particular stagger location in a Grid

INTERFACE:

```
   ! Private name; call using ESMF_GridGet()
       subroutine ESMF_GridGetPSloc(grid, staggerloc, &
         distgrid, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)            :: grid
       type (ESMF_StaggerLoc), intent(in)            :: staggerloc
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_DistGrid),    intent(out), optional :: distgrid
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets information about a particular stagger location. This information is useful for creating an ESMF Array to hold the data at the stagger location.
The arguments are:

grid Grid to get the information from.

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations.

[distgrid] The structure describing the distribution of this staggerloc in this grid.

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

27.6.34 ESMF_GridGet - Get information about a particular stagger location and tile

INTERFACE:

```
   ! Private name; call using ESMF_GridGet()
       subroutine ESMF_GridGetPSlocPTile(grid, tile, staggerloc, &
         minIndex, maxIndex, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)            :: grid
       integer,                intent(in)            :: tile
       type (ESMF_StaggerLoc), intent(in)            :: staggerloc
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,        target, intent(out), optional :: minIndex(:)
       integer,        target, intent(out), optional :: maxIndex(:)
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets information about a particular stagger location. This information is useful for creating an ESMF Array to hold the data at the stagger location.
The arguments are:

grid Grid to get the information from.

tile The tile number to get the data from. Tile numbers range from 1 to TileCount.

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations.

[minIndex] Upon return this holds the global lower index of this stagger location. `minIndex` must be allocated to be of size equal to the grid DimCount. Note that this value is only for the first Grid tile, as multigrid support is added, this interface will likely be changed or moved to adapt.

[maxIndex] Upon return this holds the global upper index of this stagger location. `maxIndex` must be allocated to be of size equal to the grid DimCount. Note that this value is only for the first Grid tile, as multigrid support is added, this interface will likely be changed or moved to adapt.

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

27.6.35 ESMF_GridGetCoord - Get a Fortran pointer to Grid coord data and coord bounds

INTERFACE:

```
        subroutine ESMF_GridGetCoord(grid, coordDim,    &
          staggerloc, localDE, <pointer argument>,                      &
          exclusiveLBound, exclusiveUBound, exclusiveCount,             &
          computationalLBound, computationalUBound, computationalCount,  &
          totalLBound, totalUBound, totalCount,                         &
          datacopyflag, rc)
```

ARGUMENTS:

```
      type(ESMF_Grid),       intent(in)            :: grid
      integer,               intent(in)            :: coordDim
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type (ESMF_StaggerLoc) intent(in),  optional :: staggerloc
      integer,               intent(in),  optional :: localDE
      <pointer argument>, see below for supported values
      integer,               intent(out), optional :: exclusiveLBound(:)
      integer,               intent(out), optional :: exclusiveUBound(:)
      integer,               intent(out), optional :: exclusiveCount(:)
      integer,               intent(out), optional :: computationalLBound(:)
      integer,               intent(out), optional :: computationalUBound(:)
      integer,               intent(out), optional :: computationalCount(:)
      integer,               intent(out), optional :: totalLBound(:)
      integer,               intent(out), optional :: totalUBound(:)
      integer,               intent(out), optional :: totalCount(:)
      type(ESMF_DataCopy_Flag), intent(in),  optional :: datacopyflag
      integer,               intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets a Fortran pointer to the piece of memory which holds the coordinate data on the local DE for the given coordinate dimension and stagger locations. This is useful, for example, for setting the coordinate values in a Grid, or for reading the coordinate values. Currently this method supports up to three coordinate dimensions, of either R4 or R8 datatype. See below for specific supported values. If the coordinates that you are trying to retrieve are of higher dimension, use the `ESMF_GetCoord()` interface that returns coordinate values in an `ESMF_Array` instead. That interface supports the retrieval of coordinates up to 7D.
Supported values for the <pointer argument> are:

real(ESMF_KIND_R4), pointer :: farrayPtr(:)

real(ESMF_KIND_R4), pointer :: farrayPtr(:,:)

real(ESMF_KIND_R4), pointer :: farrayPtr(:,:,:)

real(ESMF_KIND_R8), pointer :: farrayPtr(:)

real(ESMF_KIND_R8), pointer :: farrayPtr(:,:)

real(ESMF_KIND_R8), pointer :: farrayPtr(:,:,:)

The arguments are:

grid Grid to get the information from.

coordDim The coordinate dimension to get the data from (e.g. 1=x).

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

[localDE] The local DE to get the information for. `[0,..,localDeCount-1]`

farrayPtr The pointer to the coordinate data.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to the coord dimCount.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to the coord dimCount.

[exclusiveCount] Upon return this holds the number of items, `exclusiveUBound-exclusiveLBound+1`, in the exclusive region per dimension. `exclusiveCount` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the stagger region. `computationalLBound` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the stagger region. `exclusiveUBound` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalUBound] Upon return this holds the upper bounds of the total region. `totalUBound` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `totalCount` must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[datacopyflag] If not specified, default to `ESMF_DATACOPY_REFERENCE`, in this case farrayPtr is a reference to the data in the Grid coordinate arrays. Please see Section 9.12 for further description and a list of valid values.

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

27.6.36 ESMF_GridGetCoord - Get coordinates and put in an Array

INTERFACE:

```
   ! Private name; call using ESMF_GridGetCoord()
       subroutine ESMF_GridGetCoordIntoArray(grid, coordDim, staggerloc, &
         array, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)            :: grid
       integer,                intent(in)            :: coordDim
       type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
       type(ESMF_Array),       intent(out)           :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method allows the user to get access to the ESMF Array holding coordinate data at a particular stagger location. This is useful, for example, to set the coordinate values. To have an Array to access, the coordinate Arrays must have already been allocated, for example by `ESMF_GridAddCoord` or `ESMF_GridSetCoord`.
The arguments are:

grid The grid to get the coord array from.

coordDim The coordinate dimension to get the data from (e.g. 1=x).

staggerloc The stagger location from which to get the arrays. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

array An array into which to put the coordinate infomation.

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

27.6.37 ESMF_GridGetCoord - Get coordinates from a specific index location

INTERFACE:

```
   ! Private name; call using ESMF_GridGetCoord()
       subroutine ESMF_GridGetCoordR4(grid, staggerloc, localDe, &
         index, coord, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)             :: grid
       type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
       integer,                intent(in),  optional :: localDE
       integer,                intent(in)             :: index(:)
       real(ESMF_KIND_R4),     intent(out)            :: coord(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Given a specific index location in a Grid, this method returns the full set of coordinates from that index location. This method will eventually be overloaded to support the full complement of types supported by the Grid.
The arguments are:

grid Grid to get the information from.

[localDE] The local DE to get the information for. `[0,..,localDeCount-1]`

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

index This array holds the index location to be queried in the Grid. This array must at least be of the size Grid rank.

coord This array will be filled with the coordinate data. This array must at least be of the size Grid rank.

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

27.6.38 ESMF_GridGetCoord - Get coordinates from a specific index location

INTERFACE:

```
   ! Private name; call using ESMF_GridGetCoord()
       subroutine ESMF_GridGetCoordR8(grid, staggerloc, localDE, &
         index, coord, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)             :: grid
       type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
       integer,                intent(in),  optional :: localDE
       integer,                intent(in)             :: index(:)
       real(ESMF_KIND_R8),     intent(out)            :: coord(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Given a specific index location in a Grid, this method returns the full set of coordinates from that index location. This method will eventually be overloaded to support the full complement of types supported by the Grid.
The arguments are:

grid Grid to get the information from.

[localDE] The local DE to get the information for. `[0,..,localDeCount-1]`

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

index This array holds the index location to be queried in the Grid. This array must at least be of the size Grid rank.

coord This array will be filled with the coordinate data. This array must at least be of the size Grid rank.

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

27.6.39 ESMF_GridGetCoordBounds - Get Grid coordinate bounds

INTERFACE:

```
      subroutine ESMF_GridGetCoordBounds(grid, coordDim,   &
        staggerloc, localDE, exclusiveLBound, exclusiveUBound, &
        exclusiveCount, computationalLBound, computationalUBound , &
        computationalCount, totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
      type(ESMF_Grid),        intent(in)            :: grid
      integer,                intent(in)            :: coordDim
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
      integer,                intent(in),  optional :: localDE
      integer,        target, intent(out), optional :: exclusiveLBound(:)
      integer,        target, intent(out), optional :: exclusiveUBound(:)
      integer,        target, intent(out), optional :: exclusiveCount(:)
      integer,        target, intent(out), optional :: computationalLBound(:)
      integer,        target, intent(out), optional :: computationalUBound(:)
      integer,        target, intent(out), optional :: computationalCount(:)
      integer,        target, intent(out), optional :: totalLBound(:)
      integer,        target, intent(out), optional :: totalUBound(:)
      integer,        target, intent(out), optional :: totalCount(:)
      integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets information about the range of index space which a particular piece of coordinate data occupies. In other words, this method returns the bounds of the coordinate arrays. Note that unlike the output from the Array, these values also include the undistributed dimensions and are ordered to reflect the order of the indices in the coordinate.

So, for example, totalLBound and totalUBound should match the bounds of the Fortran array retrieved by ESMF_GridGetCoord.
The arguments are:

grid Grid to get the information from.

coordDim The coordinate dimension to get the information for (e.g. 1=x).

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

[localDE] The local DE from which to get the information. [0,..,localDeCount-1]

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. exclusiveLBound must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. exclusiveUBound must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items, exclusiveUBound-exclusiveLBound+1, in the exclusive region per dimension. exclusiveCount must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the stagger region. computationalLBound must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the stagger region. computationalUBound must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. computationalUBound-computationalLBound+1). computationalCount must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. totalLBound must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalUBound] Upon return this holds the upper bounds of the total region. totalUBound must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. totalUBound-totalLBound+1). totalCount must be allocated to be of size equal to the coord dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

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

27.6.40 ESMF_GridGetItem - Get a Fortran pointer to Grid item data and item bounds

INTERFACE:

```
      subroutine ESMF_GridGetItem(grid, itemflag,  &
        staggerloc, localDE, <pointer argument>,
        exclusiveLBound, exclusiveUBound, exclusiveCount, &
        computationalLBound, computationalUBound, computationalCount,  &
        totalLBound, totalUBound, totalCount,                          &
        datacopyflag, rc)
```

ARGUMENTS:

```
     type(ESMF_Grid),      intent(in)           :: grid
     type (ESMF_GridItem_Flag),intent(in)       :: itemflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
     integer,              intent(in),  optional :: localDE
     <pointer argument>, see below for supported values
     integer,              intent(out), optional :: exclusiveLBound(:)
     integer,              intent(out), optional :: exclusiveUBound(:)
     integer,              intent(out), optional :: exclusiveCount(:)
     integer,              intent(out), optional :: computationalLBound(:)
     integer,              intent(out), optional :: computationalUBound(:)
     integer,              intent(out), optional :: computationalCount(:)
     integer,              intent(out), optional :: totalLBound(:)
     integer,              intent(out), optional :: totalUBound(:)
     integer,              intent(out), optional :: totalCount(:)
     type(ESMF_DataCopy_Flag), intent(in),  optional :: datacopyflag
     integer,              intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets a Fortran pointer to the piece of memory which holds the item data on the local DE for the given stagger locations. This is useful, for example, for setting the item values in a Grid, or for reading the item values. Currently this method supports up to three grid dimensions, but is limited to the I4 datatype. See below for specific supported values. If the item values that you are trying to retrieve are of higher dimension, use the `ESMF_GetItem()` interface that returns coordinate values in an `ESMF_Array` instead. That interface supports the retrieval of coordinates up to 7D.
Supported values for the <pointer argument> are:

integer(ESMF_KIND_I4), pointer :: farrayPtr(:)

integer(ESMF_KIND_I4), pointer :: farrayPtr(:,:)

integer(ESMF_KIND_I4), pointer :: farrayPtr(:,:,:)

real(ESMF_KIND_R4), pointer :: farrayPtr(:)

real(ESMF_KIND_R4), pointer :: farrayPtr(:,:)

real(ESMF_KIND_R4), pointer :: farrayPtr(:,:,:)

real(ESMF_KIND_R8), pointer :: farrayPtr(:)

real(ESMF_KIND_R8), pointer :: farrayPtr(:,:)

real(ESMF_KIND_R8), pointer :: farrayPtr(:,:,:)

The arguments are:

grid Grid to get the information from.

itemflag The item to get the information for. Please see Section 27.2.3 for a list of valid items.

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

localDE The local DE to get the information for. `[0,..,localDeCount-1]`

farrayPtr The pointer to the item data.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to the grid dimCount.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to the grid dimCount.

[exclusiveCount] Upon return this holds the number of items in the exclusive region per dimension (i.e. `exclusiveUBound-exclusiveLBound+1`). `exclusiveCount` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the stagger region. `computationalLBound` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the stagger region. `exclusiveUBound` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalUBound] Upon return this holds the upper bounds of the total region. `totalUBound` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `totalCount` must be allocated to be of size equal to the grid dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[datacopyflag] If not specified, default to `ESMF_DATACOPY_REFERENCE`, in this case farrayPtr is a reference to the data in the Grid item arrays. Please see Section 9.12 for further description and a list of valid values.

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

27.6.41 ESMF_GridGetItem - Get item and put into an Array

INTERFACE:

```
   ! Private name; call using ESMF_GridGetItem()
       subroutine ESMF_GridGetItemIntoArray(grid, itemflag,  staggerloc, &
         array, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),            intent(in)            :: grid
       type (ESMF_GridItem_Flag), intent(in)            :: itemflag
       type (ESMF_StaggerLoc),    intent(in),  optional :: staggerloc
       type(ESMF_Array),          intent(out)           :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method allows the user to get access to the ESMF Array holding item data at a particular stagger location. This is useful, for example, to set the item values. To have an Array to access, the item Array must have already been allocated, for example by `ESMF_GridAddItem` or `ESMF_GridSetItem`.
The arguments are:

itemflag The item from which to get the arrays. Please see Section 27.2.3 for a list of valid items.

staggerloc The stagger location from which to get the arrays. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

array An array into which to put the item infomation.

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

27.6.42 ESMF_GridGetItemBounds - Get Grid item bounds

INTERFACE:

```
       subroutine ESMF_GridGetItemBounds(grid, itemflag,  &
         staggerloc, localDE, &
         exclusiveLBound, exclusiveUBound, exclusiveCount, &
         computationalLBound, computationalUBound, computationalCount,  &
         totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
       type(ESMF_Grid),        intent(in)            :: grid
       type (ESMF_GridItem_Flag), intent(in)         :: itemflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
       integer,                intent(in),  optional :: localDE
       integer,        target, intent(out), optional :: exclusiveLBound(:)
       integer,        target, intent(out), optional :: exclusiveUBound(:)
       integer,        target, intent(out), optional :: exclusiveCount(:)
```

```
      integer,        target, intent(out), optional :: computationalLBound(:)
      integer,        target, intent(out), optional :: computationalUBound(:)
      integer,        target, intent(out), optional :: computationalCount(:)
      integer,        target, intent(out), optional :: totalLBound(:)
      integer,        target, intent(out), optional :: totalUBound(:)
      integer,        target, intent(out), optional :: totalCount(:)
      integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method gets information about the range of index space which a particular piece of item data occupies. In other words, this method returns the bounds of the item arrays. Note that unlike the output from the Array, these values also include the undistributed dimensions and are ordered to reflect the order of the indices in the item. So, for example, `totalLBound` and `totalUBound` should match the bounds of the Fortran array retrieved by `ESMF_GridGetItem`.
The arguments are:

grid Grid to get the information from.

itemflag The item to get the information for. Please see Section 27.2.3 for a list of valid items.

staggerloc The stagger location to get the information for. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

localDE The local DE from which to get the information. `[0,..,localDeCount-1]`

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items, `exclusiveUBound-exclusiveLBound+1`, in the exclusive region per dimension. `exclusiveCount` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the stagger region. `computationalLBound` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the stagger region. `computationalUBound` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalUBound] Upon return this holds the upper bounds of the total region. `totalUBound` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `totalCount` must be allocated to be of size equal to the item dimCount. Please see Section 27.3.16 for a description of the regions and their associated bounds and counts.

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

27.6.43 ESMF_GridMatch - Check if two Grid objects match

INTERFACE:

```
  function ESMF_GridMatch(grid1, grid2, rc)
```

RETURN VALUE:

```
    type(ESMF_GridMatch_Flag) :: ESMF_GridMatch
```

ARGUMENTS:

```
    type(ESMF_Grid),  intent(in)              :: grid1
    type(ESMF_Grid),  intent(in)              :: grid2
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,          intent(out),  optional  :: rc
```

DESCRIPTION:

Check if `grid1` and `grid2` match. Returns a range of values of type ESMF_GridMatch indicating how closely the Grids match. For a description of the possible return values, please see 27.2.4. Please also note that this call returns the match for the piece of the Grids on the local PET only. It's entirely possible for this call to return a different match on different PETs for the same Grids. The user is responsible for computing the global match across the set of PETs. The arguments are:

grid1 `ESMF_Grid` object.

grid2 `ESMF_Grid` object.

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

27.6.44 ESMF_GridSetCoord - Set coordinates using Arrays

INTERFACE:

```
      subroutine ESMF_GridSetCoordFromArray(grid, coordDim, staggerloc, &
        array, rc)
```

ARGUMENTS:

```
      type(ESMF_Grid),        intent(in)            :: grid
      integer,                intent(in)            :: coordDim
      type (ESMF_StaggerLoc), intent(in),  optional :: staggerloc
      type(ESMF_Array),       intent(in)            :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method sets the passed in Array as the holder of the coordinate data for stagger location `staggerloc` and coordinate `coord`. If the location already contains an Array, then this one overwrites it.
The arguments are:

grid The grid to set the coord in.

coordDim The coordinate dimension to put the data in (e.g. 1=x).

staggerloc The stagger location into which to copy the arrays. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

array An array to set the grid coordinate information from.

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

27.6.45 ESMF_GridSetItem - Set an item using an Array

INTERFACE:

```
      subroutine ESMF_GridSetItemFromArray(grid, itemflag,  staggerloc, &
        array, rc)
```

ARGUMENTS:

```
      type(ESMF_Grid),            intent(in)            :: grid
      type (ESMF_GridItem_Flag), intent(in)            :: itemflag
      type (ESMF_StaggerLoc),    intent(in),  optional :: staggerloc
      type(ESMF_Array),          intent(in)            :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method sets the passed in Array as the holder of the item data for stagger location `staggerloc` and coordinate `coord`. If the location already contains an Array, then this one overwrites it.
Eventually there should be an Add, Get,... like for the Coords to make things easy for the user (except restricted to just I4??)
The arguments are:

grid The grid in which to set the array.

itemflag The item into which to copy the arrays. Please see Section 27.2.3 for a list of valid items.

staggerloc The stagger location into which to copy the arrays. Please see Section 27.2.7 for a list of predefined stagger locations. If not present, defaults to ESMF_STAGGERLOC_CENTER.

array An array to set the grid item information from.

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

27.6.46 ESMF_GridValidate - Validate Grid internals

INTERFACE:

```
   subroutine ESMF_GridValidate(grid, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the `Grid` is internally consistent. Note that one of the checks that the Grid validate does is the Grid status. Currently, the validate will return an error if the grid is not at least `ESMF_GRIDSTATUS_COMPLETE`. This means that if a Grid was created with the `ESMF_GridEmptyCreate` method, it must also have been finished with `ESMF_GridEmptyComplete()` to be valid. If a Grid was created with another create call it should automatically have the correct status level to pass the status part of the validate. The Grid validate at this time doesn't check for the presence or consistency of the Grid coordinates. The method returns an error code if problems are found.
The arguments are:

grid Specified `ESMF_Grid` object.

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

27.7 Class API: StaggerLoc Methods

27.7.1 ESMF_StaggerLocSet - Set a StaggerLoc to a particular position in the cell

INTERFACE:

```
   ! Private name; call using ESMF_StaggerLocSet()
      subroutine ESMF_StaggerLocSetAllDim(staggerloc, loc, keywordenforcer, rc)
```

ARGUMENTS:

```
       type (ESMF_StaggerLoc), intent(inout) :: staggerloc
       integer,                intent(in)    :: loc(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, optional                     :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets a custom `staggerloc` to a position in a cell by using the array `loc`. The values in the array should only be 0,1. If loc(i) is 0 it means the position should be in the center in that dimension. If loc(i) is 1 then for dimension i, the position should be on the side of the cell. Please see Section 27.3.22 for diagrams and further discussion of custom stagger locations.
The arguments are:

staggerloc Grid location to be initialized

loc Array holding position data. Each entry in `loc` should only be 0 or 1. note that dimensions beyond those specified are set to 0.

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

27.7.2 ESMF_StaggerLocSet - Set one dimension of a StaggerLoc to a particular position

INTERFACE:

```
   ! Private name; call using ESMF_StaggerLocSet()
       subroutine ESMF_StaggerLocSetDim(staggerloc, dim, loc, keywordenforcer, rc)
```

ARGUMENTS:

```
       type (ESMF_StaggerLoc), intent(inout) :: staggerloc
       integer,                intent(in)    :: dim
       integer,                intent(in)    :: loc
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, optional                     :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets a particular dimension of a custom `staggerloc` to a position in a cell by using the variable `loc`. The variable `loc` should only be 0,1. If `loc` is 0 it means the position should be in the center in that dimension. If `loc` is +1 then for the dimension, the position should be on the positive side of the cell. Please see Section 27.3.22 for diagrams and further discussion of custom stagger locations.
The arguments are:

staggerloc Stagger location to be initialized

dim Dimension to be changed (1-7).

loc Position data should be either 0,1.

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

27.7.3 ESMF_StaggerLocString - Return a StaggerLoc as a string

INTERFACE:

```
       subroutine ESMF_StaggerLocString(staggerloc, string, keywordenforcer, rc)
```

ARGUMENTS:

```
       type(ESMF_StaggerLoc), intent(in)  :: staggerloc
       character (len = *),   intent(out) :: string
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, optional,     intent(out) :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Return an `ESMF_StaggerLoc` as a printable string.
The arguments are:

staggerloc The `ESMF_StaggerLoc` to be turned into a string.

string Return string.

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

27.7.4 ESMF_StaggerLocPrint - Print information of a StaggerLoc object

INTERFACE:

```
      subroutine ESMF_StaggerLocPrint(staggerloc, keywordenforcer, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Print the internal data members of an `ESMF_StaggerLoc` object.

The arguments are:

staggerloc ESMF_StaggerLoc object as the method input

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

28 LocStream Class

28.1 Description

A location stream (LocStream) is used to represent the locations of a set of data points. The values of the data points are stored within a Field or FieldBundle created using the LocStream.
In the data assimilation world, LocStreams can be thought of as a set of observations. Their locations are generally described using Cartesian (x, y, z), or (lat, lon, height) coordinates. There is no assumption of any regularity in the positions of the points. To make the concept more general, the locations for each data point are represented using a construct called *keys*, which can include other descriptors besides location.
Although keys are similar in concept to ESMF Attributes they have important differences. First, keys always occur as vectors, never as scalars. Second, keys are local to the DE: each DE can have a different key list with a different number of of elements. Third, the local key list always has the same number of elements as there are local observations on that DE. Finally, keys may be used for the distribution of LocStreams. As such, they must be defined before the LocStream is distributed.
LocStreams can be very large. Data assimilation systems might use LocStreams with up to 10^8 observations, so efficiency is critical.
Common operations involving LocStreams are similar to those involving Grids. In data assimilation, for example, there is an immediate need to:

1. Create a Field or FieldBundle on a LocStream
2. Redistribute data between Fields defined on LocStreams
3. Gather or scatter a FieldBundle defined on a LocStream from/to a root DE
4. Halo region exchange for a Field defined by a haloed LocStream
5. Extract Fortran array from Field which was defined by a LocStream

The operations on the Fortran arrays underlying LocStreams are usually simple numerical ones. However, it is necessary to sort them in place, and access only portions of the them. It would not be efficient to continually create new LocStreams to reflect this sorting. Instead, the sorting is managed by the application through permutation arrays while keeping the data in place. Locations can become inactive, e.g., if the quality control asserts that observation is invalid. This can be managed again by the application through masks.

A LocStream differs from a Grid in that no topological structure is maintained between the points (e.g. the class contains no information about which point is the neighbor of which other point).

A LocStream is similar to a Mesh consists in that both are collections of irregularly positioned points. However, the two structures differ in that a Mesh has connectivity also: each data point has a set of neighboring data points. There is no requirement that the points in a LocStream have connectivity, in fact there is no requirement that a point has any particular spatial relationship to another.

28.2 Use and Examples

28.2.1 Create a LocStream with user allocated memory

The following is an example of creating a LocStream object. After creation, key data is added, and a Field is created to hold data (temperature) at each location.

```
!-------------------------------------------------------------------
! Allocate and set example location information
!-------------------------------------------------------------------
allocate(lon(numLocationsOnThisPet))
allocate(lat(numLocationsOnThisPet))

do i=1,numLocationsOnThisPet
   lon(i)=360.0/numLocationsOnThisPet
   lat(i)=0.0
enddo

!-------------------------------------------------------------------
! Allocate and set example Field data
!-------------------------------------------------------------------
allocate(temperature(numLocationsOnThisPet))

do i=1,numLocationsOnThisPet
   temperature(i)=90.0
enddo


!-------------------------------------------------------------------
! Create the LocStream:  Allocate space for the LocStream object,
! define the number and distribution of the locations.
!-------------------------------------------------------------------
locstream=ESMF_LocStreamCreate(name="Equatorial Measurements",   &
                               localCount=numLocationsOnThisPet, &
                               rc=rc)
```

```
!------------------------------------------------------------------
! Add key data, referencing a user data pointer. By changing the
! datacopyflag to ESMF_DATACOPY_VALUE an internally allocated copy of the
! user data may also be set.
!------------------------------------------------------------------
call ESMF_LocStreamAddKey(locstream,              &
                          keyName="Lat",          &
                          farray=lat,             &
                          datacopyflag=ESMF_DATACOPY_REFERENCE, &
                          keyUnits="Degrees",     &
                          keyLongName="Latitude", rc=rc)

call ESMF_LocStreamAddKey(locstream,              &
                          keyName="Lon",          &
                          farray=lon,             &
                          datacopyflag=ESMF_DATACOPY_REFERENCE, &
                          keyUnits="Degrees",     &
                          keyLongName="Longitude", rc=rc)

!------------------------------------------------------------------
! Create a Field on the Location Stream. In this case the
! Field is created from a user array, but any of the other
! Field create methods (e.g. from ArraySpec) would also apply.
!------------------------------------------------------------------
field_temperature=ESMF_FieldCreate(locstream,   &
                                temperature, &
                                name="temperature", &
                                rc=rc)
```

28.2.2 Create a LocStream with internally allocated memory

The following is an example of creating a LocStream object. After creation, key data is internally allocated, the pointer is retrieved, and the data is set. A Field is also created on the LocStream to hold data (temperature) at each location.

```
!------------------------------------------------------------------
! Allocate and set example Field data
!------------------------------------------------------------------
allocate(temperature(numLocationsOnThisPet))

do i=1,numLocationsOnThisPet
   temperature(i)=80.0
enddo


!------------------------------------------------------------------
! Create the LocStream:  Allocate space for the LocStream object,
! define the number and distribution of the locations.
!------------------------------------------------------------------
locstream=ESMF_LocStreamCreate(name="Equatorial Measurements", &
                               localCount=numLocationsOnThisPet, &
                               rc=rc)
```

```
!-------------------------------------------------------------------
! Add key data (internally allocating memory).
!-------------------------------------------------------------------
call ESMF_LocStreamAddKey(locstream,                    &
                          keyName="Lat",                &
                          KeyTypeKind=ESMF_TYPEKIND_R8, &
                          keyUnits="Degrees",           &
                          keyLongName="Latitude", rc=rc)

call ESMF_LocStreamAddKey(locstream,                    &
                          keyName="Lon",                &
                          KeyTypeKind=ESMF_TYPEKIND_R8, &
                          keyUnits="Degrees",           &
                          keyLongName="Longitude", rc=rc)


!-------------------------------------------------------------------
! Get key data.
!-------------------------------------------------------------------
call ESMF_LocStreamGetKey(locstream,                    &
                          localDE=0,                    &
                          keyName="Lat",                &
                          farray=lat,                   &
                          rc=rc)

call ESMF_LocStreamGetKey(locstream,                    &
                          localDE=0,                    &
                          keyName="Lon",                &
                          farray=lon,                   &
                          rc=rc)

!-------------------------------------------------------------------
! Set key data.
!-------------------------------------------------------------------
do i=1,numLocationsOnThisPet
   lon(i)=360.0/numLocationsOnThisPet
   lat(i)=0.0
enddo


!-------------------------------------------------------------------
! Create a Field on the Location Stream. In this case the
! Field is created from a user array, but any of the other
! Field create methods (e.g. from ArraySpec) would also apply.
!-------------------------------------------------------------------
field_temperature=ESMF_FieldCreate(locstream,   &
                              temperature, &
                              name="temperature", &
                              rc=rc)
```

28.2.3 Create a LocStream from a background Grid

The following is an example of creating a LocStream object from another LocStream object using a background Grid. The new LocStream contains the data present in the old LocStream, but is redistributed so that entries with a given set of coordinates are on the same PET as the piece of the background Grid which contains those coordinates.

```
!-------------------------------------------------------------------
! Create the LocStream:  Allocate space for the LocStream object,
! define the number and distribution of the locations.
!-------------------------------------------------------------------
locstream=ESMF_LocStreamCreate(name="Equatorial Measurements", &
                               localCount=numLocationsOnThisPet, &
                               rc=rc)
!-------------------------------------------------------------------
! Add key data (internally allocating memory).
!-------------------------------------------------------------------
call ESMF_LocStreamAddKey(locstream,                   &
                          keyName="Lon",               &
                          KeyTypeKind=ESMF_TYPEKIND_R8, &
                          keyUnits="Degrees",          &
                          keyLongName="Longitude", rc=rc)

call ESMF_LocStreamAddKey(locstream,                   &
                          keyName="Lat",               &
                          KeyTypeKind=ESMF_TYPEKIND_R8, &
                          keyUnits="Degrees",          &
                          keyLongName="Latitude", rc=rc)


!-------------------------------------------------------------------
! Get Fortran arrays which hold the key data, so that it can be set.
! Using localDE=0, because the locstream was created with 1 DE per PET.
!-------------------------------------------------------------------
call ESMF_LocStreamGetKey(locstream,                   &
                          localDE=0,                   &
                          keyName="Lon",               &
                          farray=lon,                  &
                          rc=rc)

call ESMF_LocStreamGetKey(locstream,                   &
                          localDE=0,                   &
                          keyName="Lat",               &
                          farray=lat,                  &
                          rc=rc)

!-------------------------------------------------------------------
! Set the longitude and latitude coordinates of the points in the
! LocStream. Each PET contains points scattered around the equator.
!-------------------------------------------------------------------
do i=1,numLocationsOnThisPet
   lon(i)=0.5+REAL(i-1)*360.0/numLocationsOnThisPet
   lat(i)=0.0
enddo

!-------------------------------------------------------------------
! Create a Grid to use as the background. The Grid is
```

```
! GridLonSize by GridLatSize with the default distribution
! (The first dimension split across the PETs). The coordinate range
! is  0 to 360 in longitude and -90 to 90 in latitude. Note that we
! use indexflag=ESMF_INDEX_GLOBAL for the Grid creation. At this time
! this is required for a Grid to be usable as a background Grid.
! Note that here the points are treated as cartesian.
!-------------------------------------------------------------------
grid=ESMF_GridCreateNoPeriDim(maxIndex=(/GridLonSize,GridLatSize/), &
                              coordSys=ESMF_COORDSYS_CART, &
                              indexflag=ESMF_INDEX_GLOBAL, &
                              rc=rc)

!-------------------------------------------------------------------
! Allocate the corner stagger location in which to put the coordinates.
! (The corner stagger must be used for the Grid to be usable as a
!  background Grid.)
!-------------------------------------------------------------------
call ESMF_GridAddCoord(grid, staggerloc=ESMF_STAGGERLOC_CORNER, rc=rc)


!-------------------------------------------------------------------
! Get access to the Fortran array pointers that hold the Grid
! coordinate information and then set the coordinates to be uniformly
! distributed around the globe.
!-------------------------------------------------------------------
call ESMF_GridGetCoord(grid, localDE=0,                       &
                       staggerLoc=ESMF_STAGGERLOC_CORNER,     &
                       coordDim=1, computationalLBound=clbnd, &
                       computationalUBound=cubnd,             &
                       farrayPtr=farrayPtrLonC, rc=rc)

call ESMF_GridGetCoord(grid, localDE=0,                       &
                      staggerLoc=ESMF_STAGGERLOC_CORNER,      &
                       coordDim=2, farrayPtr=farrayPtrLatC, rc=rc)

do i1=clbnd(1),cubnd(1)
do i2=clbnd(2),cubnd(2)
   ! Set Grid longitude coordinates as 0 to 360
   farrayPtrLonC(i1,i2) = REAL(i1-1)*360.0/REAL(GridLonSize)

   ! Set Grid latitude coordinates as -90 to 90
   farrayPtrLatC(i1,i2) = -90. + REAL(i2-1)*180.0/REAL(GridLatSize) + &
                                   0.5*180.0/REAL(GridLatSize)
enddo
enddo


!-------------------------------------------------------------------
! Create newLocstream on the background Grid using the
! "Lon" and "Lat" keys as the coordinates for the entries in
! locstream. The entries in newLocstream with coordinates (lon,lat)
! are on the same PET as the piece of grid which contains (lon,lat).
!-------------------------------------------------------------------
newLocstream=ESMF_LocStreamCreate(locstream, coordKeyNames="Lon:Lat", &
               background=grid, rc=rc)
```

```
!----------------------------------------------------------------------
! A Field can now be created on newLocstream and
! ESMF_FieldRedist() can be used to move data between Fields built
! on locstream and Fields built on newLocstream.
!----------------------------------------------------------------------
```

28.3 Class API

28.3.1 ESMF_LocStreamAssignment(=) - LocStream assignment

INTERFACE:

```
  interface assignment(=)
  locstream1 = locstream2
```

ARGUMENTS:

```
  type(ESMF_LocStream) :: locstream1
  type(ESMF_LocStream) :: locstream2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign locstream1 as an alias to the same ESMF LocStream object in memory as locstream2. If locstream2 is invalid, then locstream1 will be equally invalid after the assignment.
The arguments are:

locstream1 The `ESMF_LocStream` object on the left hand side of the assignment.

locstream2 The `ESMF_LocStream` object on the right hand side of the assignment.

28.3.2 ESMF_LocStreamOperator(==) - LocStream equality operator

INTERFACE:

```
interface operator(==)
  if (locstream1 == locstream2) then ... endif
            OR
  result = (locstream1 == locstream2)
```

RETURN VALUE:

```
  logical :: result
```

ARGUMENTS:

```
  type(ESMF_LocStream), intent(in) :: locstream1
  type(ESMF_LocStream), intent(in) :: locstream2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether locstream1 and locstream2 are valid aliases to the same ESMF LocStream object in memory. For a more general comparison of two ESMF LocStreams, going beyond the simple alias test, the ESMF_LocStreamMatch() function (not yet implemented) must be used.
The arguments are:

locstream1 The `ESMF_LocStream` object on the left hand side of the equality operation.

locstream2 The `ESMF_LocStream` object on the right hand side of the equality operation.

28.3.3 ESMF_LocStreamOperator(/=) - LocStream not equal operator

INTERFACE:

```
interface operator(/=)
  if (locstream1 /= locstream2) then ... endif
            OR
  result = (locstream1 /= locstream2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream1
type(ESMF_LocStream), intent(in) :: locstream2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether locstream1 and locstream2 are *not* valid aliases to the same ESMF LocStream object in memory. For a more general comparison of two ESMF LocStreams, going beyond the simple alias test, the ESMF_LocStreamMatch() function (not yet implemented) must be used.
The arguments are:

locstream1 The `ESMF_LocStream` object on the left hand side of the non-equality operation.

locstream2 The `ESMF_LocStream` object on the right hand side of the non-equality operation.

28.3.4 ESMF_LocStreamAddKey - Add a key Array and allocate the internal memory

INTERFACE:

```
! Private name; call using ESMF_LocStreamAddKey()
subroutine ESMF_LocStreamAddKeyAlloc(locstream, keyName, keyTypeKind, &
             keyUnits, keyLongName, rc)
```

ARGUMENTS:

```
type(ESMF_Locstream),      intent(in)             :: locstream
character (len=*),         intent(in)             :: keyName
type(ESMF_TypeKind_Flag),  intent(in),  optional  :: keyTypeKind
character (len=*),         intent(in),  optional  :: keyUnits
character (len=*),         intent(in),  optional  :: keyLongName
integer,                   intent(out), optional  :: rc
```

DESCRIPTION:

Add a key to a locstream. Once a key has been added its internal data can be retrieved and used to set key values. The arguments are:

locstream The `ESMF_LocStream` object to add key to.

keyName The name of the key to add.

[keyTypeKind] The type/kind of the key data. If not specified then the type/kind will default to 8 byte reals.

[keyUnits] The units of the key data. If not specified, then the item remains blank.

[keyLongName] The long name of the key data. If not specified, then the item remains blank.

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

28.3.5 ESMF_LocStreamAddKey - Add a key Array

INTERFACE:

```
! Private name; call using ESMF_LocStreamAddKey()
subroutine ESMF_LocStreamAddKeyArray(locstream, keyName, keyArray, &
             destroyKey, keyUnits, keyLongName, rc)
```

ARGUMENTS:

```
type(ESMF_Locstream), intent(in)              :: locstream
character (len=*),    intent(in)              :: keyName
type(ESMF_Array),     intent(in)              :: keyArray
logical,              intent(in),  optional   :: destroyKey
character (len=*),    intent(in),  optional   :: keyUnits
character (len=*),    intent(in),  optional   :: keyLongName
integer,              intent(out), optional   :: rc
```

DESCRIPTION:

Add a key to a locstream. Once a key has been added its internal data can be retrieved and used to set key values. The arguments are:

locstream The `ESMF_LocStream` object to add key to.

keyName The name of the key to add.

keyArray An ESMF Array which contains the key data

[destroyKey] if .true. destroy this key array when the locstream is destroyed. Defaults to .false.

[keyUnits] The units of the key data. If not specified, then the item remains blank.

[keyLongName] The long name of the key data. If not specified, then the item remains blank.

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

28.3.6 ESMF_LocStreamAddKey - Add a key Array created around user memory

INTERFACE:

```
  ! Private name; call using ESMF_LocStreamAddKey()
   subroutine ESMF_LocStreamAddKeyI4(locstream, keyName, farray, &
                datacopyflag, keyUnits, keyLongName, rc)
```

ARGUMENTS:

```
      type(ESMF_Locstream), intent(in) :: locstream
      character (len=*), intent(in) :: keyName
      <farray>
      type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
      character (len=*), intent(in), optional :: keyUnits
      character (len=*), intent(in), optional :: keyLongName
      integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a key to a locstream. Once a key has been added its internal data can be retrieved and used to set key values. Supported values for <farray> are:

integer(ESMF_KIND_I4), intent(in) :: farray(:)

real(ESMF_KIND_R4), intent(in) :: farray(:)

real(ESMF_KIND_R8), intent(in) :: farray(:)

The arguments are:

locstream The `ESMF_LocStream` object to add key to.

keyName The name of the key to add.

farray Valid native Fortran array, i.e. memory must be associated with the actual argument. The type/kind/rank information of `farray` will be used to set the key Array's properties accordingly.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by `farray` directly or will copy the data from `farray` into a new memory allocation. Valid options are `ESMF_DATACOPY_REFERENCE` (default) or `ESMF_DATACOPY_VALUE`. Depending on the specific situation the `ESMF_DATACOPY_REFERENCE` option may be unsafe when specifying an array slice for `farray`.

[keyUnits] The units of the key data. If not specified, then the item remains blank.

[keyLongName] The long name of the key data. If not specified, then the item remains blank.

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

28.3.7 ESMF_LocStreamCreate - Create a new LocStream by projecting onto a Grid

INTERFACE:

```
      ! Private name; call using ESMF_LocStreamCreate()
      function ESMF_LocStreamCreateByBkgGrid(locstream, name, &
                 coordKeyNames, background, maskValues, &
                 unmappedaction, rc)
```

RETURN VALUE:

```
type(ESMF_LocStream) :: ESMF_LocStreamCreateByBkgGrid
```

ARGUMENTS:

```
type(ESMF_LocStream),            intent(in)             :: locstream
character (len=*),               intent(in),  optional :: name
character (len=*),               intent(in)             :: coordKeyNames
type(ESMF_Grid),                 intent(in)             :: background
integer(ESMF_KIND_I4),           intent(in),  optional :: maskValues(:)
type(ESMF_UnmappedAction_Flag), intent(in),  optional :: unmappedaction
integer,                         intent(out), optional :: rc
```

DESCRIPTION:

Create an location stream from an existing one in accordance with the distribution of the background Grid. The entries in the new location stream are redistributed, so that they lie on the same PET as the piece of Grid which contains the coordinates of the entries. The coordinates of the entries are the data in the keys named by `coordKeyNames`. To copy data in Fields or FieldBundles built on `locstream` to the new one simply use `ESMF_FieldRedist()` or `ESMF_FieldBundleRedist()`.
The arguments are:

locstream Location stream from which the new location stream is to be created

[name] Name of the resulting location stream

coordKeyNames Names of the keys used to determine the link to background Grid. The first key in this list matches up with the first coordinate of the Grid, the second key in this list matches up with the second coordinate of the Grid, and so on. The key names should be separated by the : character.

background Background Grid which determines the distribution of the entries in the new location stream. The background Grid needs to have the same number of dimensions as the number of keys in `coordKeyNames`. Note also that this subroutine uses the corner stagger location in the Grid for determining where a point lies, because this is the stagger location which fully contains the cell. A Grid must have coordinate data in this stagger location to be used in this subroutine. For a 2D Grid this stagger location is ESMF_STAGGERLOC_CORNER for a 3D Grid this stagger location is ESMF_STAGGERLOC_CORNER_VFACE. Note that currently the background Grid also needs to have been created with indexflag=ESMF_INDEX_GLOBAL to be usable here.

[maskValues] List of values that indicate a background grid point should be masked out. If not specified, no masking will occur.

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

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

28.3.8 ESMF_LocStreamCreate - Create a new LocStream by projecting onto a Mesh

INTERFACE:

```
! Private name; call using ESMF_LocStreamCreate()
function ESMF_LocStreamCreateByBkgMesh(locstream, name, &
           coordKeyNames, background, unmappedaction, rc)
```

RETURN VALUE:

```
type(ESMF_LocStream) :: ESMF_LocStreamCreateByBkgMesh
```

ARGUMENTS:

```
type(ESMF_LocStream),           intent(in)           :: locstream
character (len=*),              intent(in), optional :: name
character (len=*),              intent(in)           :: coordKeyNames
type(ESMF_Mesh),                intent(in)           :: background
type(ESMF_UnmappedAction_Flag), intent(in), optional :: unmappedaction
integer,                        intent(out),optional :: rc
```

DESCRIPTION:

Create an location stream from an existing one in accordance with the distribution of the background Mesh. The entries in the new location stream are redistributed, so that they lie on the same PET as the piece of Mesh which contains the coordinates of the entries. The coordinates of the entries are the data in the keys named by `coordKeyNames`. To copy data in Fields or FieldBundles built on `locstream` to the new one simply use `ESMF_FieldRedist()` or `ESMF_FieldBundleRedist()`.
The arguments are:

locstream Location stream from which the new location stream is to be created

[name] Name of the resulting location stream

coordKeyNames Names of the keys used to determine the link to background Mesh. The first key in this list matches up with the first coordinate of the Mesh, the second key in this list matches up with the second coordinate of the Mesh, and so on. The key names should be separated by the : character.

background Background Mesh which determines the distribution of entries in the new locatiion stream. The Mesh must have the same spatial dimension as the number of keys in `coordKeyNames`.

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

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

28.3.9 ESMF_LocStreamCreate - Create a new LocStream from a distgrid

INTERFACE:

```
! Private name: call using ESMF_LocStreamCreate()
function ESMF_LocStreamCreateFromDG(name, distgrid, &
           destroyDistgrid, indexflag, rc )
```

RETURN VALUE:

```
type(ESMF_LocStream) :: ESMF_LocStreamCreateFromDG
```

ARGUMENTS:

```
character (len=*),     intent(in),  optional :: name
type(ESMF_DistGrid),   intent(in)            :: distgrid
logical,               intent(in), optional  :: destroyDistgrid
type(ESMF_Index_Flag),  intent(in), optional  :: indexflag
integer,               intent(out), optional :: rc
```

DESCRIPTION:

Allocates memory for a new ESMF_LocStream object, constructs its internal derived types.
The arguments are:

name Name of the location stream

distgrid Distgrid specifying size and distribution. Only 1D distgrids are allowed.

[destroyDistgrid] If .true. the locstream is responsible for destroying the distgrid. Defaults to .false.

[indexflag] Flag that indicates how the DE-local indices are to be defined. Defaults to ESMF_INDEX_DELOCAL, which indicates that the index range on each DE starts at 1. See Section 9.24 for the full range of options.

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

28.3.10 ESMF_LocStreamCreate - Create a new LocStream from an irregular distribution

INTERFACE:

```
! Private name: call using ESMF_LocStreamCreate()
function ESMF_LocStreamCreateIrreg(name, minIndex, countsPerDE, &
            indexflag, rc)
```

RETURN VALUE:

```
type(ESMF_LocStream) :: ESMF_LocStreamCreateIrreg
```

ARGUMENTS:

```
character (len=*), intent(in), optional        :: name
integer, intent(in), optional                  :: minIndex
integer, intent(in)                            :: countsPerDE(:)
type(ESMF_Index_Flag), intent(in), optional     :: indexflag
integer, intent(out), optional                 :: rc
```

DESCRIPTION:

Allocates memory for a new ESMF_LocStream object, constructs its internal derived types. The ESMF_DistGrid is set up, indicating how the LocStream is distributed.
The arguments are:

name Name of the location stream

[minIndex] Number to start the index ranges at. If not present, defaults to 1.

countsPerDE This array specifies the number of locations per DE.

[indexflag] Flag that indicates how the DE-local indices are to be defined. Defaults to ESMF_INDEX_DELOCAL, which indicates that the index range on each DE starts at 1. See Section 9.24 for the full range of options.

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

28.3.11 ESMF_LocStreamCreate - Create a new LocStream from a local count

INTERFACE:

```
! Private name: call using ESMF_LocStreamCreate()
function ESMF_LocStreamCreateFromLocal(name, localCount, indexflag, rc)
```

RETURN VALUE:

```
type(ESMF_LocStream) :: ESMF_LocStreamCreateFromLocal
```

ARGUMENTS:

```
character (len=*), intent(in), optional         :: name
integer, intent(in)                             :: localCount
type(ESMF_Index_Flag), intent(in), optional      :: indexflag
integer, intent(out), optional                  :: rc
```

DESCRIPTION:

Allocates memory for a new `ESMF_LocStream` object, constructs its internal derived types. The `ESMF_DistGrid` is set up, indicating how the LocStream is distributed.
The arguments are:

name Name of the location stream

localCount Number of grid cells to be distributed to this DE.

[indexflag] Flag that indicates how the DE-local indices are to be defined. Defaults to `ESMF_INDEX_DELOCAL`, which indicates that the index range on each DE starts at 1. See Section 9.24 for the full range of options.

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

28.3.12 ESMF_LocStreamCreate - Create a new LocStream using a regular distribution

INTERFACE:

```
! Private name: call using ESMF_LocStreamCreate()
function ESMF_LocStreamCreateReg(name, &
           regDecomp, decompFlag, minIndex, maxIndex, indexflag, rc )
```

RETURN VALUE:

```
type(ESMF_LocStream) :: ESMF_LocStreamCreateReg
```

ARGUMENTS:

```
character (len=*),     intent(in),  optional  :: name
integer,               intent(in),  optional  :: regDecomp
type(ESMF_Decomp_Flag), intent(in),  optional  :: decompflag
integer,               intent(in),  optional  :: minIndex
integer,               intent(in)             :: maxIndex
type(ESMF_Index_Flag),  intent(in),  optional  :: indexflag
integer,               intent(out), optional  :: rc
```

DESCRIPTION:

Allocates memory for a new `ESMF_LocStream` object, constructs its internal derived types. The `ESMF_DistGrid` is set up, indicating how the LocStream is distributed. at a later time.
The arguments are:

name Name of the location stream

[regDecomp] Specify into how many chunks to divide the locations. If not specified, defaults to the number of PETs.

[decompFlag] Specify what to do with leftover locations after division. If not specified, defaults to `ESMF_DECOMP_BALANCED`. Please see Section 9.13 for a full description of the possible options.

[[minIndex]] The minimum index across all PETs. If not set defaults to 1.

maxIndex The maximum index across all PETs.

[indexflag] Flag that indicates how the DE-local indices are to be defined. Defaults to `ESMF_INDEX_DELOCAL`, which indicates that the index range on each DE starts at 1. See Section 9.24 for the full range of options.

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

28.3.13 ESMF_LocStreamDestroy - Release resources associated with a LocStream

INTERFACE:

```
      subroutine ESMF_LocStreamDestroy(locstream, keywordenforcer, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Deallocate an `ESMF_LocStream` object and appropriate internal structures.
The arguments are:

locstream locstream to destroy

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

28.3.14 ESMF_LocStreamGet - Return info associated with a LocStream

INTERFACE:

```
   ! Private name; call using ESMF_LocStreamGet()
   subroutine ESMF_LocStreamGetDefault(locstream, distgrid, keyCount, &
                keyNames, localDECount, indexflag, name, rc)
```

ARGUMENTS:

```
type(ESMF_Locstream), intent(in)             :: locstream
type(ESMF_DistGrid), intent(out),  optional  :: distgrid
integer, intent(out),optional                :: keyCount
character(len=ESMF_MAXSTR),optional          :: keyNames(:)
integer, intent(out),optional                :: localDECount
type(ESMF_Index_Flag), intent(out), optional  :: indexflag
character(len=*), intent(out),     optional  :: name
integer, intent(out), optional               :: rc
```

DESCRIPTION:

Query an `ESMF_LocStream` for various information. All arguments after the `locstream` are optional.
The arguments are:

locstream The `ESMF_LocStream` object to query.

[distgrid] The `ESMF_DistGrid` object that descibes

[keyCount] Number of keys in the `locstream`.

[keyNames] The names of the keys in the `locstream`. Keynames should be an array of character strings. The character strings should be of length ESMF_MAXSTR and the array's length should be at least keyCount.

[localDECount] Number of DEs on this PET in the `locstream`.

[indexflag] The indexflag for this indexflag.

[name] Name of queried item.

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

28.3.15 ESMF_LocStreamGetKey - Get an Array associated with a key

INTERFACE:

```
! Private name; call using ESMF_LocStreamGetKey()
subroutine ESMF_LocStreamGetKeyArray(locstream, keyName, keyArray, rc)
```

ARGUMENTS:

```
type(ESMF_Locstream), intent(in)            :: locstream
character (len=*),    intent(in)            :: keyName
type(ESMF_Array),     intent(out)            :: keyArray
integer, intent(out), optional :: rc
```

DESCRIPTION:

Get ESMF Array associated with key.
The arguments are:

locstream The `ESMF_LocStream` object to get key from.

keyName The name of the key to get.

keyArray Array associated with key.

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

28.3.16 ESMF_LocStreamGetKey - Get the bounds of a key Array

INTERFACE:

```
  ! Private name; call using ESMF_LocStreamGetKey()
      subroutine ESMF_LocStreamGetKeyBounds(locstream, localDE, keyName, &
          exclusiveLBound, exclusiveUBound, exclusiveCount,     &
          computationalLBound, computationalUBound, computationalCount,     &
          totalLBound, totalUBound, totalCount,     &
          rc)
```

ARGUMENTS:

```
      type(ESMF_LocStream),   intent(in)       :: locstream
      integer,                intent(in)                  :: localDE
      character (len=*),   intent(in)                :: keyName
      integer,                intent(out), optional :: exclusiveLBound
      integer,                intent(out), optional :: exclusiveUBound
      integer,                intent(out), optional :: exclusiveCount
      integer,                intent(out), optional :: computationalLBound
      integer,                intent(out), optional :: computationalUBound
      integer,                intent(out), optional :: computationalCount
      integer,                intent(out), optional :: totalLBound
      integer,                intent(out), optional :: totalUBound
      integer,                intent(out), optional :: totalCount
      integer, intent(out), optional :: rc
```

DESCRIPTION:

This method gets the bounds of a localDE for a locstream.
The arguments are:

locstream LocStream to get the information from.

localDE The local DE to get the information for. `[0,..,localDeCount-1]`

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region.

[exclusiveCount] Upon return this holds the number of items in the exclusive region (i.e. `exclusiveUBound-exclusiveLBound+1`). `exclusiveCount`.

[computationalLBound] Upon return this holds the lower bounds of the computational region.

[computationalUBound] Upon return this holds the upper bounds of the computational region.

[computationalCount] Upon return this holds the number of items in the computational region (i.e. `computationalUBound-computationalLBound+1`).

[totalLBound] Upon return this holds the lower bounds of the total region.

[totalUBound] Upon return this holds the upper bounds of the total region.

[totalCount] Upon return this holds the number of items in the total region (i.e. `totalUBound-totalLBound+1`).

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

28.3.17 ESMF_LocStreamGetKey - Get info associated with a key

INTERFACE:

```
! Private name; call using ESMF_LocStreamGetKey()
subroutine ESMF_LocStreamGetKeyInfo(locstream, keyName, keyUnits, &
              keyLongName, typekind, rc)
```

ARGUMENTS:

```
  type(ESMF_Locstream), intent(in)              :: locstream
  character (len=*),    intent(in)              :: keyName
  character (len=*),    intent(out), optional   :: keyUnits
  character (len=*),    intent(out), optional   :: keyLongName
  type(ESMF_TypeKind_Flag), intent(out), optional    :: typekind
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Get ESMF Array associated with key.
The arguments are:

locstream The `ESMF_LocStream` object to get key from.

keyName The name of the key to get.

[keyUnits] The units of the key data. If not specified, then the item remains blank.

[keyLongName] The long name of the key data. If not specified, then the item remains blank.

[typekind] The typekind of the key data

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

28.3.18 ESMF_LocStreamGetKey - Get a pointer to key values

INTERFACE:

```
! Private name; call using ESMF_LocStreamGetKey()
     subroutine ESMF_LocStreamGetKeyI4(locstream, localDE, keyName,    &
         exclusiveLBound, exclusiveUBound, exclusiveCount,             &
         computationalLBound, computationalUBound, computationalCount, &
         totalLBound, totalUBound, totalCount,                         &
         farray, datacopyflag, rc)
```

ARGUMENTS:

```
     type(ESMF_LocStream), intent(in) :: locstream
     integer, intent(in) :: localDE
     character (len=*),    intent(in)              :: keyName
     integer,                intent(out), optional :: exclusiveLBound
     integer,                intent(out), optional :: exclusiveUBound
     integer,                intent(out), optional :: exclusiveCount
     integer,                intent(out), optional :: computationalLBound
     integer,                intent(out), optional :: computationalUBound
     integer,                intent(out), optional :: computationalCount
```

```
integer,                intent(out), optional :: totalLBound
integer,                intent(out), optional :: totalUBound
integer,                intent(out), optional :: totalCount
<farray>
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

This method gets a Fortran pointer to the piece of memory which holds the key data for a particular key on the given local DE. This is useful, for example, for setting the key values in a LocStream, or for reading the values. Supported values for <farray> are:

integer(ESMF_KIND_I4), pointer :: farray(:)

real(ESMF_KIND_R4), pointer :: farray(:)

real(ESMF_KIND_R8), pointer :: farray(:)

The arguments are:

locstream LocStream to get the information from.

localDE The local DE to get the information for. `[0,..,localDeCount-1]`

keyName The key to get the information from.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region.

[exclusiveCount] Upon return this holds the number of items in the exclusive region (i.e. `exclusiveUBound-exclusiveLBound+1`). `exclusiveCount`.

[computationalLBound] Upon return this holds the lower bounds of the computational region.

[computationalUBound] Upon return this holds the upper bounds of the computational region.

[computationalCount] Upon return this holds the number of items in the computational region (i.e. `computationalUBound-computationalLBound+1`).

[totalLBound] Upon return this holds the lower bounds of the total region.

[totalUBound] Upon return this holds the upper bounds of the total region.

[totalCount] Upon return this holds the number of items in the total region (i.e. `totalUBound-totalLBound+1`).

farray The pointer to the coordinate data.

[datacopyflag] If not specified, default to `ESMF_DATACOPY_REFERENCE`, in this case farray is a reference to the data in the Grid coordinate arrays. Please see Section 9.12 for further description and a list of valid values.

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

28.3.19 ESMF_LocStreamGet - Get the local bounds of a LocStream

INTERFACE:

```
! Private name; call using ESMF_LocStreamGet()
    subroutine ESMF_LocStreamGetBounds(locstream, localDE, &
        exclusiveLBound, exclusiveUBound, exclusiveCount,     &
        computationalLBound, computationalUBound, computationalCount,     &
        rc)
```

ARGUMENTS:

```
type(ESMF_LocStream),   intent(in) :: locstream
integer,                intent(in) :: localDE
integer,                intent(out), optional :: exclusiveLBound
integer,                intent(out), optional :: exclusiveUBound
integer,                intent(out), optional :: exclusiveCount
integer,                intent(out), optional :: computationalLBound
integer,                intent(out), optional :: computationalUBound
integer,                intent(out), optional :: computationalCount
integer, intent(out), optional :: rc
```

DESCRIPTION:

This method gets the bounds of a localDE for a locstream.
The arguments are:

locstream LocStream to get the information from.

localDE The local DE to get the information for. `[0,..,localDeCount-1]`

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region.

[exclusiveCount] Upon return this holds the number of items in the exclusive region
(i.e. `exclusiveUBound-exclusiveLBound+1`). `exclusiveCount`.

[computationalLBound] Upon return this holds the lower bounds of the computational region.

[computationalUBound] Upon return this holds the upper bounds of the computational region.

[computationalCount] Upon return this holds the number of items in the computational region
(i.e. `computationalUBound-computationalLBound+1`). `computationalCount`.

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

28.3.20 ESMF_LocStreamPrint - Print the contents of a LocStream

INTERFACE:

```
subroutine ESMF_LocStreamPrint(locstream, options, rc)
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(inout) :: locstream
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the `locstream` to `stdout`. This subroutine goes through the internal data members of a locstream data type and prints information of each data member.
The arguments are:

locstream

[options] Print options are not yet supported.

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

28.3.21 ESMF_LocStreamValidate - Check validity of a LocStream

INTERFACE:

```
subroutine ESMF_LocStreamValidate(locstream, options, rc)
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(inout) :: locstream
character (len = *), intent(in), optional :: options
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `locstream` is internally consistent. Currently this method determines if the `locstream` is uninitialized or already destroyed.
The method returns an error code if problems are found.
The arguments are:

locstream `ESMF_LocStream` to validate.

[options] Validation options are not yet supported.

[rc] Return code; equals `ESMF_SUCCESS` if the `locstream` is valid.

29 Mesh Class

29.1 Description

Unstructured grids are commonly used in the computational solution of partial differential equations. These are especially useful for problems that involve complex geometry, where using the less flexible structured grids can result in grid representation of regions where no computation is needed. Finite element and finite volume methods map naturally to unstructured grids and are used commonly in hydrology, ocean modeling, and many other applications.
In order to provide support for application codes using unstructured grids, the ESMF library provides a class for representing unstructured grids called the **Mesh**. Fields can be created on a Mesh to hold data. Fields created on a Mesh can also be used as either the source or destination or both of an interpolaton (i.e. an `ESMF_FieldRegridStore()` call) which allows data to be moved between unstructured grids. This section describes the Mesh and how to create and use them in ESMF.

29.1.1 Mesh representation in ESMF

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

29.1.2 Supported Meshes

The range of Meshes supported by ESMF are defined by several factors: dimension, element types, and distribution.
ESMF currently only supports Meshes whose number of coordinate dimensions (spatial dimension) is 2 or 3. The dimension of the elements in a Mesh (parametric dimension) must be less than or equal to the spatial dimension, but also must be either 2 or 3. This means that a Mesh may be either 2D elements in 2D space, 3D elements in 3D space, or a manifold constructed of 2D elements embedded in 3D space.
ESMF currently supports two types of elements for each Mesh parametric dimension. For a parametric dimension of 2, the supported element types are triangles or quadralaterals. For a parametric dimension of 3, the supported element types are tetrahedrons and hexahedrons. See Section 29.2.1 for diagrams of these. The Mesh supports any combination of element types within a particular dimension, but types from different dimensions may not be mixed. For example, a Mesh cannot be constructed of both quadralaterals and tetrahedra.
ESMF currently only supports distributions where every node on a PET must be a part of an element on that PET. In other words, there must not be nodes without a corresponding element on any PET.

29.2 Constants

29.2.1 ESMF_MESHELEMTYPE

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

```
                  3                          4 ---------- 3
                 / \                         |            |
                /   \                        |            |
               /     \                       |            |
              /       \                      |            |
             /         \                     |            |
            1 --------- 2                    1 ---------- 2

        ESMF_MESHELEMTYPE_TRI             ESMF_MESHELEMTYPE_QUAD

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

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

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

```
                3                          8---------------7
               /|\                        /|              /|
              / | \                      / |             / |
             /  |  \                    /  |            /  |
            /   |   \                  /   |           /   |
           /    |    \                5---------------6    |
          4-----|-----2               |    |          |    |
           \    |    /                |    4----------|----3
            \   |   /                 |   /           |   /
             \  |  /                  |  /            |  /
              \ | /                   | /             | /
```

```
              \|/                          |/              |/
               1                           1---------------2

       ESMF_MESHELEMTYPE_TETRA            ESMF_MESHELEMTYPE_HEX

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

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

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

29.2.2 ESMF_FILEFORMAT

DESCRIPTION:
This option is used by `ESMF_MeshCreate` to specify the type of the input grid file. See section 29.3.5 for more detailed description of the two file formats.
The type of this flag is:
`type(ESMF_FileFormat_Flag)`
The valid values are:

ESMF_FILEFORMAT_SCRIP SCRIP format grid file. The SCRIP format is the format accepted by the SCRIP regridding tool [13]. For Mesh creation, files of this type only work when the `grid_rank` in the file is equal to 1.

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

29.3 Use and Examples

This section describes the use of the ESMF Mesh class. It starts with an explanation and examples of creating a Mesh and then goes through other Mesh methods. This set of sections covers the use of the Mesh class interfaces, for further detail which applies to using a Field specifically on created on a Mesh, please see Section 22.3.18.

29.3.1 Mesh creation

To create a Mesh we need to set some properties of the Mesh as a whole, some properties of each node in the mesh and then some properties of each element which connects the nodes.
For the Mesh as a whole we set its parametric dimension (`parametricDim`) and spatial dimension (`spatialDim`). The parametric dimension of a Mesh is the dimension of the topology of the Mesh, this can be thought of as the dimension of the elements which make up the Mesh. For example, a Mesh composed of triangles would have a parametric dimension of 2, whereas a Mesh composed of tetrahedra would have a parametric dimension of 3. A Mesh's spatial dimension, on the other hand, is the dimension of the space the Mesh is embedded in, in other words the number of coordinate dimensions needed to describe the location of the nodes making up the Mesh. For example, a Mesh constructed of squares on a plane would have a parametric dimension of 2 and a spatial dimension of 2, whereas if that same Mesh were used to represent the 2D surface of a sphere then the Mesh would still have a parametric dimension of 2, but now its spatial dimension would be 3.
The structure of the per node and element information used to create a Mesh is influenced by the Mesh distribution strategy. The Mesh class is distributed by elements. This means that a node must be present on any PET that contains an element associated with that node, but not on any other PET (a node can't be on a PET without an element ""home"). Since a node may be used by two or more elements located on different PETs, a node may be duplicated on muliple PETs. When a node is duplicated in this manner, one and only one of the PETs that contain the node must "own" the node. The user sets this ownership when they define the nodes during Mesh creation. When a Field is created on a Mesh (i.e. on the Mesh nodes), on each PET the Field is only created on the nodes which are owned by that PET. This means that the size of the Field memory on the PET can be smaller than the number of nodes used to create the

Mesh on that PET. Please see Section 22.3.18 in Field for further explanation and examples of this issue and others in working with Fields on Meshes.

For each node in the Mesh we set three properties: the global id of the node (`nodeIds`), node coordinates (`nodeCoords`), and which PET owns the node (`nodeOwners`). The node id is a unique (across all PETs) integer attached to the particular node. It is used to indicate which nodes are the same when connecting together pieces of the Mesh on different processors. The node coordinates indicate the location of a node in space and are used in the `ESMF_FieldRegrid()` functionality when interpolating. The node owner indicates which PET is in charge of the node. This is used when creating a Field on the Mesh to indicate which PET should contain a Field location for the data.

For each element in the Mesh we set three properties: the global id of the element (`elementIds`), the topology type of the element (`elementTypes`), and which nodes are connected together to form the element (`elementConn`). The element id is a unique (across all PETs) integer attached to the particular element. The element type describes the topology of the element (e.g. a triangle vs. a quadralateral). The range of choices for the topology of the elements in a Mesh are restricted by the Mesh's parametric dimension (e.g. a Mesh can't contain a 2D element like a triangle, when its parametric dimension is 3D), but it can contain any combination of elements appropriate to its dimension. The element connectivity indicates which nodes are to be connected together to form the element. The number of nodes connected together for each element is implied by the elements topology type (`elementTypes`). It is IMPORTANT to note, that the entries in this list are NOT the global ids of the nodes, but are indices into the PET local lists of node info used in the Mesh Create. In other words, the element connectivity isn't specified in terms of the global list of nodes, but instead is specified in terms of the locally described node info. One other important point about connectivities is that the order of the nodes in the connectivity list of an element is important. Please see Section 29.2.1 for diagrams illustrating the correct order of nodes in an element.

Mesh creation may either be performed as a one step process using the full `ESMF_MeshCreate()` call, or may be done in three steps. The three step process starts with a more minimal `ESMF_MeshCreate()` call. It is then followed by the `ESMF_MeshAddNodes()` to specify nodes, and then the `ESMF_MeshAddElements()` call to specify elements. This three step sequence is useful to conserve memory because the node arrays being used for the `ESMF_MeshAddNodes()` call can be deallocated before creating the arrays to be used in the `ESMF_MeshAddElements()` call.

29.3.2 Create a small single PET Mesh in one step

```
 2.0   7 ------- 8 ------- 9
       |         |         |
       |    4    |    5    |
       |         |         |
 1.0   4 ------- 5 ------- 6
       |         |  \   3  |
       |    1    |    \    |
       |         |  2   \  |
 0.0   1 ------- 2 ------- 3

      0.0       1.0        2.0

        Node Id labels at corners
       Element Id labels in centers
       (Everything owned by PET 0)
```

This example is intended to illustrate the creation of a small Mesh on one PET. The reason for starting with a single PET case is so that the user can start to familiarize themselves with the concepts of Mesh creation without the added complication of multiple processors. Later examples illustrate the multiple processor case. This example creates the small 2D Mesh which can be seen in the figure above. Note that this Mesh consists of 9 nodes and 5 elements, where the elements are a mixture of quadralaterals and triangles. The coordinates of the nodes in the Mesh range from 0.0

to 2.0 in both dimensions. The node ids are in the corners of the elements whereas the element ids are in the centers. The following section of code illustrates the creation of this Mesh.

```
! Set number of nodes
numNodes=9

! Allocate and fill the node id array.
allocate(nodeIds(numNodes))
nodeIds=(/1,2,3,4,5,6,7,8,9/)

! Allocate and fill node coordinate array.
! Since this is a 2D Mesh the size is 2x the
! number of nodes.
allocate(nodeCoords(2*numNodes))
nodeCoords=(/0.0,0.0, & ! node id 1
             1.0,0.0, & ! node id 2
             2.0,0.0, & ! node id 3
             0.0,1.0, & ! node id 4
             1.0,1.0, & ! node id 5
             2.0,1.0, & ! node id 6
             0.0,2.0, & ! node id 7
             1.0,2.0, & ! node id 8
             2.0,2.0 /) ! node id 9

! Allocate and fill the node owner array.
! Since this Mesh is all on PET 0, it’s just set to all 0.
allocate(nodeOwners(numNodes))
nodeOwners=0 ! everything on PET 0


! Set the number of each type of element, plus the total number.
numQuadElems=3
numTriElems=2
numTotElems=numQuadElems+numTriElems


! Allocate and fill the element id array.
allocate(elemIds(numTotElems))
elemIds=(/1,2,3,4,5/)


! Allocate and fill the element topology type array.
allocate(elemTypes(numTotElems))
elemTypes=(/ESMF_MESHELEMTYPE_QUAD, & ! elem id 1
            ESMF_MESHELEMTYPE_TRI,  & ! elem id 2
            ESMF_MESHELEMTYPE_TRI,  & ! elem id 3
            ESMF_MESHELEMTYPE_QUAD, & ! elem id 4
            ESMF_MESHELEMTYPE_QUAD/)  ! elem id 5


! Allocate and fill the element connection type array.
! Note that entries in this array refer to the
! positions in the nodeIds, etc. arrays and that
! the order and number of entries for each element
! reflects that given in the Mesh options
```

```
! section for the corresponding entry
! in the elemTypes array. The number of
! entries in this elemConn array is the
! number of nodes in a quad. (4) times the
! number of quad. elements plus the number
! of nodes in a triangle (3) times the number
! of triangle elements.
allocate(elemConn(4*numQuadElems+3*numTriElems))
elemConn=(/1,2,5,4, &  ! elem id 1
           2,3,5,   &  ! elem id 2
           3,6,5,   &  ! elem id 3
           4,5,8,7, &  ! elem id 4
           5,6,9,8/)   ! elem id 5


! Create Mesh structure in 1 step
mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=nodeIds, nodeCoords=nodeCoords, &
       nodeOwners=nodeOwners, elementIds=elemIds,&
       elementTypes=elemTypes, elementConn=elemConn, &
       rc=localrc)


! After the creation we are through with the arrays, so they may be
! deallocated.
deallocate(nodeIds)
deallocate(nodeCoords)
deallocate(nodeOwners)
deallocate(elemIds)
deallocate(elemTypes)
deallocate(elemConn)


! Set arrayspec for example field create
! Use a dimension of 1, because Mesh data is linearized
! into a one dimensional array.
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=localrc)

! At this point the mesh is ready to use. For example, as is
! illustrated here, to have a field created on it. Note that
! the Field only contains data for nodes owned by the current PET.
! Please see Section "Create a Field from a Mesh" under Field
! for more information on creating a Field on a Mesh.
field = ESMF_FieldCreate(mesh, arrayspec,  rc=localrc)
```

29.3.3 Create a small single PET Mesh in three steps

This example is intended to illustrate the creation of a small Mesh in three steps on one PET. The Mesh being created is exactly the same one as in the last example (Section 29.3.2), but the three step process allows the creation to occur in a more memory efficient manner.

```
! Create the mesh structure setting the dimensions
mesh = ESMF_MeshCreate(parametricDim=2,spatialDim=2, rc=localrc)
```

```
! Set number of nodes
numNodes=9

! Allocate and fill the node id array.
allocate(nodeIds(numNodes))
nodeIds=(/1,2,3,4,5,6,7,8,9/)

! Allocate and fill node coordinate array.
! Since this is a 2D Mesh the size is 2x the
! number of nodes.
allocate(nodeCoords(2*numNodes))
nodeCoords=(/0.0,0.0, & ! node id 1
             1.0,0.0, & ! node id 2
             2.0,0.0, & ! node id 3
             0.0,1.0, & ! node id 4
             1.0,1.0, & ! node id 5
             2.0,1.0, & ! node id 6
             0.0,2.0, & ! node id 7
             1.0,2.0, & ! node id 8
             2.0,2.0 /) ! node id 9

! Allocate and fill the node owner array.
! Since this Mesh is all on PET 0, it’s just set to all 0.
allocate(nodeOwners(numNodes))
nodeOwners=0 ! everything on PET 0

! Add the nodes to the Mesh
call ESMF_MeshAddNodes(mesh, nodeIds=nodeIds, &
       nodeCoords=nodeCoords, nodeOwners=nodeOwners, rc=localrc)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! HERE IS THE POINT OF THE THREE STEP METHOD
! WE CAN DELETE THESE NODE ARRAYS BEFORE
! ALLOCATING THE ELEMENT ARRAYS, THEREBY
! REDUCING THE AMOUNT OF MEMORY NEEDED
! AT ONE TIME.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
deallocate(nodeIds)
deallocate(nodeCoords)
deallocate(nodeOwners)


! Set the number of each type of element, plus the total number.
numQuadElems=3
numTriElems=2
numTotElems=numQuadElems+numTriElems

! Allocate and fill the element id array.
allocate(elemIds(numTotElems))
elemIds=(/1,2,3,4,5/)

! Allocate and fill the element topology type array.
allocate(elemTypes(numTotElems))
elemTypes=(/ESMF_MESHELEMTYPE_QUAD, & ! elem id 1
```

```
            ESMF_MESHELEMTYPE_TRI,  & ! elem id 2
            ESMF_MESHELEMTYPE_TRI,  & ! elem id 3
            ESMF_MESHELEMTYPE_QUAD, & ! elem id 4
            ESMF_MESHELEMTYPE_QUAD/)  ! elem id 5


! Allocate and fill the element connection type array.
! Note that entries in this array refer to the
! positions in the nodeIds, etc. arrays and that
! the order and number of entries for each element
! reflects that given in the Mesh options
! section for the corresponding entry
! in the elemTypes array. The number of
! entries in this elemConn array is the
! number of nodes in a quad. (4) times the
! number of quad. elements plus the number
! of nodes in a triangle (3) times the number
! of triangle elements.
allocate(elemConn(4*numQuadElems+3*numTriElems))
elemConn=(/1,2,5,4, &  ! elem id 1
           2,3,5,   &  ! elem id 2
           3,6,5,   &  ! elem id 3
           4,5,8,7, &  ! elem id 4
           5,6,9,8/)   ! elem id 5


! Finish the creation of the Mesh by adding the elements
call ESMF_MeshAddElements(mesh, elementIds=elemIds,&
       elementTypes=elemTypes, elementConn=elemConn, &
       rc=localrc)

! After the creation we are through with the arrays, so they may be
! deallocated.
deallocate(elemIds)
deallocate(elemTypes)
deallocate(elemConn)


! Set arrayspec for example field create
! Use a dimension of 1, because Mesh data is linearized
! into a one dimensional array.
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=localrc)

! At this point the mesh is ready to use. For example, as is
! illustrated here, to have a field created on it. Note that
! the Field only contains data for nodes owned by the current PET.
! Please see Section "Create a Field from a Mesh" under Field
! for more information on creating a Field on a Mesh.
field = ESMF_FieldCreate(mesh, arrayspec,  rc=localrc)
```

29.3.4 Create a small Mesh on 4 PETs in one step

```
2.0   7 ------- 8        [8] ------ 9
      |         |         |         |
      |    4    |         |    5    |
      |         |         |         |
1.0  [4] ----- [5]       [5] ----- [6]

     0.0       1.0       1.0       2.0

         PET 2               PET 3


1.0   4 ------- 5        [5] ------ 6
      |         |         |  \   3  |
      |    1    |         |    \    |
      |         |         | 2    \  |
0.0   1 ------- 2        [2] ------ 3

     0.0       1.0       1.0      2.0

         PET 0               PET 1

            Node Id labels at corners
           Element Id labels in centers
```

This example is intended to illustrate the creation of a small Mesh on multiple PETs. This example creates the same small 2D Mesh as the previous two examples (See Section 29.3.2 for a diagram), however, in this case the Mesh is broken up across 4 PETs. The figure above illustrates the distribution of the Mesh across the PETs. As in the previous diagram, the node ids are in the corners of the elements and the element ids are in the centers. In this figure '[' and ']' around a character indicate a node which is owned by another PET. The nodeOwner parameter indicates which PET owns the node. Note that the three step creation illustrated in Section 29.3.3 could also be used in a parallel Mesh creation such as this by simply interleaving the three calls in the appropriate places between the node and element array definitions.

```
! Break up what's being set by PET
if (localPET .eq. 0) then !!! This part only for PET 0
   ! Set number of nodes
    numNodes=4

   ! Allocate and fill the node id array.
   allocate(nodeIds(numNodes))
   nodeIds=(/1,2,4,5/)

   ! Allocate and fill node coordinate array.
   ! Since this is a 2D Mesh the size is 2x the
   ! number of nodes.
   allocate(nodeCoords(2*numNodes))
   nodeCoords=(/0.0,0.0, & ! node id 1
                1.0,0.0, & ! node id 2
                0.0,1.0, & ! node id 4
                1.0,1.0 /) ! node id 5

   ! Allocate and fill the node owner array.
```

```
   allocate(nodeOwners(numNodes))
   nodeOwners=(/0, & ! node id 1
               0, & ! node id 2
               0, & ! node id 4
               0/)  ! node id 5

   ! Set the number of each type of element, plus the total number.
   numQuadElems=1
   numTriElems=0
   numTotElems=numQuadElems+numTriElems

   ! Allocate and fill the element id array.
   allocate(elemIds(numTotElems))
   elemIds=(/1/)

   ! Allocate and fill the element topology type array.
   allocate(elemTypes(numTotElems))
   elemTypes=(/ESMF_MESHELEMTYPE_QUAD/) ! elem id 1

   ! Allocate and fill the element connection type array.
   ! Note that entry are local indices
   allocate(elemConn(4*numQuadElems+3*numTriElems))
   elemConn=(/1,2,4,3/) ! elem id 1

 else if (localPET .eq. 1) then !!! This part only for PET 1
   ! Set number of nodes
    numNodes=4

   ! Allocate and fill the node id array.
   allocate(nodeIds(numNodes))
   nodeIds=(/2,3,5,6/)

   ! Allocate and fill node coordinate array.
   ! Since this is a 2D Mesh the size is 2x the
   ! number of nodes.
   allocate(nodeCoords(2*numNodes))
   nodeCoords=(/1.0,0.0, & ! node id 2
               2.0,0.0, & ! node id 3
               1.0,1.0, & ! node id 5
               2.0,1.0 /) ! node id 6

   ! Allocate and fill the node owner array.
   allocate(nodeOwners(numNodes))
   nodeOwners=(/0, & ! node id 2
               1, & ! node id 3
               0, & ! node id 5
               1/)  ! node id 6

   ! Set the number of each type of element, plus the total number.
   numQuadElems=0
   numTriElems=2
   numTotElems=numQuadElems+numTriElems

   ! Allocate and fill the element id array.
   allocate(elemIds(numTotElems))
```

```
  elemIds=(/2,3/)

  ! Allocate and fill the element topology type array.
  allocate(elemTypes(numTotElems))
  elemTypes=(/ESMF_MESHELEMTYPE_TRI, & ! elem id 2
              ESMF_MESHELEMTYPE_TRI/)  ! elem id 3

  ! Allocate and fill the element connection type array.
  allocate(elemConn(4*numQuadElems+3*numTriElems))
  elemConn=(/1,2,3, & ! elem id 2
             2,4,3/)  ! elem id 3

else if (localPET .eq. 2) then !!! This part only for PET 2
  ! Set number of nodes
   numNodes=4

  ! Allocate and fill the node id array.
  allocate(nodeIds(numNodes))
  nodeIds=(/4,5,7,8/)

  ! Allocate and fill node coordinate array.
  ! Since this is a 2D Mesh the size is 2x the
  ! number of nodes.
  allocate(nodeCoords(2*numNodes))
  nodeCoords=(/0.0,1.0, & ! node id 4
               1.0,1.0, & ! node id 5
               0.0,2.0, & ! node id 7
               1.0,2.0 /) ! node id 8

  ! Allocate and fill the node owner array.
  ! Since this Mesh is all on PET 0, it's just set to all 0.
  allocate(nodeOwners(numNodes))
  nodeOwners=(/0, & ! node id 4
               0, & ! node id 5
               2, & ! node id 7
               2/)  ! node id 8

  ! Set the number of each type of element, plus the total number.
  numQuadElems=1
  numTriElems=0
  numTotElems=numQuadElems+numTriElems

  ! Allocate and fill the element id array.
  allocate(elemIds(numTotElems))
  elemIds=(/4/)

  ! Allocate and fill the element topology type array.
  allocate(elemTypes(numTotElems))
  elemTypes=(/ESMF_MESHELEMTYPE_QUAD/) ! elem id 4

  ! Allocate and fill the element connection type array.
  allocate(elemConn(4*numQuadElems+3*numTriElems))
  elemConn=(/1,2,4,3/) ! elem id 4

else if (localPET .eq. 3) then !!! This part only for PET 3
```

```
  ! Set number of nodes
   numNodes=4

  ! Allocate and fill the node id array.
  allocate(nodeIds(numNodes))
  nodeIds=(/5,6,8,9/)

  ! Allocate and fill node coordinate array.
  ! Since this is a 2D Mesh the size is 2x the
  ! number of nodes.
  allocate(nodeCoords(2*numNodes))
  nodeCoords=(/1.0,1.0, &  ! node id 5
               2.0,1.0, &  ! node id 6
               1.0,2.0, &  ! node id 8
               2.0,2.0 /)  ! node id 9

  ! Allocate and fill the node owner array.
  allocate(nodeOwners(numNodes))
  nodeOwners=(/0, & ! node id 5
               1, & ! node id 6
               2, & ! node id 8
               3/)  ! node id 9

  ! Set the number of each type of element, plus the total number.
  numQuadElems=1
  numTriElems=0
  numTotElems=numQuadElems+numTriElems

  ! Allocate and fill the element id array.
  allocate(elemIds(numTotElems))
  elemIds=(/5/)

  ! Allocate and fill the element topology type array.
  allocate(elemTypes(numTotElems))
  elemTypes=(/ESMF_MESHELEMTYPE_QUAD/) ! elem id 5

  ! Allocate and fill the element connection type array.
  allocate(elemConn(4*numQuadElems+3*numTriElems))
  elemConn=(/1,2,4,3/) ! elem id 5
endif


! Create Mesh structure in 1 step
mesh=ESMF_MeshCreate(parametricDim=2, spatialDim=2, &
       nodeIds=nodeIds, nodeCoords=nodeCoords, &
       nodeOwners=nodeOwners, elementIds=elemIds,&
       elementTypes=elemTypes, elementConn=elemConn, &
       rc=localrc)


! After the creation we are through with the arrays, so they may be
! deallocated.
deallocate(nodeIds)
deallocate(nodeCoords)
deallocate(nodeOwners)
```

```
deallocate(elemIds)
deallocate(elemTypes)
deallocate(elemConn)


! Set arrayspec for example field create
! Use a dimension of 1, because Mesh data is linearized
! into a one dimensional array.
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=localrc)

! At this point the mesh is ready to use. For example, as is
! illustrated here, to have a field created on it. Note that
! the Field only contains data for nodes owned by the current PET.
! Please see Section "Create a Field from a Mesh" under Field
! for more information on creating a Field on a Mesh.
field = ESMF_FieldCreate(mesh, arrayspec,  rc=localrc)
```

29.3.5 Create a Mesh from a SCRIP Grid file or an ESMF unstructured Grid file

ESMF supports the creation of a Mesh from a 2D unstructured grid defined in a SCRIP format grid file [13] or an ESMF format grid file. Both the SCRIP grid file and the ESMF grid file are in NetCDF format. 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:
 double grid_area(grid_size) ;
 grid_area:units = "radians^2" ;
 grid_area:long_name = "area weights" ;
 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. ;
 double grid_imask(grid_size) ;
 grid_imask:_FillValue = -9999. ;
 int grid_dims(grid_rank) ;
}
```

The grid cells are organized as a one dimensional array (`grid_rank = 1`). The cell connection is defined using `grid_corner_lat` and `grid_corner_lon` with the maximum number of corners defined in `grid_corners`. `grid_imask` is not used in the Mesh object in the current implementation. The data is located at the center of the grid cell in a SCRIP grid; whereas the data is located at the corner of a cell in an ESMF Mesh object. Therefore, we create a Mesh object by default by constructing a "dual" mesh using `grid_center_lat` and `grid_center_lon`. If the user wishes to not construct the dual mesh, the optional argument `convertToDual` may be used to control this

behavior. When `comvertToDual` is set to .false. the Mesh constructed from the file will not be the dual. This is necessary when using the Mesh as part of a conservative regridding operation in the `ESMF_FieldRegridStore()` call, so the weights are properly generated for the cell centers in the file.
The following example code depicts how to create a Mesh using a SCRIP file. Note that you have to set the filetypeflag to ESMF_FILEFORMAT_SCRIP. If the optional argument `convert3D` is set to .true., the coordinates will be converted into 3D Cartesian first. If the grid is a global grid and will be used in a regrid operation, this flag should be set to .true.

```
mesh = ESMF_MeshCreate(filename="data/ne4np4-pentagons.nc", &
filetypeflag=ESMF_FILEFORMAT_SCRIP, convert3D=.true., rc=localrc)
```

In addition to the SCRIP format, ESMF also supports a more general unstructured grid file format for describing meshes. In the ESMF file format, the node coordinates are defined in a separate array `nodeCoords` and indices to the `nodeCoords` array are used in the element connectivity array `elementConn`. While in the SCRIP format, the two are combined into `grid_corner_lat` and `grid_corner_lon` arrays. The ESMF file format works better 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 also more general than the SCRIP format because it supports higher dimension coordinates and more general topologies. Currently, ESMF_MeshCreate() does not support conversion to a dual mesh for this format. All regrid methods are supported on Meshes in this format. The following is a sample header of a mesh described in the ESMF format.

```
 netcdf ne4np4-esmf {
 dimensions:
 nodeCount = 866 ;
 elementCount = 936 ;
 maxNodePElement = 4 ;
 coordDim = 2 ;
variables:
 double  nodeCoords(numNode, coordDim);
 nodeCoords:units = "degrees,degrees" ;
 int elementConn(numElement, maxNodePElement) ;
 elementConn:long_name = "Node Indices that define the element connectivity";
 elementConn:_FillValue = -1 ;
 byte numElementConn(numElement) ;
 numElementConn:long_name = "Number of nodes per element" ;
 double centerCoords(numElement, coordDim) ;
 centerCoords:units = "degrees" ;
 double elementArea(numElement) ;
 elementArea:units = "radians^2" ;
 elementArea:long_name = "area weights" ;
 int  elementMask(numElement) ;
 elementMask:_FillValue = -9999. ;
// global attributes:
 :gridType="unstructured";
 :version = "0.9" ;
 :inputFile = "ne4np4-pentagons.nc" ;
 :timeGenerated = "Fri Apr 16 16:05:24 2010" ;
}
```

Here is an example of creating a Mesh from an ESMF unstructured grid file. Note that you have to set the filetypeflag to ESMF_FILEFORMAT_ESMFMESH. As with the previous example, we set `convert3D` to true because this is a global grid.

```
mesh = ESMF_MeshCreate(filename="data/ne4np4-esmf.nc", &
         filetypeflag=ESMF_FILEFORMAT_ESMFMESH, convert3D=.true., rc=localrc)
```

29.3.6 Remove Mesh memory

There are two different levels that the memory in a Mesh can be removed. The first of these is the standard destroy call, `ESMF_MeshDestroy()`. As with other classes, this call removes all memory associated with the object, and afterwards the object can not be used further (i.e. should not be used in any methods). The second, which is unique to Mesh, is the `ESMF_MeshFreeMemory()` call. This call removes the connection and coordinate information associated with the Mesh, but leaves the distgrid information. The coordinate and connection information held in the Mesh can consume a large amount of memory for a big Mesh, so using this call can very significantly reduce the amount of memory used. However, once this method has been used on a Mesh there are some restriction on what may be done with it. Once a Mesh has had its memory freed using this method, any Field built on the Mesh can no longer be used as part of an `ESMF_FieldRegridStore()` call. However, because the distgrid information is still part of the Mesh, Fields built on such a Mesh can still be part of an `ESMF_FieldRegrid()` call (where the routehandle was generated previous to the `ESMF_MeshFreeMemory()` operation). Fields may also still be created on these Meshes. The following short piece of code illustrates the use of this call.

```
! Here a Field built on a mesh may be used
! as part of a ESMF_FieldRegridStore() call

! This call removes connection and coordinate
! information, significantly reducing the memory used by
! mesh, but limiting what can be done with it.
call ESMF_MeshFreeMemory(mesh, rc=localrc)

! Here a new Field may be built on mesh, or
! a field built on a mesh may be used as part
! of an ESMF_FieldRegrid() call

! Destroy the mesh
call ESMF_MeshDestroy(mesh, rc=localrc)

! Here mesh can't be used for anything
```

29.4 Class API

29.4.1 ESMF_MeshAssignment(=) - Mesh assignment

INTERFACE:

```
interface assignment(=)
mesh1 = mesh2
```

ARGUMENTS:

```
type(ESMF_Mesh) :: mesh1
type(ESMF_Mesh) :: mesh2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign mesh1 as an alias to the same ESMF Mesh object in memory as mesh2. If mesh2 is invalid, then mesh1 will be equally invalid after the assignment.
The arguments are:

mesh1 The ESMF_Mesh object on the left hand side of the assignment.

mesh2 The ESMF_Mesh object on the right hand side of the assignment.

29.4.2 ESMF_MeshOperator(==) - Mesh equality operator

INTERFACE:

```
   interface operator(==)
     if (mesh1 == mesh2) then ... endif
               OR
     result = (mesh1 == mesh2)
```

RETURN VALUE:

```
     logical :: result
```

ARGUMENTS:

```
     type(ESMF_Mesh), intent(in) :: mesh1
     type(ESMF_Mesh), intent(in) :: mesh2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether mesh1 and mesh2 are valid aliases to the same ESMF Mesh object in memory. For a more general comparison of two ESMF Meshes, going beyond the simple alias test, the ESMF_MeshMatch() function (not yet implemented) must be used.
The arguments are:

mesh1 The ESMF_Mesh object on the left hand side of the equality operation.

mesh2 The ESMF_Mesh object on the right hand side of the equality operation.

29.4.3 ESMF_MeshOperator(/=) - Mesh not equal operator

INTERFACE:

```
   interface operator(/=)
     if (mesh1 /= mesh2) then ... endif
               OR
     result = (mesh1 /= mesh2)
```

RETURN VALUE:

```
     logical :: result
```

ARGUMENTS:

```
     type(ESMF_Mesh), intent(in) :: mesh1
     type(ESMF_Mesh), intent(in) :: mesh2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether mesh1 and mesh2 are *not* valid aliases to the same ESMF Mesh object in memory. For a more general comparison of two ESMF Meshes, going beyond the simple alias test, the ESMF_MeshMatch() function (not yet implemented) must be used.
The arguments are:

mesh1 The `ESMF_Mesh` object on the left hand side of the non-equality operation.

mesh2 The `ESMF_Mesh` object on the right hand side of the non-equality operation.

29.4.4 ESMF_MeshAddElements - Add elements to a Mesh

INTERFACE:

```
subroutine ESMF_MeshAddElements(mesh, elementIds, elementTypes, &
             elementConn, rc)
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(inout)         :: mesh
integer,         intent(in)     :: elementIds(:)
integer,         intent(in)     :: elementTypes(:)
integer,         intent(in)     :: elementConn(:)
integer,         intent(out), optional :: rc
```

DESCRIPTION:

This call is the third and last part of the three part mesh create sequence and should be called after the mesh is created with `ESMF_MeshCreate()` (29.4.6) and after the nodes are added with `ESMF_MeshAddNodes()` (29.4.5). This call adds the elements to the mesh and finalizes the create. After this call the Mesh is usable, for example a Field may be built on the created Mesh object and this Field may be used in a `ESMF_FieldRegridStore()` call.
The parameters to this call `elementIds`, `elementTypes`, and `elementConn` describe the elements to be created. The description for a particular element lies at the same index location in `elementIds` and `elementTypes`. Each entry in `elementConn` consists of the list of nodes used to create that element, so the connections for element e in the `elementIds` array will start at $number\_of\_nodes\_in\_element(1) + number\_of\_nodes\_in\_element(2) + \cdots + number\_of\_nodes\_in\_element(e-1) + 1$ in `elementConn`.
This call is *collective* across the current VM.

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

elementTypes An array containing the types of the elements to be created on this PET. The types used must be appropriate for the parametric dimension of the Mesh. Please see Section 29.2.1 for the list of options. This input consists of a 1D array the size of the number of elements on this PET.

elementConn An array containing the indexes of the sets of nodes to be connected together to form the elements to be created on this PET. The entries in this list are NOT node global ids, but rather each entry is a local index (1 based) into the list of nodes which were created on this PET by the previous `ESMF_MeshAddNodes()` call. In other words, an entry of 1 indicates that this element contains the node described by `nodeIds(1)`, `nodeCoords(1)`, etc. passed into the `ESMF_MeshAddNodes()` call on this PET. It is also important to note that the order of the nodes in an element connectivity list matters. Please see Section 29.2.1 for diagrams illustrating the correct order of nodes in a element. This input consists of a 1D array with a total size equal to the sum of the number of nodes in each element on this PET. The number of nodes in each element is implied by its element type in `elementTypes`. The nodes for each element are in sequence in this array (e.g. the nodes for element 1 are elementConn(1), elementConn(2), etc.).

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

29.4.5 ESMF_MeshAddNodes - Add nodes to a Mesh

INTERFACE:

```
subroutine ESMF_MeshAddNodes(mesh, nodeIds, nodeCoords, nodeOwners, rc)
```

ARGUMENTS:

```
type(ESMF_Mesh),    intent(inout)          :: mesh
integer,            intent(in)             :: nodeIds(:)
real(ESMF_KIND_R8), intent(in)             :: nodeCoords(:)
integer,            intent(in)             :: nodeOwners(:)
integer,            intent(out), optional  :: rc
```

DESCRIPTION:

This call is the second part of the three part mesh create sequence and should be called after the mesh's dimensions are set using `ESMF_MeshCreate()` (29.4.6). This call adds the nodes to the mesh. The next step is to call `ESMF_MeshAddElements()` (29.4.4).
The parameters to this call `nodeIds`, `nodeCoords`, and `nodeOwners` describe the nodes to be created on this PET. The description for a particular node lies at the same index location in `nodeIds` and `nodeOwners`. Each entry in `nodeCoords` consists of spatial dimension coordinates, so the coordinates for node n in the `nodeIds` array will start at $(n-1) * spatialDim + 1$.

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

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

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

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

29.4.6 ESMF_MeshCreate - Create a Mesh as a 3 step process

INTERFACE:

```
   ! Private name; call using ESMF_MeshCreate()
     function ESMF_MeshCreate3Part(parametricDim, spatialDim, rc)
```

RETURN VALUE:

```
     type(ESMF_Mesh)         :: ESMF_MeshCreate3Part
```

ARGUMENTS:

```
     integer,                intent(in)            :: parametricDim
     integer,                intent(in)            :: spatialDim
     integer,                intent(out), optional :: rc
```

DESCRIPTION:

This call is the first part of the three part mesh create sequence. This call sets the dimension of the elements in the mesh (`parametricDim`) and the number of coordinate dimensions in the mesh (`spatialDim`). The next step is to call `ESMF_MeshAddNodes()` (29.4.5) to add the nodes and then `ESMF_MeshAddElements()` (29.4.4) to add the elements and finalize the mesh.
This call is *collective* across the current VM.

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

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

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

29.4.7 ESMF_MeshCreate - Create a Mesh all at once

INTERFACE:

```
   ! Private name; call using ESMF_MeshCreate()
     function ESMF_MeshCreate1Part(parametricDim, spatialDim, &
                          nodeIds, nodeCoords, nodeOwners, &
                          elementIds, elementTypes, elementConn, &
                          rc)
```

RETURN VALUE:

```
     type(ESMF_Mesh)                          :: ESMF_MeshCreate1Part
```

ARGUMENTS:

```
     integer,            intent(in)            :: parametricDim
     integer,            intent(in)            :: spatialDim
     integer,            intent(in)            :: nodeIds(:)
     real(ESMF_KIND_R8), intent(in)            :: nodeCoords(:)
     integer,            intent(in)        :: nodeOwners(:)
     integer,            intent(in)        :: elementIds(:)
     integer,            intent(in)        :: elementTypes(:)
     integer,            intent(in)        :: elementConn(:)
     integer,            intent(out), optional :: rc
```

DESCRIPTION:

Create a Mesh object in one step. After this call the Mesh is usable, for example, a Field may be built on the created Mesh object and this Field may be used in a `ESMF_FieldRegridStore()` call.
This call sets the dimension of the elements in the mesh (`parametricDim`) and the number of coordinate dimensions in the mesh (`spatialDim`). It then creates the nodes, and then creates the elements by connecting together the nodes.
The parameters to this call `nodeIds`, `nodeCoords`, and `nodeOwners` describe the nodes to be created on this PET. The description for a particular node lies at the same index location in `nodeIds` and `nodeOwners`. Each entry in `nodeCoords` consists of spatial dimension coordinates, so the coordinates for node n in the `nodeIds` array will start at $(n-1)*spatialDim+1$.
The parameters to this call `elementIds`, `elementTypes`, and `elementConn` describe the elements to be created. The description for a particular element lies at the same index location in `elementIds` and `elementTypes`. Each entry in `elementConn` consists of the list of nodes used to create that element, so the connections for element e in the `elementIds` array will start at $number\_of\_nodes\_in\_element(1)+number\_of\_nodes\_in\_element(2)+\cdots+number\_of\_nodes\_in\_element(e-1)+1$ in `elementConn`.
This call is *collective* across the current VM.

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

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

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

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

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

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

elementTypes An array containing the types of the elements to be created on this PET. The types used must be appropriate for the parametric dimension of the Mesh. Please see Section 29.2.1 for the list of options. This input consists of a 1D array the size of the number of elements on this PET.

elementConn An array containing the indexes of the sets of nodes to be connected together to form the elements to be created on this PET. The entries in this list are NOT node global ids, but rather each entry is a local index (1 based) into the list of nodes to be created on this PET by this call. In other words, an entry of 1 indicates that this element contains the node described by `nodeIds(1)`, `nodeCoords(1)`, etc. on this PET. It is also important to note that the order of the nodes in an element connectivity list matters. Please see Section 29.2.1 for diagrams illustrating the correct order of nodes in a element. This input consists of a 1D array with a total size equal to the sum of the number of nodes contained in each element on this PET. The number of nodes in each element is implied by its element type in `elementTypes`. The nodes for each element are in sequence in this array (e.g. the nodes for element 1 are elementConn(1), elementConn(2), etc.).

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

29.4.8 ESMF_MeshCreate - Create a Mesh from a file

INTERFACE:

```
! Private name; call using ESMF_MeshCreate()
  function ESMF_MeshCreateFromFile(filename, filetypeflag, convert3D, &
              convertToDual, rc)
```

RETURN VALUE:

```
type(ESMF_Mesh)         :: ESMF_MeshCreateFromFile
```

ARGUMENTS:

```
character(len=*),            intent(in)            :: filename
type(ESMF_FileFormat_Flag), intent(in)            :: filetypeflag
logical,                     intent(in),  optional :: convert3D
logical,                     intent(in),  optional :: convertToDual
integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Create a Mesh from a file. Provides options to convert to 3D and in the case of SCRIP format files, allows the dual of the mesh to be created.
This call is *collective* across the current VM.

filename The name of the grid file

filetypeflag The file type of the grid file to be read, please see Section 29.2.2 for a list of valid options.

convert3D if TRUE, the node coordinates will be converted into 3D Cartisian, which is required for a global grid

convertToDual if TRUE, the mesh will be converted to its dual. If not specified, defaults to true. Converting to dual is not supported with file type `ESMF_FILEFORMAT_ESMFMESH`, so when using that file type this parameter has no effect.

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

29.4.9 ESMF_MeshDestroy - Release resources associated with a Mesh

INTERFACE:

```
subroutine ESMF_MeshDestroy(mesh, keywordenforcer, rc)
```

RETURN VALUE:

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

mesh Mesh object to be destroyed.

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

29.4.10 ESMF_MeshFreeMemory - Remove a Mesh and its memory

INTERFACE:

```
      subroutine ESMF_MeshFreeMemory(mesh, rc)
```

RETURN VALUE:

ARGUMENTS:

```
    type(ESMF_Mesh), intent(inout)        :: mesh
    integer,        intent(out), optional :: rc
```

DESCRIPTION:

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

mesh Mesh object whose memory is to be freed.

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

29.4.11 ESMF_MeshGet - Get information from a Mesh

INTERFACE:

```
      subroutine ESMF_MeshGet(mesh, parametricDim, spatialDim, &
                   nodalDistgrid, elementDistgrid, &
                   numOwnedNodes, ownedNodeCoords, &
                   numOwnedElements, isMemFreed, rc)
```

RETURN VALUE:

ARGUMENTS:

```
type(ESMF_Mesh),     intent(inout)        :: mesh
integer,             intent(out), optional :: parametricDim
integer,             intent(out), optional :: spatialDim
type(ESMF_DistGrid), intent(out), optional :: nodalDistgrid
type(ESMF_DistGrid), intent(out), optional :: elementDistgrid
integer,             intent(out), optional :: numOwnedNodes
real(ESMF_KIND_R8),  intent(out), optional :: ownedNodeCoords(:)
integer,             intent(out), optional :: numOwnedElements
logical,             intent(out), optional :: isMemFreed
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Get various information from a mesh.
The arguments are:

mesh Mesh object to retrieve information from.

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

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

[nodalDistgrid] A 1D arbitrary distgrid describing the distribution of the nodes across the PETs. Note that on each PET the distgrid will only contain entries for nodes owned by that PET. This is the DistGrid that would be used to construct the Array in a Field that is constructed on `mesh`.

[elementDistgrid] A 1D arbitrary distgrid describing the distribution of elements across the PETs. Note that on each PET the distgrid will only contain entries for elements owned by that PET.

[numOwnedNodes] The number of local nodes which are owned by this PET. This is the number of PET local entries in the nodalDistgrid.

[ownedNodeCoords] The coordinates for the local nodes. These coordinates will be in the proper order to correspond with the nodes in the `nodalDistgrid` returned by this call, and hence with a Field built on `mesh`. The size of the input array should be the spatial dim of `mesh` times `numOwnedNodes`.

[numOwnedElements] The number of local elements which are owned by this PET. Note that every element is owned by the PET it resides on, so unlike for nodes, `numOwnedElements` is identical to the number of elements on the PET. It is also the number of PET local entries in the elementDistgrid.

[isMemFreed] Indicates if the coordinate and connection memory been freed from `mesh`. If so, it can no longer be used as part of an `ESMF_FieldRegridStore()` call.

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

30 XGrid Class

30.1 Description

An exchange grid represents the 2D boundary layer usually between the atmosphere on one side and ocean and land on the other in an Earth system model. There are dynamical and thermodynamical processes on either side of the boundary layer and on the boundary layer itself. The boundary layer exchanges fluxes from either side and adjusts boundary conditions for the model components involved. For climate modeling, it is critical that the fluxes transferred by the boundary layer are conservative.
The exchange grid is implemented as a collection of the intersected cells between atmosphere and ocean/land[22]. These cells can have irregular shapes and can be broken down into triangles facilitating a finite element approach. In practice, there is a threshold of minimum cell area below which intersections are discarded.

30.2 Constants

30.2.1 ESMF_XGRIDSIDE

DESCRIPTION:
Specify which side of the `ESMF_XGrid` the current operation is taking place.
The type of this flag is:
`type(ESMF_XGridSide_Flag)`
The valid values are:

ESMF_XGRIDSIDE_A A side of the eXchange Grid, corresponding to the A side of the Grids used to create an XGrid.

ESMF_XGRIDSIDE_B B side of the eXchange Grid, corresponding to the B side of the Grids used to create an XGrid.

ESMF_XGRIDSIDE_BALANCED The internally generated balanced side of the eXchange Grid in the middle.

30.3 Use and Examples

30.3.1 Create an XGrid from user input data then use it for regridding

XGrid can be created from user input data, such as Grids on either side, area and centroid information of XGrid cells, sparse matrix matmul information such as factorList and factorIndexList. The functionalities provided by the XGrid object is constrained by the user supplied input during its creation time.
In this example, we will set up a simple XGrid from overlapping Grids on either side of the XGrid. Then we perform a flux exchange from one side to the other side of the XGrid. The Grids are laid out in the following figure:

We start by creating the Grids on both sides and associate coordinates with the Grids. For details of Grid creation and coordinate use, please refer to Grid class documentation.

```
    sideA(1) = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/2,2/), &
        coordDep1=(/1/), &
        coordDep2=(/2/), &
        name='source Grid 1 on side A', rc=localrc)


    sideA(2) = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/2,1/), &
        coordDep1=(/1/), &
        coordDep2=(/2/), &
        name='source Grid 2 on side A', rc=localrc)


    do i = 1, 2
        call ESMF_GridAddCoord(sideA(i), staggerloc=ESMF_STAGGERLOC_CENTER, &
            rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo
```

Coordinate for the Grids on sideA, refer to the Grid layout diagram for the interpretation of the coordinate values:

```
    ! SideA first grid
    centroidA1X=(/0.5, 1.5/)
    centroidA1Y=(/0.5, 1.5/)
    call ESMF_GridGetCoord(sideA(1), localDE=0, &
        staggerLoc=ESMF_STAGGERLOC_CENTER, coordDim=1, &
```

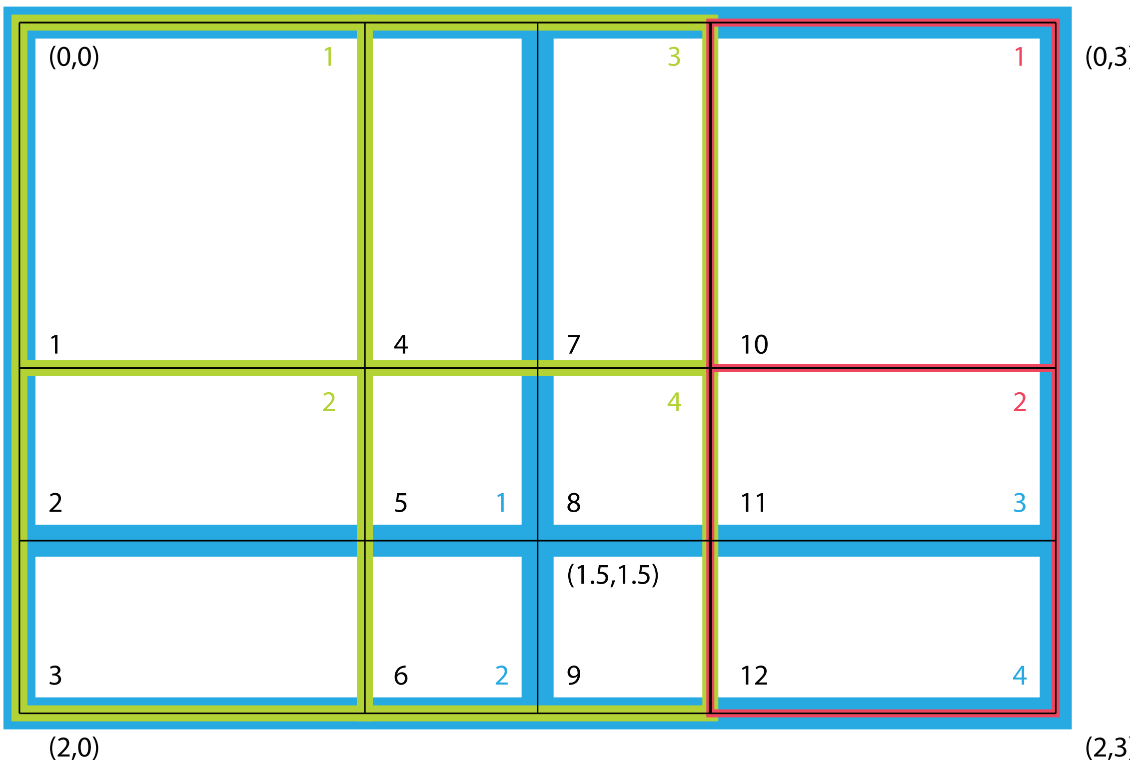


Figure 20: Grid layout for simple XGrid creation example. Overlapping of 3 Grids (Green 2x2, Red 2x1, Blue 2x2). Green and red Grids on side A, blue Grid on side B, black indicates the resulting XGrid. Color coded sequence indices are shown. Physical coordinates are the tuples in parenthese, e.g. at the four corners of rectangular computational domain.

```
        farrayPtr=coordX, rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    coordX = centroidA1X
    call ESMF_GridGetCoord(sideA(1), localDE=0, &
        staggerLoc=ESMF_STAGGERLOC_CENTER, coordDim=2, &
        farrayPtr=coordY, rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
         endflag=ESMF_END_ABORT)
    coordY = centroidA1Y

    ! SideA second grid
    centroidA2X=(/0.5, 1.5/)
    centroidA2Y=(/2.5/)
    call ESMF_GridGetCoord(sideA(2), localDE=0, &
        staggerLoc=ESMF_STAGGERLOC_CENTER, coordDim=1, &
        farrayPtr=coordX, rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
        endflag=ESMF_END_ABORT)
    coordX = centroidA2X
    call ESMF_GridGetCoord(sideA(2), localDE=0, &
        staggerLoc=ESMF_STAGGERLOC_CENTER, coordDim=2, &
        farrayPtr=coordY, rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    coordY = centroidA2Y
```

Create the destination grid on side B, only one Grid exists on side B. Also associate coordinate with the Grid:

```
    sideB(1) = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/2,2/), &
        coordDep1=(/1/), coordDep2=(/2/), &
        name='destination Grid on side B', rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    do i = 1, 1
        call ESMF_GridAddCoord(sideB(i), staggerloc=ESMF_STAGGERLOC_CENTER, &
            rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo

    ! SideB grid
    centroidBX=(/0.75, 1.75/)
    centroidBY=(/0.75, 2.25/)
    call ESMF_GridGetCoord(sideB(1), localDE=0, &
        staggerLoc=ESMF_STAGGERLOC_CENTER, coordDim=1, farrayPtr=coordX, &
                rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    coordX = centroidBX
    call ESMF_GridGetCoord(sideB(1), localDE=0, &
        staggerLoc=ESMF_STAGGERLOC_CENTER, coordDim=2, farrayPtr=coordY, &
                rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
```

```
coordY = centroidBY
```

Set up the mapping indices and weights from A side to the XGrid. For details of sequence indices, factorIndexList, and factorList, please see section 24.2.17 in the reference manual. Please refer to the figure above for interpretation of the sequence indices used here.
In order to compute the destination flux on sideB through the XGrid as an mediator, we need to set up the factorList (weights) and factorIndexList (indices) for sparse matrix matmul in this formulation: dst_flux = W'*W*src_flux, where W' is the weight matrix from the XGrid to destination; and W is the weight matrix from source to the XGrid. The weight matrix is generated using destination area weighted algorithm. Please refer to figure 20 for details.

```
! Set up mapping from A1 -> X
sparseMatA2X(1)%factorIndexList(1,1)=1    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,2)=2    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,3)=2    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,4)=3    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,5)=4    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,6)=4    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,7)=3    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,8)=4    ! src seq index (green)
sparseMatA2X(1)%factorIndexList(1,9)=4    ! src seq index (green)

sparseMatA2X(1)%factorIndexList(2,1)=1    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,2)=2    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,3)=3    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,4)=4    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,5)=5    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,6)=6    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,7)=7    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,8)=8    ! dst seq index (black)
sparseMatA2X(1)%factorIndexList(2,9)=9    ! dst seq index (black)

! Set up mapping from A2 -> X
sparseMatA2X(2)%factorIndexList(1,1)=1    ! src seq index (red)
sparseMatA2X(2)%factorIndexList(1,2)=2    ! src seq index (red)
sparseMatA2X(2)%factorIndexList(1,3)=2    ! src seq index (red)

sparseMatA2X(2)%factorIndexList(2,1)=10   ! dst seq index (black)
sparseMatA2X(2)%factorIndexList(2,2)=11   ! dst seq index (black)
sparseMatA2X(2)%factorIndexList(2,3)=12   ! dst seq index (black)
```

Set up the mapping weights from side A to the XGrid:

```
! Note that the weights are dest area weighted, they are ratio
! of areas with destination area as the denominator.
! Set up mapping weights from A1 -> X
sparseMatA2X(1)%factorList(:)=1.

! Set up mapping weights from A2 -> X
sparseMatA2X(2)%factorList(:)=1.
```

Set up the mapping indices and weights from the XGrid to B side:

```
! Set up mapping from X -> B
sparseMatX2B(1)%factorIndexList(1,1)=1    ! src seq index (black)
```

```
sparseMatX2B(1)%factorIndexList(1,2)=2    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,3)=3    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,4)=4    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,5)=5    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,6)=6    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,7)=7    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,8)=8    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,9)=9    ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,10)=10  ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,11)=11  ! src seq index (black)
sparseMatX2B(1)%factorIndexList(1,12)=12  ! src seq index (black)

sparseMatX2B(1)%factorIndexList(2,1)=1    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,2)=1    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,3)=2    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,4)=1    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,5)=1    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,6)=2    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,7)=3    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,8)=3    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,9)=4    ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,10)=3   ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,11)=3   ! dst seq index (blue)
sparseMatX2B(1)%factorIndexList(2,12)=4   ! dst seq index (blue)

! Set up mapping weights from X -> B
sparseMatX2B(1)%factorList(1)=4./9.
sparseMatX2B(1)%factorList(2)=2./9.
sparseMatX2B(1)%factorList(3)=2./3.
sparseMatX2B(1)%factorList(4)=2./9.
sparseMatX2B(1)%factorList(5)=1./9.
sparseMatX2B(1)%factorList(6)=1./3.
sparseMatX2B(1)%factorList(7)=2./9.
sparseMatX2B(1)%factorList(8)=1./9.
sparseMatX2B(1)%factorList(9)=1./3.
sparseMatX2B(1)%factorList(10)=4./9.
sparseMatX2B(1)%factorList(11)=2./9.
sparseMatX2B(1)%factorList(12)=2./3.
```

Optionally the area can be setup to compute surface area weighted flux integrals:

```
! Set up destination areas to adjust weighted flux
xgrid_area(1) = 1.
xgrid_area(2) = 0.5
xgrid_area(3) = 0.5
xgrid_area(4) = 0.5
xgrid_area(5) = 0.25
xgrid_area(6) = 0.25
xgrid_area(7) = 0.5
xgrid_area(8) = 0.25
xgrid_area(9) = 0.25
xgrid_area(10) = 1.
xgrid_area(11) = 0.5
xgrid_area(12) = 0.5
```

Create an XGrid based on the user supplied regridding parameters:

```
xgrid = ESMF_XGridCreate(sideA, sideB, offline=.true., area=xgrid_area, &
    centroid=centroid, sparseMatA2X=sparseMatA2X, &
    sparseMatX2B=sparseMatX2B, rc=localrc)
```

Create an `ESMF_Field` on the XGrid:

```
field = ESMF_FieldCreate(xgrid, typekind=ESMF_TYPEKIND_R8, &
            rc=localrc)
```

Query the Field for its Fortran data pointer and its exclusive bounds:

```
call ESMF_FieldGet(field, farrayPtr=xfarrayPtr, &
    exclusiveLBound=xlb, exclusiveUBound=xub, rc=localrc)
```

Setup and initialize src and dst Fields on side A and side B Grids, source Fields have different source flux:

```
    do i = 1, 2
        srcField(i) = ESMF_FieldCreate(sideA(i), &
                typekind=ESMF_TYPEKIND_R8, rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        call ESMF_FieldGet(srcField(i), farrayPtr=farrayPtr, rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        farrayPtr = i
    enddo
    do i = 1, 1
        dstField(i) = ESMF_FieldCreate(sideB(i), &
                typekind=ESMF_TYPEKIND_R8, rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        call ESMF_FieldGet(dstField(i), farrayPtr=farrayPtr, rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        farrayPtr = 0.0
    enddo
```

The current implementation requires that Grids used to generate the XGrid must not match, i.e. they are different either topologically or geometrically or both. In this example, the first source Grid is topologically identical to the destination Grid but their geometric coordinates are different. This requirement will be relaxed in a future release. First we compute the regrid routehandles, these routehandles can be used repeatedly afterwards. Then we initialize the values in the Fields. Finally we execute the Regrid.

```
    ! Compute regrid routehandles. The routehandles can be used
    ! repeatedly afterwards.
    ! From A -> X
    do i = 1, 2
        call ESMF_FieldRegridStore(xgrid, srcField(i), field, &
routehandle=rh_src2xgrid(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
```

```
    enddo
    ! from X -> B
    do i = 1, 1
        call ESMF_FieldRegridStore(xgrid, field, dstField(i), &
routehandle=rh_xgrid2dst(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo

    ! Initialize values in the source Fields on side A
    do i = 1, 2
        call ESMF_FieldGet(srcField(i), farrayPtr=farrayPtr, rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        farrayPtr = i
    enddo
    ! Initialize values in the destination Field on XGrid
    xfarrayPtr = 0.0
    ! Initialize values in the destination Field on Side B
    do i = 1, 1
        call ESMF_FieldGet(dstField(i), farrayPtr=farrayPtr, rc=localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        farrayPtr = 0.0
    enddo
```

First we regrid from the Fields on side A to the Field on the XGrid:

```
    ! Execute regrid from A -> X
    do i = 1, 2
        call ESMF_FieldRegrid(srcField(i), field, &
            routehandle=rh_src2xgrid(i), &
            zeroregion=ESMF_REGION_SELECT, rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo
```

Next we regrid from the Field on XGrid to the destination Field on side B:

```
    ! Execute the regrid store
    do i = 1, 1
        call ESMF_FieldRegrid(field, dstField(i), &
            routehandle=rh_xgrid2dst(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo
```

In the above example, we first set up all the required paramters to create an XGrid from user supplied input. Then we create Fields on the XGrid and the Grids on either side. Finally we use the `ESMF_FieldRegrid()` interface to perform a flux exchange from the source side to the destination side.

30.3.2 Query the XGrid for its internal information

One can query the XGrid for its internal information:

```
    call ESMF_XGridGet(xgrid, &
        ngridA=ngridA, &    ! number of Grids on side A
        ngridB=ngridB, &    ! number of Grids on side B
        sideA=l_sideA, &    ! list of Grids on side A
        sideB=l_sideB, &    ! list of Grids on side B
        area=l_area, &      ! list of area of XGrid
        centroid=l_centroid, &  ! list of centroid of XGrid
        distgridA=l_sideAdg, &  ! list of Distgrids on side A
        distgridM = distgrid, & ! balanced distgrid
        sparseMatA2X=l_sparseMatA2X, & !sparse matrix matmul parameters A to X
        sparseMatX2B=l_sparseMatX2B, & !sparse matrix matmul parameters X to B
        rc=localrc)


    call ESMF_XGridGet(xgrid, localDe=0, &
        elementCount=eleCount, &    ! elementCount on the localDE
        exclusiveCount=ec, &        ! exclusive count
        exclusiveLBound=elb, &      ! exclusive lower bound
        exclusiveUBound=eub, &      ! exclusive upper bound
        rc=localrc)


    call ESMF_XGridGet(xgrid, &
        xgridSide=ESMF_XGRIDSIDE_A, & ! side of the XGrid to query
        gridIndex=1, &              ! index of the distgrid
        distgrid=distgrid, &        ! the distgrid returned
        rc=localrc)
```

30.3.3 Destroying the XGrid and other resources

Clean up the resources by destroy the XGrid and other objects:

```
    ! After the regridding is successful.
    ! Clean up all the allocated resources:
    call ESMF_FieldDestroy(field, rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)

    call ESMF_XGridDestroy(xgrid, rc=localrc)
    if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)

    do i = 1, 2
        call ESMF_FieldDestroy(srcField(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
        call ESMF_GridDestroy(sideA(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo

    do i = 1, 1
        call ESMF_FieldDestroy(dstField(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
```

```
        call ESMF_GridDestroy(sideB(i), rc = localrc)
        if(localrc /= ESMF_SUCCESS) call ESMF_Finalize(rc=localrc, &
endflag=ESMF_END_ABORT)
    enddo

    deallocate(sparseMatA2X(1)%factorIndexList, sparseMatA2X(1)%factorList)
    deallocate(sparseMatA2X(2)%factorIndexList, sparseMatA2X(2)%factorList)
    deallocate(sparseMatX2B(1)%factorIndexList, sparseMatX2B(1)%factorList)
```

30.4 Restrictions and Future Work

30.4.1 Restrictions and Future Work

1. **CAUTION:** The XGrid class and its APIs are only tested in a uni-processor setup, however in principle it should also work multi-processor. This limitation will be removed in a future release.
2. More convenient `ESMF_XGridCreate()` API will be provided in the future that will not require a user to supply the interpolation matrix.

30.5 Design and Implementation Notes

1. The XGrid class is implemented in Fortran, and as such is defined inside the framework by a XGrid derived type and a set of subprograms (functions and subroutines) which operate on that derived type. The XGrid class contains information needed to create Grid, Field, and Sparse Matrix MatMul.
2. XGrids follow the framework-wide convention of the *unison* creation and operation rule: All PETs which are part of the currently executing VM must create the same XGrids at the same point in their execution. Since an early user request was that global object creation not impose the overhead of a barrier or synchronization point, XGrid creation does no inter-PET communication. For this to work, each PET must query the total number of PETs in this VM, and which local PET number it is. It can then compute which DE(s) are part of the local decomposition, and any global information can be computed in unison by all PETs independently of the others. In this way the overhead of communication is avoided, at the cost of more difficulty in diagnosing program bugs which result from not all PETs executing the same create calls.

30.6 Class API

30.6.1 ESMF_XGridAssignment(=) - XGrid assignment

INTERFACE:

```
interface assignment(=)
xgrid1 = xgrid2
```

ARGUMENTS:

```
type(ESMF_XGrid) :: xgrid1
type(ESMF_XGrid) :: xgrid2
```

DESCRIPTION:

Assign xgrid1 as an alias to the same ESMF XGrid object in memory as xgrid2. If xgrid2 is invalid, then xgrid1 will be equally invalid after the assignment.
The arguments are:

xgrid1 The `ESMF_XGrid` object on the left hand side of the assignment.

xgrid2 The `ESMF_XGrid` object on the right hand side of the assignment.

30.6.2 ESMF_XGridOperator(==) - XGrid equality operator

INTERFACE:

```
interface operator(==)
if (xgrid1 == xgrid2) then ... endif
          OR
result = (xgrid1 == xgrid2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid1
type(ESMF_XGrid), intent(in) :: xgrid2
```

DESCRIPTION:

Test whether xgrid1 and xgrid2 are valid aliases to the same ESMF XGrid object in memory. For a more general comparison of two ESMF XGrids, going beyond the simple alias test, the ESMF_XGridMatch() function (not yet implemented) must be used.
The arguments are:

xgrid1 The `ESMF_XGrid` object on the left hand side of the equality operation.

xgrid2 The `ESMF_XGrid` object on the right hand side of the equality operation.

30.6.3 ESMF_XGridOperator(/=) - XGrid not equal operator

INTERFACE:

```
interface operator(/=)
if (xgrid1 /= xgrid2) then ... endif
          OR
result = (xgrid1 /= xgrid2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid1
type(ESMF_XGrid), intent(in) :: xgrid2
```

DESCRIPTION:

Test whether xgrid1 and xgrid2 are *not* valid aliases to the same ESMF XGrid object in memory. For a more general comparison of two ESMF XGrids, going beyond the simple alias test, the ESMF_XGridMatch() function (not yet implemented) must be used.
The arguments are:

xgrid1 The `ESMF_XGrid` object on the left hand side of the non-equality operation.

xgrid2 The `ESMF_XGrid` object on the right hand side of the non-equality operation.

30.6.4 ESMF_XGridCreate - Create an XGrid from user input

INTERFACE:

```
function ESMF_XGridCreate(sideA, sideB, &
&
sideAToXGridScheme, sideBToXGridScheme, &
sideAPriority, sideBPriority, &
sideAMaskValues, sideBMaskValues, &
storeOverlay, &
offline, &
sparseMatA2X, sparseMatX2A, sparseMatB2X, sparseMatX2B, &
area, centroid, &
name, rc)
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in)                  :: sideA(:), sideB(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional                :: sideAToXGridScheme
integer, intent(in), optional                :: sideBToXGridScheme
integer, intent(in), optional                :: sideAPriority(:)
integer, intent(in), optional                :: sideBPriority(:)
integer(ESMF_KIND_I4), intent(in), optional :: sideAMaskValues(:)
integer(ESMF_KIND_I4), intent(in), optional :: sideBMaskValues(:)
logical, intent(in), optional                :: storeOverlay
logical, intent(in), optional                :: offline
type(ESMF_XGridSpec), intent(in), optional  :: sparseMatA2X(:)
type(ESMF_XGridSpec), intent(in), optional  :: sparseMatX2A(:)
type(ESMF_XGridSpec), intent(in), optional  :: sparseMatB2X(:)
type(ESMF_XGridSpec), intent(in), optional  :: sparseMatX2B(:)
real(ESMF_KIND_R8), intent(in), optional    :: area(:)
real(ESMF_KIND_R8), intent(in), optional    :: centroid(:,:)
character(len=*), intent(in), optional      :: name
integer, intent(out), optional               :: rc
```

RETURN VALUE:

```
type(ESMF_XGrid)             :: ESMF_XGridCreate
```

DESCRIPTION:

Create an XGrid from user input
The arguments are:

sideA 2D Grids on side A

sideB 2D Grids on side B

[sideAToXGridScheme] Specify the geometry and unit of metric of the Grids on A side.

[sideBToXGridScheme] Specify the geometry and unit of metric of the Grids on B side.

[sideAPriority] Priority array of Grids on sideA during overlay generation. The priority arrays describe the priorities of Grids at the overlapping region. Flux contributions at the overlapping region are computed from the Grid of the highest priority.

[sideBPriority] priority of Grids on sideB during overlay generation The priority arrays describe the priorities of Grids at the overlapping region. Flux contributions at the overlapping region are computed from the Grid of the highest priority.

[storeOverlay] Setting the storeOverlay optional argument to .false. (default) allows a user to bypass internal calculation of the fully unstructured grid and its storage.

[offline] Turn on offline XGrid creation and will use user supplied Sparse MatMul, area, centroid information.

[sparseMatA2X] indexlist from a Grid index space on side A to xgrid index space; indexFactorlist from a Grid index space on side A to xgrid index space.

[sparseMatX2A] indexlist from xgrid index space to a Grid index space on side A; indexFactorlist from xgrid index space to a Grid index space on side A.

[sparseMatB2X] indexlist from a Grid index space on side B to xgrid index space; indexFactorlist from a Grid index space on side B to xgrid index space.

[sparseMatX2B] indexlist from xgrid index space to a Grid index space on side B; indexFactorlist from xgrid index space to a Grid index space on side B.

[area] area of the xgrid cells.

[centroid] coordinates at the area weighted center of the xgrid cells.

[name] name of the xgrid object.

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

30.6.5 ESMF_XGridDestroy - Release resources associated with an XGrid

INTERFACE:

```
   subroutine ESMF_XGridDestroy(xgrid, keywordenforcer, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroys an `ESMF_XGrid`, releasing the resources associated with the object.
The arguments are:

xgrid `ESMF_XGrid` object.

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

30.6.6 ESMF_XGridGet - Get default information from an XGrid

INTERFACE:

```
   ! Private name; call using ESMF_XGridGet()

 subroutine ESMF_XGridGetDefault(xgrid, &
     sideA, sideB, ngridA, ngridB, area, centroid, &
     distgridA, distgridB, distgridM, &
     dimCount, localDECount, &
     sparseMatA2X, sparseMatX2A, sparseMatB2X, sparseMatX2B, &
     name, &
     rc)
```

ARGUMENTS:

```
 type(ESMF_XGrid), intent(in)                :: xgrid
 type(ESMF_Grid), intent(out), optional      :: sideA(:), sideB(:)
 integer, intent(out), optional              :: ngridA, ngridB
 real*8, intent(out), optional               :: area(:)
 real*8, intent(out), optional               :: centroid(:,:)
 type(ESMF_DistGrid), intent(out), optional  :: distgridA(:)
 type(ESMF_DistGrid), intent(out), optional  :: distgridB(:)
 type(ESMF_DistGrid), intent(out), optional  :: distgridM
 integer, intent(out), optional              :: dimCount
 integer, intent(out), optional              :: localDECount
 type(ESMF_XGridSpec), intent(out), optional :: sparseMatA2X(:)
 type(ESMF_XGridSpec), intent(out), optional :: sparseMatX2A(:)
 type(ESMF_XGridSpec), intent(out), optional :: sparseMatB2X(:)
 type(ESMF_XGridSpec), intent(out), optional :: sparseMatX2B(:)
 character (len=*), intent(out), optional    :: name
 integer, intent(out), optional              :: rc
```

DESCRIPTION:

Get information about XGrid
The arguments are:

xgrid The xgrid object used to retrieve information from.

[sideA] 2D Grids on side A

[sideB] 2D Grids on side B

[ngridA] Number of grids on the A side

[ngridB] Number of grids on the B side

[area] area of the xgrid cells

[centroid] coordinates at the area weighted center of the xgrid cells

[distgridA] list of distgrids whose sequence index list is an overlap between a Grid on sideA and the xgrid object.

[distgridB] list of distgrids whose sequence index list is an overlap between a Grid on sideB and the xgrid object.

[distgridM] the distgrid whose sequence index list fully describes the xgrid object.

[dimCount] dimension of the xgrid

[localDECount] number of local DEs on local PET

[sparseMatA2X] indexlist from a Grid index space on side A to xgrid index space indexFactorlist from a Grid index space on side A to xgrid index space

[sparseMatX2A] indexlist from xgrid index space to a Grid index space on side A indexFactorlist from xgrid index space to a Grid index space on side A

[sparseMatB2X] indexlist from a Grid index space on side B to xgrid index space indexFactorlist from a Grid index space on side B to xgrid index space

[sparseMatX2B] indexlist from xgrid index space to a Grid index space on side B indexFactorlist from xgrid index space to a Grid index space on side B

[name] name of the xgrid object.

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

30.6.7 ESMF_XGridGet - Get an individual DistGrid

INTERFACE:

```
   ! Private name; call using ESMF_XGridGet()

 subroutine ESMF_XGridGetDG(xgrid, distgrid, xgridside, gridindex, &
     rc)
```

ARGUMENTS:

```
 type(ESMF_XGrid), intent(in)                  :: xgrid
 type(ESMF_DistGrid), intent(out)              :: distgrid
 type(ESMF_XGridSide_Flag), intent(in), optional   :: xgridside
 integer, intent(in), optional                 :: gridindex
 integer, intent(out), optional                :: rc
```

DESCRIPTION:

Get a distgrid from XGrid from a specific side.
The arguments are:

xgrid The xgrid object used to retrieve information from.

distgrid Distgrid whose sequence index list is an overlap between gridIndex-th Grid on xgridSide and the xgrid object.

[xgridside] Which side of the XGrid to retrieve the distgrid from (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index selects the Distgrid associated with the Grid on that side. If not provided, defaults to 1.

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

30.6.8 ESMF_XGridGet - Get information about an XGrid

INTERFACE:

```
   ! Private name; call using ESMF_XGridGet()

 subroutine ESMF_XGridGetEle(xgrid, &
     localDE, elementCount, &
     exclusiveCount, exclusiveLBound, exclusiveUBound, &
     rc)
```

ARGUMENTS:

```
 type(ESMF_XGrid), intent(in)                 :: xgrid
 integer, intent(in)                          :: localDE
 integer, intent(out), optional               :: elementCount
 integer, intent(out), optional               :: exclusiveCount
 integer, intent(out), optional               :: exclusiveLBound
 integer, intent(out), optional               :: exclusiveUBound
 integer, intent(out), optional               :: rc
```

DESCRIPTION:

Get localDE specific information about XGrid
The arguments are:

xgrid The xgrid object used to retrieve information from.

localDE Local DE for which information is requested. [0,..,localDeCount-1]

[elementCount] Number of elements in exclusive region per DE

[exclusiveLBound] Lower bound of sequence indices in exclusive region per DE

[exclusiveUBound] Upper bound of sequence indices in exclusive region per DE

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

31 DistGrid Class

31.1 Description

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

Fortran API of higher classes based on DistGrid. The rank of a DELayout object contained within a DistGrid object must be equal to the DistGrid rank. Higher class objects that use the DistGrid, such as an Array object, may be of different rank than the associated DistGrid object. The higher class object will hold the mapping information between its dimensions and the DistGrid dimensions.

31.2 Constants

31.2.1 ESMF_DISTGRIDMATCH

DESCRIPTION:
Indicates the level to which two DistGrid variables match.
The type of this flag is:
`type(ESMF_DistGridMatch_Flag)`
The valid values are:

ESMF_DISTGRIDMATCH_INVALID: Indicates a non-valid matching level.

ESMF_DISTGRIDMATCH_NONE: The lowest valid level of DistGrid matching. This indicates that the DistGrid variables don't match at any of the higher levels.

ESMF_DISTGRIDMATCH_EXACT: All the DistGrid pieces except the name match between the two DistGrid variables.

ESMF_DISTGRIDMATCH_ALIAS: Both DistGrid variables are aliases to the exact same DistGrid object in memory.

31.3 Use and Examples

The following examples demonstrate how to create, use and destroy DistGrid objects. In order to produce complete and valid DistGrid objects all of the `ESMF_DistGridCreate()` calls require to be called in unison i.e. on *all* PETs of a component with a complete set of valid arguments.

31.3.1 Single tile DistGrid with regular decomposition

The minimum information required to create an `ESMF_DistGrid` object for a single tile with default decomposition are the corners of the tile in index space. The following call will create a 1D DistGrid for a 1D index space tile with elements from 1 through 1000.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/1000/), rc=rc)
```

A default DELayout with 1 DE per PET will be created during `ESMF_DistGridCreate()`. The 1000 elements of the specified 1D tile will then be block decomposed across the available DEs, i.e. across all PETs. Hence, for 4 PETs the (min) $\sim$ (max) corners of the DE-local LR regions will be:

```
DE 0 - (1) ~ (250)
DE 1 - (251) ~ (500)
DE 2 - (501) ~ (750)
DE 3 - (751) ~ (1000)
```

DistGrids with rank > 1 can also be created with default decompositions, specifying only the corners of the tile. The following will create a 2D DistGrid for a 5x5 tile with default decomposition.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), rc=rc)
```

The default decomposition for a DistGrid of rank N will be $(nDEs \times 1 \times ... \times 1)$, where $nDEs$ is the number of DEs in the DELayout and there are $N-1$ factors of 1. For the 2D example above this means a 4×1 regular decomposition if executed on 4 PETs and will result in the following DE-local LR regions:

```
DE 0 - (1,1) ~ (2,5)
DE 1 - (3,1) ~ (3,5)
DE 2 - (4,1) ~ (4,5)
DE 3 - (5,1) ~ (5,5)
```

In many cases the default decomposition will not suffice for higher rank DistGrids (rank > 1). For this reason a decomposition descriptor `regDecomp` argument is available during `ESMF_DistGridCreate()`. The following call creates a DistGrid on the same 2D tile as before, but now with a user specified regular decomposition of $2 \times 3 = 6$ DEs.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

The default DE labeling sequence follows column major order for the `regDecomp` argument:

```
 -----------> 2nd dimension
 |  0  2  4
 |  1  3  5
 v
1st dimension
```

By default grid points along all dimensions are homogeneously divided between the DEs. The maximum element count difference between DEs along any dimension is 1. The (min) $\sim$ (max) corners of the DE-local LR domains of the above example are as follows:

```
DE 0 - (1,1) ~ (3,2)
DE 1 - (4,1) ~ (5,2)
DE 2 - (1,3) ~ (3,4)
DE 3 - (4,3) ~ (5,4)
DE 4 - (1,5) ~ (3,5)
DE 5 - (4,5) ~ (5,5)
```

The specifics of the tile decomposition into DE-local LR domains can be modified by the optional `decompflag` argument. The following line shows how this argument is used to keep ESMF's default decomposition in the first dimension but move extra grid points of the second dimension to the last DEs in that direction. Extra elements occur if the number of DEs for a certain dimension does not evenly divide its extent. In this example there are 2 extra grid points for the second dimension because its extent is 5 but there are 3 DEs along this index space axis.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), decompflag=(/ESMF_DECOMP_DEFAULT, &
  ESMF_DECOMP_RESTLAST/), rc=rc)
```

Now DE 4 and DE 5 will hold the extra elements along the 2nd dimension.

```
DE 0 - (1,1) ~ (3,1)
DE 1 - (4,1) ~ (5,1)
DE 2 - (1,2) ~ (3,2)
DE 3 - (4,2) ~ (5,2)
DE 4 - (1,3) ~ (3,5)
DE 5 - (4,3) ~ (5,5)
```

An alternative way of indicating the DE-local LR regions is to list the index space coordinate as given by the associated DistGrid tile for each dimension. For this 2D example there are two lists (dim 1) / (dim 2) for each DE:

```
DE 0 - (1,2,3) / (1)
DE 1 - (4,5)   / (1)
DE 2 - (1,2,3) / (2)
DE 3 - (4,5)   / (2)
DE 4 - (1,2,3) / (3,4,5)
DE 5 - (4,5)   / (3,4,5)
```

Information about DE-local LR regions in the latter format can be obtained from the DistGrid object by use of `ESMF_DistGridGet()` methods:

```
allocate(dimExtent(2, 0:5)) ! (dimCount, deCount)
call ESMF_DistGridGet(distgrid, delayout=delayout, &
  indexCountPDe=dimExtent, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
call ESMF_DELayoutGet(delayout, localDeCount=localDeCount, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
allocate(localDeList(0:localDeCount-1))
call ESMF_DELayoutGet(delayout, localDeList=localDeList, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
do localDe=0, localDeCount-1
  de = localDeList(localDe)
  do dim=1, 2
    allocate(localIndexList(dimExtent(dim, de))) ! allocate list
                                                 ! to hold indices
    call ESMF_DistGridGet(distgrid, localDe=localDe, dim=dim, &
      indexList=localIndexList, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
    print *, "local DE ", localDe," - DE ",de, &
      " localIndexList along dim=", dim," :: ", localIndexList
    deallocate(localIndexList)
  enddo
enddo
deallocate(localDeList)
deallocate(dimExtent)
```

The advantage of the `localIndexList` format over the min-/max-corner format is that it can be used directly for DE-local to tile index dereferencing. Furthermore the `localIndexList` allows to express very general decompositions such as the cyclic decompositions in the first dimension generated by the following call:

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), &
  decompflag=(/ESMF_DECOMP_CYCLIC,ESMF_DECOMP_RESTLAST/), rc=rc)
```

with decomposition:

```
DE 0 - (1,3,5) / (1)
DE 1 - (2,4)   / (1)
DE 2 - (1,3,5) / (2)
DE 3 - (2,4)   / (2)
DE 4 - (1,3,5) / (3,4,5)
DE 5 - (2,4)   / (3,4,5)
```

Finally, a DistGrid object is destroyed by calling

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

31.3.2 DistGrid and DELayout

The examples of this section use the 2D DistGrid of the previous section to show the interplay between DistGrid and DELayout. By default, i.e. without specifying the `delayout` argument, a DELayout will be created during DistGrid creation that provides as many DEs as the DistGrid object requires. The implicit call to `ESMF_DELayoutCreate()` is issued with a fixed number of DEs and default settings in all other aspects. The resulting DE to PET mapping depends on the number of PETs of the current VM context. Assuming 6 PETs in the VM

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

will result in the following domain decomposition in terms of DEs

```
0  2  4
1  3  5
```

and their layout or distribution over the available PETs:

```
DE 0  -> PET 0
DE 1  -> PET 1
DE 2  -> PET 2
DE 3  -> PET 3
DE 4  -> PET 4
DE 5  -> PET 5
```

Running the same example on a 4 PET VM will not change the domain decomposition into 6 DEs as specified by

```
0  2  4
1  3  5
```

but the layout across PETs will now contain multiple DE-to-PET mapping with default cyclic distribution:

```
DE 0  -> PET 0
DE 1  -> PET 1
DE 2  -> PET 2
DE 3  -> PET 3
DE 4  -> PET 0
DE 5  -> PET 1
```

Sometimes it may be desirable for performance tuning to construct a DELayout with specific characteristics. For instance, if the 6 PETs of the above example are running on 3 nodes of a dual-SMP node cluster and there is a higher communication load along the first dimension of the model than along the second dimension it would be sensible to place DEs according to this knowledge.
The following example first creates a DELayout with 6 DEs where groups of 2 DEs are to be in fast connection. This DELayout is then used to create a DistGrid.

```
delayout = ESMF_DELayoutCreate(deCount=6, deGrouping=(/(i/2,i=0,5)/), rc=rc)
```

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), delayout=delayout, rc=rc)
```

This will ensure a distribution of DEs across the cluster resource in the following way:

```
  0   2   4
  1   3   5
 SMP SMP SMP
```

The interplay between DistGrid and DELayout may at first seem complicated. The simple but important rule to understand is that DistGrid describes a domain decomposition and each domain is labeled with a DE number. The DELayout describes how these DEs are laid out over the compute resources of the VM, i.e. PETs. The DEs are purely logical elements of decomposition and may be relabeled to fit the algorithm or legacy code better. The following example demonstrates this by describing the exact same distribution of the domain data across the fictitious cluster of SMP-nodes with a different choice of DE labeling:

```
delayout = ESMF_DELayoutCreate(deCount=6, deGrouping=(/(mod(i,3),i=0,5)/), &
  rc=rc)


distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), deLabelList=(/0,3,1,4,2,5/), delayout=delayout, rc=rc)
```

Here the `deLabelList` argument changes the default DE label sequence from column major to row major. The DELayout compensates for this change in DE labeling by changing the `deGrouping` argument to map the first dimension to SMP nodes as before. The decomposition and layout now looks as follows:

```
  0   1   2
  3   4   5
 SMP SMP SMP
```

Finally, in order to achieve a completely user-defined distribution of the domain data across the PETs of the VM a DELayout may be created from a `petMap` before using it in the creation of a DistGrid. If for instance the desired distribution of a 2 x 3 decomposition puts the DEs of the first row onto 3 separate PETs (PET 0, 1, 2) and groups the DEs of the second row onto PET 3 a `petMap` must first be setup that takes the DE labeling of the DistGrid into account.The following lines of code result in the desired distribution using column major DE labeling by first create a DELayout and then using it in the DistGrid creation.

```
delayout = ESMF_DELayoutCreate(petMap=(/0,3,1,3,2,3/), rc=rc)


distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), delayout=delayout, rc=rc)
```

This decomposes the global domain into

```
  0   2   4
  1   3   5
```

and associates the DEs to the following PETs:

```
 DE 0  -> PET 0
 DE 1  -> PET 3
 DE 2  -> PET 1
 DE 3  -> PET 3
 DE 4  -> PET 2
 DE 5  -> PET 3
```

31.3.3 Single tile DistGrid with decomposition by DE blocks

The examples of the previous sections showed how DistGrid objects with regular decompositions are created. However, in some cases a regular decomposition may not be specific enough. The following example shows how the `deBlockList` argument is used to create a DistGrid object with completely user-defined decomposition.
A single 5x5 LR domain is to be decomposed into 6 DEs. To this end a list is constructed that holds the min and max corners of all six DE LR blocks. The DE-local LR blocks are arranged as to cover the whole tile domain without overlap.

```
allocate(deBlockList(2, 2, 6))  ! (dimCount, 2, deCount)
deBlockList(:,1,1) = (/1,1/)  ! minIndex  1st deBlock
deBlockList(:,2,1) = (/3,2/)  ! maxIndex  1st deBlock
deBlockList(:,1,2) = (/4,1/)  ! minIndex  2nd deBlock
deBlockList(:,2,2) = (/5,2/)  ! maxIndex  2nd deBlock
deBlockList(:,1,3) = (/1,3/)
deBlockList(:,2,3) = (/2,4/)
deBlockList(:,1,4) = (/3,3/)
deBlockList(:,2,4) = (/5,4/)
deBlockList(:,1,5) = (/1,5/)
deBlockList(:,2,5) = (/3,5/)
deBlockList(:,1,6) = (/4,5/)  ! minIndex  6th deBlock
deBlockList(:,2,6) = (/5,5/)  ! maxInbex  6th deBlock


distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  deBlockList=deBlockList, rc=rc)
```

31.3.4 Single tile DistGrid with periodic boundaries

By default the edges of all tiles have solid wall boundary conditions. Periodic boundary conditions can be imposed by specifying connections between tiles. For the single LR domain of the last section periodic boundaries along the first dimension are imposed by adding a `connectionList` argument with only one element to the create call.

```
allocate(connectionList(1))
```

The connection element holds information about `tileIndex_A`, `tileIndex_B`, `positionVector`, and `orientationVector/)`.

```
call ESMF_DistGridConnectionSet(connection=connectionList(1), &
   tileIndexA=1, tileIndexB=1, &
   positionVector=(/5, 0/), &
   orientationVector=(/1, 2/), &
   rc=rc)
```

The `tileIndexA` and `tileIndexB` arguments specify that this is a connection within tile 1. The `positionVector` indicates that there is no offset between tileB and tileA along the second dimension, but there is an offset of 5 along the first dimension (which in this case is the length of dimension 1). This aligns tileB (which is tile 1) right next to tileA (which is also tile 1).
The `orientationVector` fixes the orientation of the tileB index space to be the same as the orientation of tileA (it maps index 1 of tileA to index 1 of tileB and the same for index 2). The `orientationVector` could have been omitted in this case which corresponds to the default orientation.
The `connectionList` can now be used to create a `DistGrid` object with the desired boundary conditions.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  deBlockList=deBlockList, connectionList=connectionList, rc=rc)
```

```
deallocate(connectionList)
```

This closes the tile along the first dimension on itself, thus imposing periodic boundaries along this direction.

31.3.5 2D tilework DistGrid with regular decomposition

Creating a DistGrid from a list of LR domains is a straight forward extension of the case with a single LR domain. The first four arguments of `ESMF_DistGridCreate()` are promoted to rank 2, the second dimension being the tile count index.
The following 2D tilework domain consisting of 3 LR tiles will be used in the examples of this section:

```
 ----------------------------------------> 2nd dim
 |
 |                   (1,11)-----(1,20)
 |                   |               |
 |                   |               |
 |                   |               |
 |                   |               |
 |                   |               |
 |                   (10,11)---(10,20)
 |  (11,1)----(11,10)(11,11)---(11,20)
 |  |               ||               |
 |  |               ||               |
 |  |               ||               |
 |  |               ||               |
 |  |               ||               |
 |  (20,1)----(20,10)(20,11)---(20,20)
 |
 |
 v
1st dim
```

The first step in creating a tilework global domain is to construct the `minIndex` and `maxIndex` arrays.

```
allocate(minIndex(2,3))    ! (dimCount, number of tiles)
allocate(maxIndex(2,3))    ! (dimCount, number of tiles)
minIndex(:,1) = (/11,1/)
maxIndex(:,1) = (/20,10/)
minIndex(:,2) = (/11,11/)
maxIndex(:,2) = (/20,20/)
minIndex(:,3) = (/1,11/)
maxIndex(:,3) = (/10,20/)
```

Next the regular decomposition for each tile is set up in the `regDecomp` array. In this example each tile is associated with a single DE.

```
allocate(regDecomp(2,3))    ! (dimCount, number of tiles)
regDecomp(:,1) = (/1,1/)    ! one DE
regDecomp(:,2) = (/1,1/)    ! one DE
regDecomp(:,3) = (/1,1/)    ! one DE
```

Finally the DistGrid can be created by calling

```
  distgrid = ESMF_DistGridCreate(minIndexPTile=minIndex, &
    maxIndexPTile=maxIndex, regDecompPTile=regDecomp, rc=rc)
```

The default DE labeling sequence is identical to the tile labeling sequence and follows the sequence in which the tiles are defined during the create call. However, DE labels start at 0 whereas tile labels start at 1. In this case the DE labels look as:

```
          2
      0   1
```

Each tile can be decomposed differently into DEs. The default DE labeling follows the column major order for each tile. This is demonstrated in the following case where the tilework global domain is decomposed into 9 DEs,

```
  regDecomp(:,1) = (/2,2/)    ! 4 DEs
  regDecomp(:,2) = (/1,3/)    ! 3 DEs
  regDecomp(:,3) = (/2,1/)    ! 2 DEs

  distgrid = ESMF_DistGridCreate(minIndexPTile=minIndex, &
    maxIndexPTile=maxIndex, regDecompPTile=regDecomp, rc=rc)
```

resulting in the following decomposition:

```
             +-------+
             |   7   |
             |       |
             |   8   |
     +-------+-------+
     | 0   2 |       |
     |       | 4 5 6 |
     | 1   3 |       |
     +-------+-------+


     DE 0 - (11,1)  ~ (15,5)
     DE 1 - (16,1)  ~ (20,5)
     DE 2 - (11,6)  ~ (15,10)
     DE 3 - (16,6)  ~ (20,10)
     DE 4 - (11,11) ~ (20,14)
     DE 5 - (11,15) ~ (20,17)
     DE 6 - (11,18) ~ (20,20)
     DE 7 - (1,11)  ~ (5,20)
     DE 8 - (6,11)  ~ (10,20)
```

The `decompflag` and `deLabelList` arguments can be used much like in the single LR domain case to overwrite the default grid decomposition (per tile) and to change the overall DE labeling sequence, respectively.

31.3.6 Arbitrary DistGrids with user-supplied sequence indices

The DistGrid class supports the communication methods of higher classes, like Array and Field, by associating a unique *sequence index* with each DistGrid index tuple. This sequence index can be used to address every Array or Field element. By default, the DistGrid does not actually generate and store the sequence index of each element. Instead a default sequence through the elements is implemented in the DistGrid code. This default sequence is used internally when needed.

The DistGrid class provides two `ESMF_DistGridCreate()` calls that allow the user to specify arbitrary sequence indices, overriding the use of the default sequence index scheme. The user sequence indices are passed to the DistGrid in form of 1d Fortran arrays, one array on each PET. The local size of this array on each PET determines the number of DistGrid elements on the PET. The supplied sequence indices must be unique across all PETs.

```
allocate(arbSeqIndexList(10))   ! each PET will have 10 elements

do i=1, 10
  arbSeqIndexList(i) = (i-1)*petCount + localPet ! initialize unique
                                                 ! seq. indices
enddo
```

A default DELayout will be created automatically during `ESMF_DistGridCreate()`, associating 1 DE per PET.

```
distgrid = ESMF_DistGridCreate(arbSeqIndexList=arbSeqIndexList, rc=rc)
```

The user provided sequence index array can be deallocated once it has been used.

```
deallocate(arbSeqIndexList)
```

The `distgrid` object can be used just like any other DistGrid object. The "arbitrary" nature of `distgrid` will only become visible during Array or Field communication methods, where source and destination objects map elements according to the sequence indices provided by the associated DistGrid objects.

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

The second `ESMF_DistGridCreate()` call, that accepts the `arbSeqIndexList` argument, allows the user to specify additional, regular DistGrid dimensions. These additional DistGrid dimensions are not decomposed across DEs, but instead are simply "added" or "multiplied" to the 1D arbitrary dimension.
The same `arbSeqIndexList` array as before is used to define the user supplied sequence indices.

```
allocate(arbSeqIndexList(10))   ! each PET will have 10 elements

do i=1, 10
  arbSeqIndexList(i) = (i-1)*petCount + localPet  ! initialize unique
                                                  ! seq. indices
enddo
```

The additional DistGrid dimensions are specified in the usual manner using `minIndex` and `maxIndex` arguments. The `dimCount` of the resulting DistGrid is the size of the `minIndex` and `maxIndex` arguments plus 1 for the arbitrary dimension. The `arbDim` argument is used to indicate which or the resulting DistGrid dimensions is associated with the arbitrary sequence indices provided by the user.

```
distgrid = ESMF_DistGridCreate(arbSeqIndexList=arbSeqIndexList, &
  arbDim=1, minIndexPTile=(/1,1/), maxIndexPTile=(/5,7/), rc=rc)
```

```
deallocate(arbSeqIndexList)
```

```
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

31.4 Restrictions and Future Work

- Multi-tile DistGrids from deBlockList are not yet supported.
- The fastAxis feature has not been implemented yet.

31.5 Design and Implementation Notes

This section will be updated as the implementation of the DistGrid class nears completion.

31.6 Class API

31.6.1 ESMF_DistGridAssignment(=) - DistGrid assignment

INTERFACE:

```
interface assignment(=)
distgrid1 = distgrid2
```

ARGUMENTS:

```
type(ESMF_DistGrid) :: distgrid1
type(ESMF_DistGrid) :: distgrid2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign distgrid1 as an alias to the same ESMF DistGrid object in memory as distgrid2. If distgrid2 is invalid, then distgrid1 will be equally invalid after the assignment.
The arguments are:

distgrid1 The `ESMF_DistGrid` object on the left hand side of the assignment.

distgrid2 The `ESMF_DistGrid` object on the right hand side of the assignment.

31.6.2 ESMF_DistGridOperator(==) - DistGrid equality operator

INTERFACE:

```
interface operator(==)
  if (distgrid1 == distgrid2) then ... endif
            OR
  result = (distgrid1 == distgrid2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid1
type(ESMF_DistGrid), intent(in) :: distgrid2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether distgrid1 and distgrid2 are valid aliases to the same ESMF DistGrid object in memory. For a more general comparison of two ESMF DistGrids, going beyond the simple alias test, the `ESMF_DistGridMatch()` function (not yet fully implemented) must be used.
The arguments are:

distgrid1 The `ESMF_DistGrid` object on the left hand side of the equality operation.

distgrid2 The `ESMF_DistGrid` object on the right hand side of the equality operation.

31.6.3 ESMF_DistGridOperator(/=) - DistGrid not equal operator

INTERFACE:

```
interface operator(/=)
  if (distgrid1 /= distgrid2) then ... endif
            OR
  result = (distgrid1 /= distgrid2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid1
type(ESMF_DistGrid), intent(in) :: distgrid2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether distgrid1 and distgrid2 are *not* valid aliases to the same ESMF DistGrid object in memory. For a more general comparison of two ESMF DistGrids, going beyond the simple alias test, the `ESMF_DistGridMatch()` function (not yet fully implemented) must be used.
The arguments are:

distgrid1 The `ESMF_DistGrid` object on the left hand side of the non-equality operation.

distgrid2 The `ESMF_DistGrid` object on the right hand side of the non-equality operation.

31.6.4 ESMF_DistGridCreate - Create DistGrid object from DistGrid

INTERFACE:

```
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateDG(distgrid, &
  firstExtra, lastExtra, indexflag, connectionList, rc)
```

ARGUMENTS:

```
    type(ESMF_DistGrid),           intent(in)            :: distgrid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer, target,               intent(in),  optional :: firstExtra(:)
    integer, target,               intent(in),  optional :: lastExtra(:)
    type(ESMF_Index_Flag),         intent(in),  optional :: indexflag
    type(ESMF_DistGridConnection), intent(in),  optional :: connectionList(:)
    integer,                       intent(out), optional :: rc
```

RETURN VALUE:

```
    type(ESMF_DistGrid) :: ESMF_DistGridCreateDG
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create a new DistGrid from an existing DistGrid, keeping the decomposition unchanged. The `firstExtra` and `lastExtra` arguments allow extra elements to be added at the first/last edge DE in each dimension. The method also allows the `indexflag` to be set. Further, if the `connectionList` argument is provided it will be used to set connections in the newly created DistGrid, otherwise the connections of the incoming DistGrid will be used. If neither `firstExtra`, `lastExtra`, `indexflag`, nor `connectionList` arguments are specified, the method reduces to a deep copy of the incoming DistGrid object.
The arguments are:

distgrid Incoming DistGrid object.

[firstExtra] Extra elements on the edge of the first DEs along each dimension. The default is a zero vector.

[lastExtra] Extra elements on the edge of the last DEs along each dimension. The default is a zero vector.

[indexflag] Indicates whether the indices provided by the `minIndex` and `maxIndex` arguments are to be interpreted to form a global index space or not. The default is `ESMF_INDEX_DELOCAL`. See section 9.24 for a complete list of options.

[connectionList] List of `ESMF_DistGridConnection` objects, defining connections between DistGrid tiles in index space. See section 31.7.1 for the associated Set() method.

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

31.6.5 ESMF_DistGridCreate - Create DistGrid object from DistGrid

INTERFACE:

```
  ! Private name; call using ESMF_DistGridCreate()
  function ESMF_DistGridCreateDGT(distgrid, firstExtraPTile, &
    lastExtraPTile, indexflag, connectionList, rc)
```

ARGUMENTS:

```
    type(ESMF_DistGrid),        intent(in)            :: distgrid
    integer, target,            intent(in)            :: firstExtraPTile(:,:)
    integer, target,            intent(in)            :: lastExtraPTile(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Index_Flag),      intent(in),  optional :: indexflag
    type(ESMF_DistGridConnection), intent(in),  optional :: connectionList(:)
    integer,                    intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_DistGrid) :: ESMF_DistGridCreateDGT
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create a new DistGrid from an existing DistGrid, keeping the decomposition unchanged. The `firstExtraPTile` and `lastExtraPTile` arguments allow extra elements to be added at the first/last edge DE in each dimension. The method also allows the `indexflag` to be set. Further, if the `connectionList` argument provided in it will be used to set connections in the newly created DistGrid, otherwise the connections of the incoming DistGrid will be used. If neither `firstExtraPTile`, `lastExtraPTile`, `indexflag`, nor `connectionList` arguments are specified, the method reduces to a deep copy of the incoming DistGrid object.
The arguments are:

distgrid Incoming DistGrid object.

firstExtraPTile Extra elements on the edge of the first DEs along each dimension.

lastExtraPTile Extra elements on the edge of the last DEs along each dimension.

[indexflag] Indicates whether the indices provided by the `minIndex` and `maxIndex` arguments are to be interpreted to form a global index space or not. The default is `ESMF_INDEX_DELOCAL`. See section 9.24 for a complete list of options.

[connectionList] List of `ESMF_DistGridConnection` objects, defining connections between DistGrid tiles in index space. See section 31.7.1 for the associated Set() method.

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

31.6.6 ESMF_DistGridCreate - Create DistGrid object with regular decomposition

INTERFACE:

```
   ! Private name; call using ESMF_DistGridCreate()
   function ESMF_DistGridCreateRD(minIndex, maxIndex, regDecomp, &
     decompflag, regDecompFirstExtra, regDecompLastExtra, deLabelList, &
     indexflag, connectionList, delayout, vm, rc)
```

ARGUMENTS:

```
     integer,                   intent(in)            :: minIndex(:)
     integer,                   intent(in)            :: maxIndex(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,      target,      intent(in),  optional :: regDecomp(:)
     type(ESMF_Decomp_Flag), target, intent(in),  optional :: decompflag(:)
     integer,      target,      intent(in),  optional :: regDecompFirstExtra(:)
     integer,      target,      intent(in),  optional :: regDecompLastExtra(:)
     integer,      target,      intent(in),  optional :: deLabelList(:)
     type(ESMF_Index_Flag),     intent(in),  optional :: indexflag
     type(ESMF_DistGridConnection),  intent(in),  optional :: connectionList(:)
     type(ESMF_DELayout),       intent(in),  optional :: delayout
     type(ESMF_VM),             intent(in),  optional :: vm
     integer,                   intent(out), optional :: rc
```

RETURN VALUE:

```
type(ESMF_DistGrid) :: ESMF_DistGridCreateRD
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_DistGrid` from a single logically rectangular (LR) tile with regular decomposition. A regular decomposition is of the same rank as the tile and decomposes each dimension into a fixed number of DEs. A regular decomposition of a single tile is expressed by a single `regDecomp` list of DE counts in each dimension.
The arguments are:

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

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

[regDecomp] List of DE counts for each dimension. The default decomposition will be `deCount`$\times 1 \times ... \times 1$. The value of `deCount` for a default DELayout equals `petCount`, i.e. the default decomposition will be into as many DEs as there are PETs and the distribution will be 1 DE per PET.

[decompflag] List of decomposition flags indicating how each dimension of the tile is to be divided between the DEs. The default setting is `ESMF_DECOMP_BALANCED` in all dimensions. See section 9.13 for a list of valid decomposition options.

[regDecompFirstExtra] Extra elements on the first DEs along each dimension in a regular decomposition. The default is a zero vector.

[regDecompLastExtra] Extra elements on the last DEs along each dimension in a regular decomposition. The default is a zero vector.

[deLabelList] List assigning DE labels to the default sequence of DEs. The default sequence is given by the column major order of the `regDecomp` argument.

[indexflag] Indicates whether the indices provided by the `minIndex` and `maxIndex` arguments are to be interpreted to form a global index space or not. The default is `ESMF_INDEX_DELOCAL`. See section 9.24 for a complete list of options.

[connectionList] List of `ESMF_DistGridConnection` objects, defining connections between DistGrid tiles in index space. See section 31.7.1 for the associated Set() method.

[delayout] Optional `ESMF_DELayout` object to be used. By default a new DELayout object will be created with the correct number of DEs. If a DELayout object is specified its number of DEs must match the number indicated by `regDecomp`.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

31.6.7 ESMF_DistGridCreate - Create DistGrid object with DE blocks

INTERFACE:

```
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateDB(minIndex, maxIndex, deBlockList, &
  deLabelList, indexflag, connectionList, delayout, vm, rc)
```

ARGUMENTS:

```
     integer,                        intent(in)            :: minIndex(:)
     integer,                        intent(in)            :: maxIndex(:)
     integer,                        intent(in)            :: deBlockList(:,:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                        intent(in),  optional :: deLabelList(:)
     type(ESMF_Index_Flag),          intent(in),  optional :: indexflag
     type(ESMF_DistGridConnection), intent(in),  optional ::connectionList(:)
     type(ESMF_DELayout),            intent(in),  optional :: delayout
     type(ESMF_VM),                  intent(in),  optional :: vm
     integer,                        intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_DistGrid) :: ESMF_DistGridCreateDB
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_DistGrid` from a single logically rectangular (LR) tile with decomposition specified by `deBlockList`.
The arguments are:

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

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

deBlockList List of DE-local LR blocks. The third index of `deBlockList` steps through the deBlock elements, which are defined by the first two indices. The first index must be of size `dimCount` and the second index must be of size 2. Each 2D element of `deBlockList` defined by the first two indices hold the following information.

```
 +---------------------------------------> 2nd index
 |    1               2
 | 1  minIndex(1)    maxIndex(1)
 | 2  minIndex(2)    maxIndex(2)
 | .  minIndex(.)    maxIndex(.)
 | .
 v
1st index
```

It is required that there be no overlap between the LR segments defined by deBlockList.

[deLabelList] List assigning DE labels to the default sequence of DEs. The default sequence is given by the column major order of the `regDecomp` argument.

[indexflag] Indicates whether the indices provided by the `minIndex` and `maxIndex` arguments are to be interpreted to form a global index space or not. The default is `ESMF_INDEX_DELOCAL`. See section 9.24 for a complete list of options.

[connectionList] List of `ESMF_DistGridConnection` objects, defining connections between DistGrid tiles in index space. See section 31.7.1 for the associated Set() method.

[delayout] Optional `ESMF_DELayout` object to be used. By default a new DELayout object will be created with the correct number of DEs. If a DELayout object is specified its number of DEs must match the number indicated by `regDecomp`.

[vm] Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method's overhead.

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

31.6.8 ESMF_DistGridCreate - Create DistGrid object from tilework with regular decomposition

INTERFACE:

```
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateRDT(minIndexPTile, maxIndexPTile, &
  regDecompPTile, decompflagPTile, regDecompFirstExtraPTile,&
  regDecompLastExtraPTile, deLabelList, indexflag, connectionList, &
  delayout, vm, rc)
```

ARGUMENTS:

```
     integer,             intent(in)            :: minIndexPTile(:,:)
     integer,             intent(in)            :: maxIndexPTile(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(in),  optional :: regDecompPTile(:,:)
     type(ESMF_Decomp_Flag), target, intent(in),  optional ::decompflagPTile(:,:)
     integer,     target, intent(in),  optional :: regDecompFirstExtraPTile(:,:)
     integer,     target, intent(in),  optional :: regDecompLastExtraPTile(:,:)
     integer,             intent(in),  optional :: deLabelList(:)
     type(ESMF_Index_Flag), intent(in),  optional :: indexflag
     type(ESMF_DistGridConnection), intent(in),  optional :: connectionList(:)
     type(ESMF_DELayout), intent(in),  optional :: delayout
     type(ESMF_VM),       intent(in),  optional :: vm
     integer,             intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_DistGrid) :: ESMF_DistGridCreateRDT
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an ESMF_DistGrid from a tilework of logically rectangular (LR) tiles with regular decomposition. A regular decomposition is of the same rank as the tile and decomposes each dimension into a fixed number of DEs. A regular decomposition of a tilework of tiles is expressed by a list of DE count vectors, one vector for each tile. Each vector contained in the regDecompPTile argument ascribes DE counts for each dimension. It is erroneous to provide more tiles than there are DEs.
The arguments are:

minIndexPTile The first index provides the global coordinate tuple of the lower corner of a tile. The second index indicates the tile number.

maxIndexPTile The first index provides the global coordinate tuple of the upper corner of a tile. The second index indicates the tile number.

[regDecompPTile] List of DE counts for each dimension. The second index indicates the tile number. The default decomposition will be deCount$\times 1 \times ... \times 1$. The value of deCount for a default DELayout equals petCount, i.e. the default decomposition will be into as many DEs as there are PETs and the distribution will be 1 DE per PET.

[decompflagPTile] List of decomposition flags indicating how each dimension of each tile is to be divided between the DEs. The default setting is ESMF_DECOMP_BALANCED in all dimensions for all tiles. See section 9.13 for a list of valid decomposition flag options. The second index indicates the tile number.

[regDecompFirstExtraPTile] Extra elements on the first DEs along each dimension in a regular decomposition. The default is a zero vector. The second index indicates the tile number.

[regDecompLastExtraPTile] Extra elements on the last DEs along each dimension in a regular decomposition. The default is a zero vector. The second index indicates the tile number.

[deLabelList] List assigning DE labels to the default sequence of DEs. The default sequence is given by the column major order of the regDecompPTile elements in the sequence as they appear following the tile index.

[indexflag] Indicates whether the indices provided by the minIndex and maxIndex arguments are to be interpreted to form a global index space or not. The default is ESMF_INDEX_DELOCAL. See section 9.24 for a complete list of options.

[connectionList] List of ESMF_DistGridConnection objects, defining connections between DistGrid tiles in index space. See section 31.7.1 for the associated Set() method.

[delayout] Optional ESMF_DELayout object to be used. By default a new DELayout object will be created with the correct number of DEs. If a DELayout object is specified its number of DEs must match the number indicated by regDecompPTile.

[vm] Optional ESMF_VM object of the current context. Providing the VM of the current context will lower the method's overhead.

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

31.6.9 ESMF_DistGridCreate - Create 1D DistGrid object from user's arbitray index list

INTERFACE:

```
   ! Private name; call using ESMF_DistGridCreate()
   function ESMF_DistGridCreateDBAI1D(arbSeqIndexList, rc)
```

ARGUMENTS:

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

RETURN VALUE:

```
     type(ESMF_DistGrid) :: ESMF_DistGridCreateDBAI1D
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an ESMF_DistGrid of dimCount 1 from a PET-local list of sequence indices. The PET-local size of the arbSeqIndexList argument determines the number of local elements in the created DistGrid. The sequence indices must be unique across all PETs. A default DELayout with 1 DE per PET across all PETs of the current VM is automatically created.
The arguments are:

arbSeqIndexList List of arbitrary sequence indices that reside on the local PET.

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

31.6.10 ESMF_DistGridCreate - Create (1+n)D DistGrid object from user's arbitray index list and minIndexPTile/maxIndexPTile

INTERFACE:

```
! Private name; call using ESMF_DistGridCreate()
function ESMF_DistGridCreateDBAI(arbSeqIndexList, arbDim, &
  minIndexPTile, maxIndexPTile, rc)
```

ARGUMENTS:

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

RETURN VALUE:

```
    type(ESMF_DistGrid) :: ESMF_DistGridCreateDBAI
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_DistGrid` of `dimCount` $1 + n$, where n = `size(minIndexPTile)` = `size(maxIndexPTile)`.
The resulting DistGrid will have a 1D distribution determined by the PET-local `arbSeqIndexList`. The PET-local size of the `arbSeqIndexList` argument determines the number of local elements along the arbitrarily distributed dimension in the created DistGrid. The sequence indices must be unique across all PETs. The associated, automatically created DELayout will have 1 DE per PET across all PETs of the current VM.
In addition to the arbitrarily distributed dimension, regular DistGrid dimensions can be specified in `minIndexPTile` and `maxIndexPTile`. The n dimensional subspace spanned by the regular dimensions is "multiplied" with the arbitrary dimension on each DE, to form a $1 + n$ dimensional total index space described by the DistGrid object. The `arbDim` argument allows to specify which dimension in the resulting DistGrid corresponds to the arbitrarily distributed one.
The arguments are:

arbSeqIndexList List of arbitrary sequence indices that reside on the local PET.

arbDim Dimension of the arbitrary distribution.

minIndexPTile Global coordinate tuple of the lower corner of the tile. The arbitrary dimension is *not* included in this tile

maxIndexPTile Global coordinate tuple of the upper corner of the tile. The arbitrary dimension is *not* included in this tile

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

31.6.11 ESMF_DistGridDestroy - Release resources associated with a DistGrid

INTERFACE:

```
subroutine ESMF_DistGridDestroy(distgrid, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroys an `ESMF_DistGrid`, releasing the resources associated with the object.
The arguments are:

distgrid `ESMF_DistGrid` object to be destroyed.

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

31.6.12 ESMF_DistGridGet - Get information about DistGrid object

INTERFACE:

```
! Private name; call using ESMF_DistGridGet()
subroutine ESMF_DistGridGetDefault(distgrid, delayout, dimCount, &
  tileCount, minIndexPTile, maxIndexPTile, elementCountPTile, &
  minIndexPDe, maxIndexPDe, elementCountPDe, deToTileMap, &
  indexCountPDe, collocation, regDecompFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_DistGrid),    intent(in)            :: distgrid
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DELayout),    intent(out), optional :: delayout
  integer,                intent(out), optional :: dimCount
  integer,                intent(out), optional :: tileCount
  integer,        target, intent(out), optional :: minIndexPTile(:,:)
  integer,        target, intent(out), optional :: maxIndexPTile(:,:)
  integer,        target, intent(out), optional :: elementCountPTile(:)
  integer,        target, intent(out), optional :: minIndexPDe(:,:)
  integer,        target, intent(out), optional :: maxIndexPDe(:,:)
  integer,        target, intent(out), optional :: elementCountPDe(:)
  integer,        target, intent(out), optional :: deToTileMap(:)
  integer,        target, intent(out), optional :: indexCountPDe(:,:)
  integer,        target, intent(out), optional :: collocation(:)
  logical,                intent(out), optional :: regDecompFlag
  integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal DistGrid information.
The arguments are:

distgrid Queried `ESMF_DistGrid` object.

[delayout] `ESMF_DELayout` object associated with `distgrid`.

[dimCount] Number of dimensions (rank) of `distgrid`.

[tileCount] Number of tiles in `distgrid`.

[minIndexPTile] Lower index space corner per `dim`, per `tile`, with `size(minIndexPTile) == (/dimCount, tileCount/)`.

[maxIndexPTile] Upper index space corner per `dim`, per `tile`, with `size(minIndexPTile) == (/dimCount, tileCount/)`.

[elementCountPTile] Number of elements in exclusive region per tile, with `size(elementCountPTile) == (/tileCount/)`

[minIndexPDe] Lower index space corner per `dim`, per `De`, with `size(minIndexPDe) == (/dimCount, deCount/)`.

[maxIndexPDe] Upper index space corner per `dim`, per `de`, with `size(minIndexPDe) == (/dimCount, deCount/)`.

[elementCountPDe] Number of elements in exclusive region per DE, with `size(elementCountPDe) == (/deCount/)`

[deToTileMap] List of tile id numbers, one for each DE, with `size(deToTileMap) == (/deCount/)`

[indexCountPDe] Array of extents per `dim`, per `de`, with `size(indexCountPDe) == (/dimCount, deCount/)`.

[collocation] List of collocation id numbers, one for each dim, with `size(collocation) == (/dimCount/)`

[regDecompFlag] Flag equal to `ESMF_TRUE` for regular decompositions and equal to `ESMF_FALSE` otherwise.

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

31.6.13 ESMF_DistGridGet - Get DE local information about DistGrid

INTERFACE:

```
   ! Private name; call using ESMF_DistGridGet()
   subroutine ESMF_DistGridGetPLocalDe(distgrid, localDe, &
     collocation, arbSeqIndexFlag, seqIndexList, elementCount, rc)
```

ARGUMENTS:

```
     type(ESMF_DistGrid),    intent(in)            :: distgrid
     integer,                intent(in)            :: localDe
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),  optional :: collocation
     logical,                intent(out), optional :: arbSeqIndexFlag
     integer,        target, intent(out), optional :: seqIndexList(:)
     integer,                intent(out), optional :: elementCount
     integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal DistGrid information.
The arguments are:

distgrid Queried `ESMF_DistGrid` object.

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

[collocation] Collocation for which information is requested. Default to first collocation in `collocationPDim` list.

[arbSeqIndexFlag] Indicates whether collocation is associated with arbitrary sequence indices.

[seqIndexList] List of DistGrid tile-local sequence indices for `localDe`, with `size(seqIndexList) == (/elementCountPDe(localDe)/)`.

[elementCount] Number of elements in the localDe, i.e. identical to elementCountPDe(localDe).

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

31.6.14 ESMF_DistGridGet - Get DE local information for dimension about DistGrid

INTERFACE:

```
   ! Private name; call using ESMF_DistGridGet()
   subroutine ESMF_DistGridGetPLocalDePDim(distgrid, localDe, dim, &
            indexList, rc)
```

ARGUMENTS:

```
     type(ESMF_DistGrid),     intent(in)            :: distgrid
     integer,                 intent(in)            :: localDe
     integer,                 intent(in)            :: dim
     integer,         target, intent(out)           :: indexList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal DistGrid information.
The arguments are:

distgrid Queried `ESMF_DistGrid` object.

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

dim Dimension for which information is requested. `[1,..,dimCount]`

indexList Upon return this holds the list of DistGrid tile-local indices for `localDe` along dimension `dim`. The supplied variable must be at least of size `indexCountPDimPDe(dim, de(localDe))`.

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

31.6.15 ESMF_DistGridMatch - Check if two DistGrid objects match

INTERFACE:

```
   function ESMF_DistGridMatch(distgrid1, distgrid2, rc)
```

RETURN VALUE:

```
     type(ESMF_DistGridMatch_Flag) :: ESMF_DistGridMatch
```

ARGUMENTS:

```
     type(ESMF_DistGrid),  intent(in)            :: distgrid1
     type(ESMF_DistGrid),  intent(in)            :: distgrid2
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,              intent(out), optional :: rc
```

DESCRIPTION:

Determine to which level `distgrid1` and `distgrid2` match.
Returns a range of values of type `ESMF_DistGridMatch_Flag`, indicating how closely the DistGrids match. For a description of the possible return values, see 31.2.1. Note that this call only performs PET local matching. Different return values may be returned on different PETs for the same DistGrid pair.
The arguments are:

distgrid1 `ESMF_DistGrid` object.

distgrid2 `ESMF_DistGrid` object.

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

31.6.16 ESMF_DistGridPrint - Print DistGrid internals

INTERFACE:

```
   subroutine ESMF_DistGridPrint(distgrid, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Prints internal information about the specified `ESMF_DistGrid` object to `stdout`.

The arguments are:

distgrid Specified `ESMF_DistGrid` object.

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

31.6.17 ESMF_DistGridValidate - Validate DistGrid internals

INTERFACE:

```
   subroutine ESMF_DistGridValidate(distgrid, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the `distgrid` is internally consistent. The method returns an error code if problems are found.
The arguments are:

distgrid Specified `ESMF_DistGrid` object.

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

31.7 Class API: DistGridConnection Methods

31.7.1 ESMF_DistGridConnectionSet - Set DistGridConnetion

INTERFACE:

```
   subroutine ESMF_DistGridConnectionSet(connection, tileIndexA, tileIndexB, &
     positionVector, orientationVector, rc)
```

ARGUMENTS:

```
     type(ESMF_DistGridConnection), intent(out)           :: connection
     integer,                       intent(in)            :: tileIndexA
     integer,                       intent(in)            :: tileIndexB
     integer,                       intent(in)            :: positionVector(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                       intent(in),  optional :: orientationVector(:)
     integer,                       intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Set an `ESMF_DistGridConnection` object to represent a connection according to the provided index space information.
The arguments are:

connection DistGridConnection object.

tileIndexA Index of one of the two tiles that are to be connected.

tileIndexB Index of one of the two tiles that are to be connected.

positionVector Position of tile B's minIndex with respect to tile A's minIndex.

[orientationVector] Associates each dimension of tile A with a dimension in tile B's index space. Negative index values may be used to indicate a reversal in index orientation. It is erroneous to associate multiple dimensions of tile A with the same index in tile B. By default `orientationVector = (/1,2,3,.../)`, i.e. same orientation as tile A.

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

32 IO Capability

32.1 Description

The ESMF IO provides an unified interface for input and output of high level ESMF objects such as Fields. In the current release, the ESMF IO capability is integrated with third-party software such as Parallel IO (PIO) to read and write Fortran array data in MPI_IO binary or NetCDF format, and Xerces Library to read and write Attribute data in XML format. Other file IO functionalities, such as writing of error and log messages, input of configuration parameters from an ASCII file, and lower-level IO utilites are covered in different sections of this document. See the Log Class 43.1, the Config Class 42.1, and the Fortran I/O Utilities, 46.1 respectively.

32.2 Attribute I/O

Metadata IO is handled via the ESMF Attribute class. The third party software Xerces C++ Library is used by ESMF to provide the ability to read and write Attribute data in XML file format. To enable this capability, the environment variable ESMF_XERCES must be set. Details can be found in the ESMF User Guide, "Building and Installing the ESMF", "Third Party Libraries".
In the current release, the following methods support Attribute XML I/O using Xerces:

`ESMF_AttributeRead()`, section 35.10.23.

`ESMF_AttributeWrite()`, section 35.10.28.

32.3 Data I/O

ESMF provides interfaces for high performance, parallel I/O using ESMF data objects such as Arrays and Fields. Currently ESMF supports I/O of binary and NetCDF files. The current ESMF implementation relies on the Parallel I/O (PIO) library developed as a collaboration between NCAR and DOE laboratories. PIO is built as part of the ESMF build when the environment variable ESMF_PIO is set to "internal"; by default it is not set. When PIO is built with ESMF, the ESMF methods internally call the PIO interfaces. When PIO is not built with ESMF, the ESMF methods are non-operable (no-op) stubs that simply return with a return code of ESMF_RC_LIB_NOT_PRESENT. Details about the environment variables can be found in ESMF User Guide, "Building and Installing the ESMF", "Third Party Libraries".
In the current release, the following methods support parallel data I/O using PIO:

`ESMF_FieldBundleRead()`, section 21.5.15.

`ESMF_FieldBundleWrite()`, section 21.5.30.

`ESMF_FieldRead()`, section 22.6.46.

`ESMF_FieldWrite()`, section 22.6.48.

`ESMF_ArrayBundleRead()`, section 23.5.15.

`ESMF_ArrayBundleWrite()`, section 23.5.26.

`ESMF_ArrayRead()`, section 24.5.24.

`ESMF_ArrayWrite()`, section 24.5.37.

32.4 Data formats

Two formats are supported, namely, NetCDF and binary (through MPI_IO). The environment variables that are enabled when ESMF is built determine the format. The environment variables ESMF_NETCDF or/and ESMF_PNETCDF should be set to "standard" to enable NetCDF IO format. If neither ESMF_NETCDF nor ESMF_PNETCDF are set, and MPI_IO is enabled in MPI, the format will be binary. Details about the environment variables can be found in ESMF User Guide, "Building and Installing the ESMF", "Third Party Libraries".

NetCDF Network Common Data Form (NetCDF) is an interface for array-oriented data access. The NetCDF library provides an implementation of the interface. It also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado. See [18]. In geoscience, NetCDF can be naturally used for represenation of fields defined on logically rectangular grids. NetCDF use in geosciences is specified by CF conventions mentioned above [17].

To the extent that data on unstructured grids (or even observations) can be represented as one-dimensional arrays, NetCDF can also be used to store these data. However, it does not provide a high-level abstraction for this type of data.

IEEE Binary Streams A natural way for a machine to represent data is to use a native binary data representation. There are two choices of ordering of bytes (so-called *Big Endian* and *Little Endian*), and a lot of ambiguity in representing floating point data. The latter, however, is specified, if IEEE Floating Point Standard 754 is satisfied. ([9], [14]). [7].

32.5 Restrictions and Future Work

Currently a small fraction of the anticipated data formats is implemented by ESMF. The data IO uses NetCDF and MPI_IO binary formats, and ESMF Attribute IO uses XML format. Different libraries are employed for these different formats. In future development, a more centralized IO technique will likely be defined to provide efficient utilities with a set of standard APIs that will allow manipulation of multiple standard formats. Also, the ability to automatically detect file formats at runtime will be developed.

32.6 Design and Implementation Notes

For data IO, the ESMF IO capability relies on the PIO, NetCDF, PNetCDF and MPI_IO libraries. For Attribute IO, the ESMF IO capability uses the Xerces library to perform reading and writing of XML files. PIO is included with the ESMF distribution; the other libraries must be installed on the machine of interest.

33 Overview of Distributed Data Methods

FieldBundles, Fields, and Arrays all have versions of the following data communication methods. In these objects, data is communicated between DEs. Depending on the underlying communication mechanism, this may translate within the framework to a data copy, an MPI call, or something else. The ESMF goal of providing performance portability means the framework will in the future attempt to select the fastest communication strategy on each hardware platform transparently to the user code. (The current implementation uses MPI for communication.)
Communication patterns, meaning exactly which bytes need to be copied or sent from one PET to another to perform the requested operation, can be precomputed during an initialization phase and then later executed repeatedly. There is a common object handle, an `ESMF_RouteHandle`, which identifies these stored communication patterns. Only the `ESMF_RouteHandle` and the source and destination data pointers must be supplied at runtime to minimize execution overhead.

33.1 Higher Level Functions

The following three methods are intended to map closely to needs of applications programs. They represent higher level communications and are described in more detail in the following sections. They are:

- **Halo** Update ghost-cell or halo regions at the boundaries of a local data decomposition.
- **Regrid** Transform data from one Grid to another, performing any necessary data interpolation.
- **Redist** Copy data associated with a single Grid from one decomposition to another. No data interpolation is necessary.

33.2 Lower Level Functions

The following methods correspond closely to the lower level MPI communications primitives. They are:

- **Gather** Reassembling data which is decomposed over a set of DEs into a single block of data on one DE.
- **AllGather** Reassembling data which is decomposed over a set of DEs into multiple copies of a single block of data, one copy per original DE.
- **Scatter** Spreading an undecomposed block of data on one DE over a set of DEs, decomposing that single block into smaller subsets of data, one data decomposition per DE.
- **AlltoAll** Spreading an undecomposed block of data from multiple DEs onto each of the other DEs in the set, resulting in a set of multiple decomposed data blocks per DE, one from each of the original source DEs.
- **Broadcast** Spreading an undecomposed block of data from one DE onto all other DEs, where the resulting data is still undecomposed and simply copied to all other DEs.
- **Reduction** Computing a single data value, e.g. the data maximum, minimum, sum, etc from a group of decomposed data blocks across a set of DEs, where the result is delivered to a single DE.
- **AllReduce** Computing a single data value, e.g. the data maximum, minimum, sum, etc from a group of decomposed data blocks across a set of DEs, where the result is delivered to all DEs in the set.

33.3 Common Options

ESMF will select an appropriate default for the internal communication strategy for executing the communications. However, additional control is available to the user by specifying the following route options. (For more details on exactly what changes with the various options, see Section 33.4.)

33.4 Design and Implementation Notes

1. There is an internal `ESMC_Route` class which supports the distributed communication methods. There are 4 additional internal-only classes which support `ESMC_Route`: `ESMC_AxisIndex`, `ESMC_XPacket`, `ESMC_CommTable`, and `ESMC_RTable`; and a public `ESMF_RouteHandle` class which is what the user sets and gets. The implementation is in C++, with interfaces in Fortran 90.

   The general communication strategy is that each DE computes its own communication information independently, in parallel, and adds entries to a per-PET route table which contains all needed sends and receives (or gets and puts) stored in terms relative to itself. (Implementation note: this code will need to be made thread-safe if multiple threads are trying to add information to the same route table.)

   AxisIndex is a small helper class which contains an index minimum and maximum for each dimension and is used to describe an n-dimensional hypercube of information in index space. These are associated with logically rectangular grids and local data arrays. There are usually multiple instances of them, for example the local data chunk, and the overall global index-space grid this data is a subset of. Within each of the local or global categories, there are also multiple instances to describe the allocated space, the total area, the computational area, and the exclusive area. (Implementation note: the allocated space is only partially implemented internally and has no external user API yet.)

   An Exchange Packet (XPacket) describes groups of memory addresses which constitute an n-dimensional hypercube of data. Each XPacket has an offset from a base address, a contiguous run length, a stride (or number of items to skip) per dimension, and a repeat count per dimension. See Figure 21 for a diagram of how the XPacket describes memory. The actual unit size stored in an XPacket is an item count, so before using an XPacket to address bytes of memory the item size must be known and the counts multiplied by the number of bytes per item. This allows the same XPacket to describe different data types which have the same memory layout, for example 4 byte integers and 8 byte reals/doubles. The XPacket methods include basic set/get, how to turn a list of AxisIndex objects into an XPacket, compute a local XPacket from one in global (undecomposed grid) space, and a method to compute the intersection of 2 XPackets and produce a 3rd XPacket describing that region.

   The Communication Table (CommTable) class encapsulates which other PETs this PET needs to talk to, and in what order. There are create and destroy methods, methods to set that a PET has data either to send or receive, and query routines that return an answer to the question 'which PET should I exchange data with next'.

   The Route Table (RTable) class contains a list of XPackets to be sent and received from other PETs. It has create/destroy methods, methods to add XPackets to the list for each PET, and methods to retrieve the XPackets from any list.

   The top level class is a Route. A Route object contains a send RTable, a recv RTable, a CommTable, and a pointer to a Virtual Machine. The VM must include all PETs which are participating in this communication. The Route methods include create/destroy, setting a send or recv XPacket for a particular PET, and some higher level functions specific to each type of communication, for example RoutePrecomputeHalo or RoutePrecomputeRedist. These latter functions are where the XPackets are actually computed and added to the Route table. Each DE computes its own set of intersections, either source or destination, and fills its own corresponding PET table. The Route methods also include a RouteRun method which executes the code which actually traverses the table and sends the information between PETs.

   A RouteHandle class is a small helper class which is returned through the public API to the user when a Route is created, and passed back in through the API to select which precomputed Route is to be executed. A RouteHandle contains a handle type and a pointer to a Route object. In addition, for use only by the Regrid code, there is an additional Route pointer and a TransformValues pointer. (TransformValues is an internal class only used by the Regridding code.) If the RouteHandle describes the Route for a FieldBundle, then the RouteHandle can contain a list of Routes, one for each Field in the FieldBundle, and for Regrid use, a list of additional Routes instead of a single Route. There is also a flag to indicate whether a single Route is applicable to all Fields in a FieldBundle or whether there are multiple Routes. The RouteHandle methods are fairly basic; mostly accessor methods for getting and setting values.

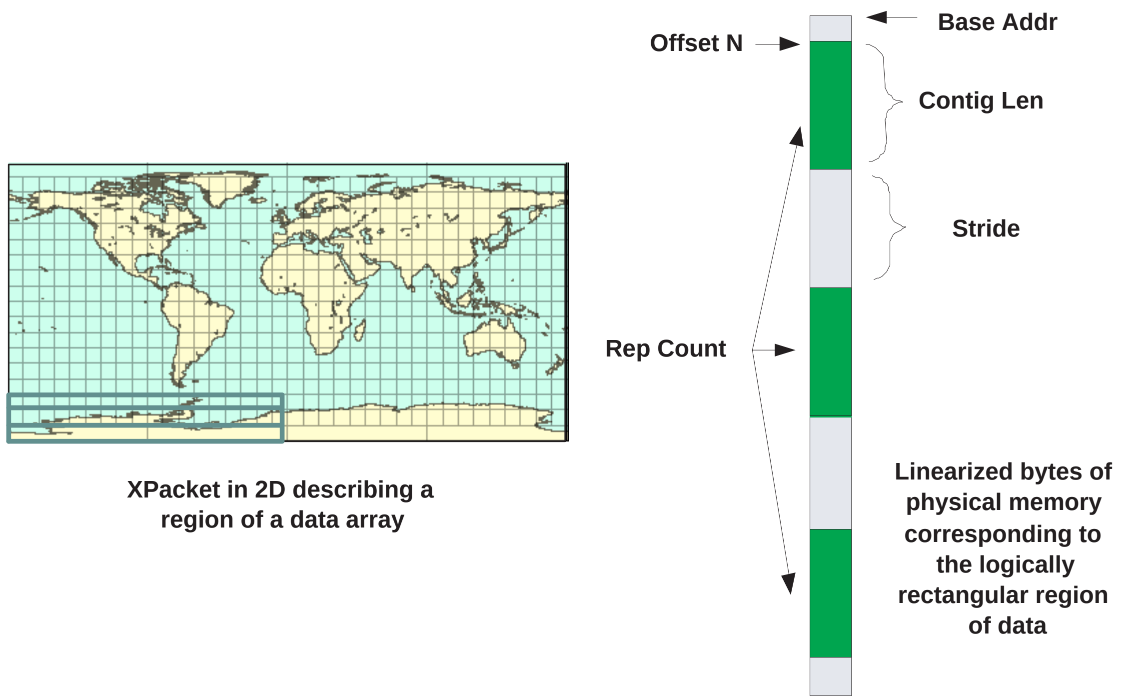


Figure 21: How an Exchange Packet (XPacket) describes the memory layout for a rectangular hypercube of data.

2. While intended for any distributed data communication method, the current implementation only builds a Route object for the halo, redist, and regrid methods. Scatter, Gather, AllGather, and AlltoAll should have the option of building a Route for operations which are executed repeatedly. This should only require writing a Precompute method for each one; the existing RouteRun can be invoked for these operations. (This is a lack-of-implementation-time issue, not a design or architecture issue.)

3. The original design included automatic detection of different Routes and internal caching, so the user API did not have to include a RouteHandle object to identify which Route was being invoked. However, users requested that the framework not cache and that explicit RouteHandle arguments be created and required to invoke the distributed data methods. Nothing prevents this code from being revived from the CVS repository and reinstated in the system, should automatic caching be desired by future users.

4. The current distributed methods have 2 related but distinct interfaces which differ in what information they require and whether they use RouteHandles:

Precompute/Run/Release
: This is the most frequently used interface set. It contains 3 distinct phases: precomputing which bytes must be moved, actually executing the communications operation, and releasing the stored information. This is intended for any communication pattern which will be executed more than once.

All-in-One
: For a communication which will only be executed once, or in any situation in which the user does not want to save a RouteHandle, there are interfaces which do not have RouteHandles as part of the argument list. Internally the code computes a Route, executes it, and releases the resources before returning.

5. The current CommTable code executes one very specific communication strategy based on input from a user who did extensive timing measurements on several different hardware platforms. Rather than broadcasting all data at once asychronously, it selects combinations of pairs of processors and has them execute a SendRecv operation, which does both a data send and a data receive in a single call. At each step in the execution, different pairs of processors exchange data until all pair combinations have been selected.

   The table itself must be a power of 2 in size; the number of PETs is rounded up to the next power of 2 and then all entries for PETs larger than the actual number are marked as no-ops.

   There are many alternative execution strategies, including a completely asynchronous execution, in numeric PET order, without computing processor pairs. Also single-direction communications are possible (only the Send XPackets are processed, or only the Receive XPackets) in either a synchronous or asynchronous mode. This would not require any changes to the XPacket or RTable classes, but would require writing a set of alternative RouteRun methods.

6. The current RouteRun routine has many possible performance options for how to make the tradeoff between time spent packing disjoint memory blocks into a single buffer to minimize the number of sends, verses simply sending the contiguous blocks without the pack overhead. The tradeoffs are not expected to be the same on all systems; hardware latency verses bandwith characteristics will differ, plus the underlying communication software (MPI, shared memory, etc) will change the performance. Also the size of the data blocks to be sent, the amount of contiguity, and limits on the number of outstanding communication buffers all affect what options are best.

   The `ESMF_RouteOptions` are listed in 33.3; the following description contains more implementation detail about what each of the options controls inside the execution of a Route. Note that the options do not affect the creation of a Route, nor any of the Precompute code, and can optionally be changed each time the Route is run.

   Packing options:

By Buffer
: If multiple memory addresses are provided to RouteRun (from bundle-level communications, for example), then this option packs data across all buffers/blocks as specified by the other packing flags before sending or receiving. Note: unlike the other packing flags, this is handled in the code at a higher level by either passing down multiple addresses into the route run routine or not. If multiple addresses are passed into the run routine, they will be packed. The "no-packing" option at this level would be identical to looping at the outermost level in the RouteRun code and therefore there is no disadvantage to calling this routine once per address (and the advantage is not adding yet another coding loop inside the already complex RouteRun code). The higher level list-of-address code can be disabled by clearing this flag (which is on by default).

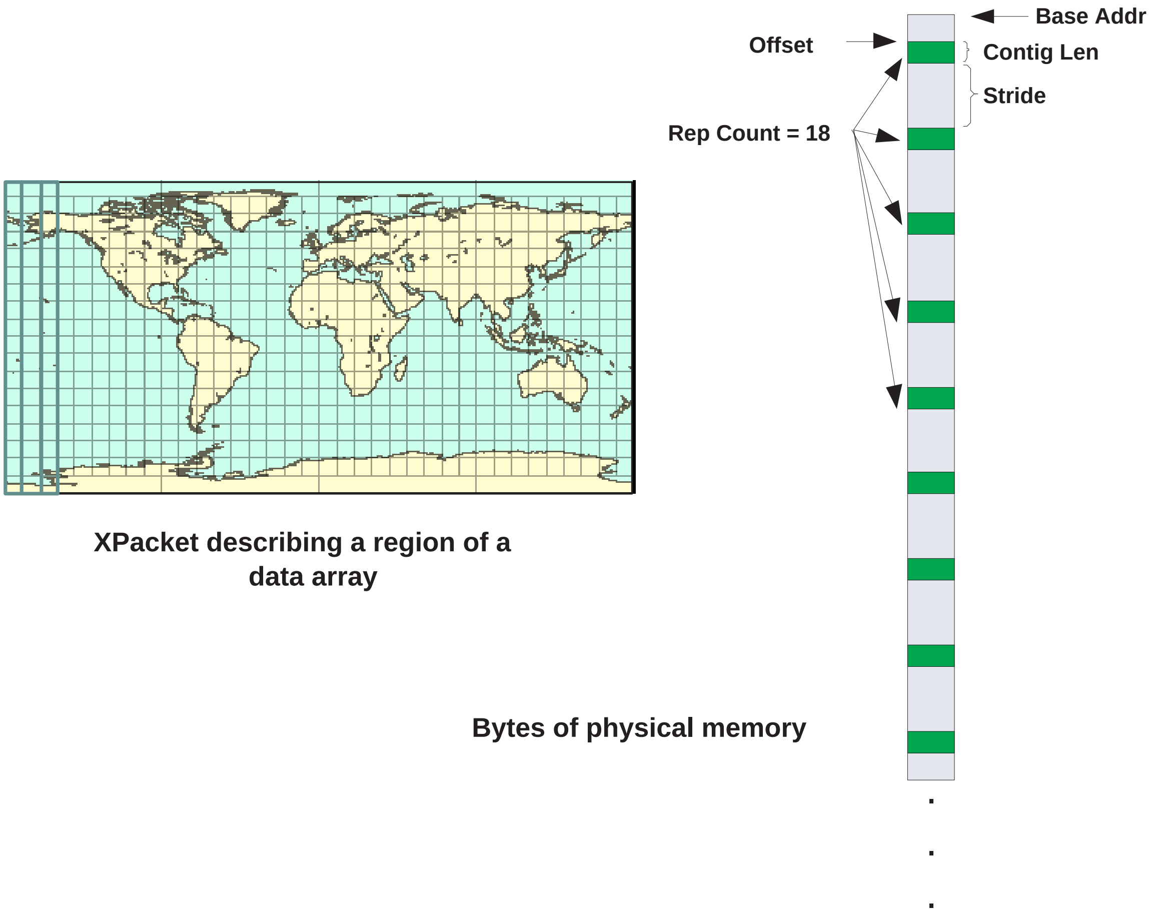


Figure 22: A common XPacket pattern which generally benefits from packing; the overlap region between 2 DEs during a halo update are often short in the contiguous dimension and have a high repeat count.

By PET All data from a single block intended for a remote PET is packed into a single send buffer, and sent in a single VM communications call. A buffer large enough to receive all data coming from that remote PET is allocated, the data is received, and then the data is copied into the final location. See 26.

By XP All data described by a single XPacket (which is a n-dimensional hyperslab of memory) is packed into a single buffer for sending, and a single buffer large enough to receive an XPacket is allocated for receiving the data. See 25.

No Packing A VM communication call is made for each single contiguous strip of memory, regardless of how long or short.

MPI Vector MPI implements a set of interfaces for sending and receiving which allows certain strided memory patterns to be sent in a single call. The actual implementation is up to the MPI library itself. But no user-level data copy is needed in this case. (Not implemented yet.)

Note that in all packing options, if the XPacket describes a chunk of memory which is completely contiguous, then the code does not allocate a packing or unpacking buffer but supplies the actual data address to the communications call so the data is read or written in place.

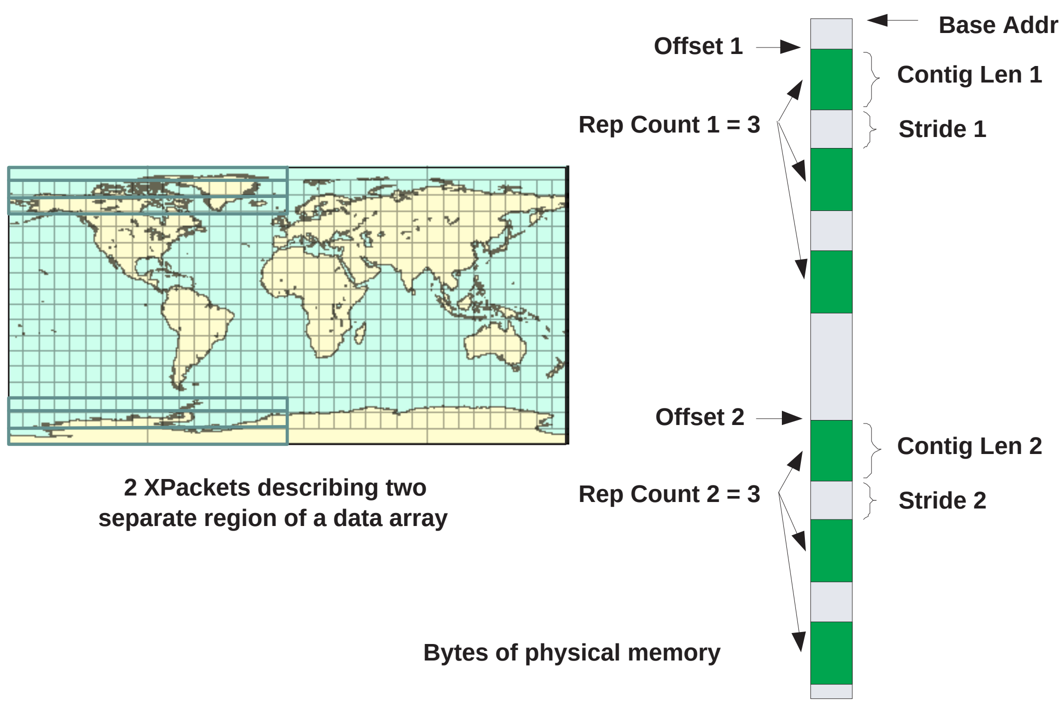
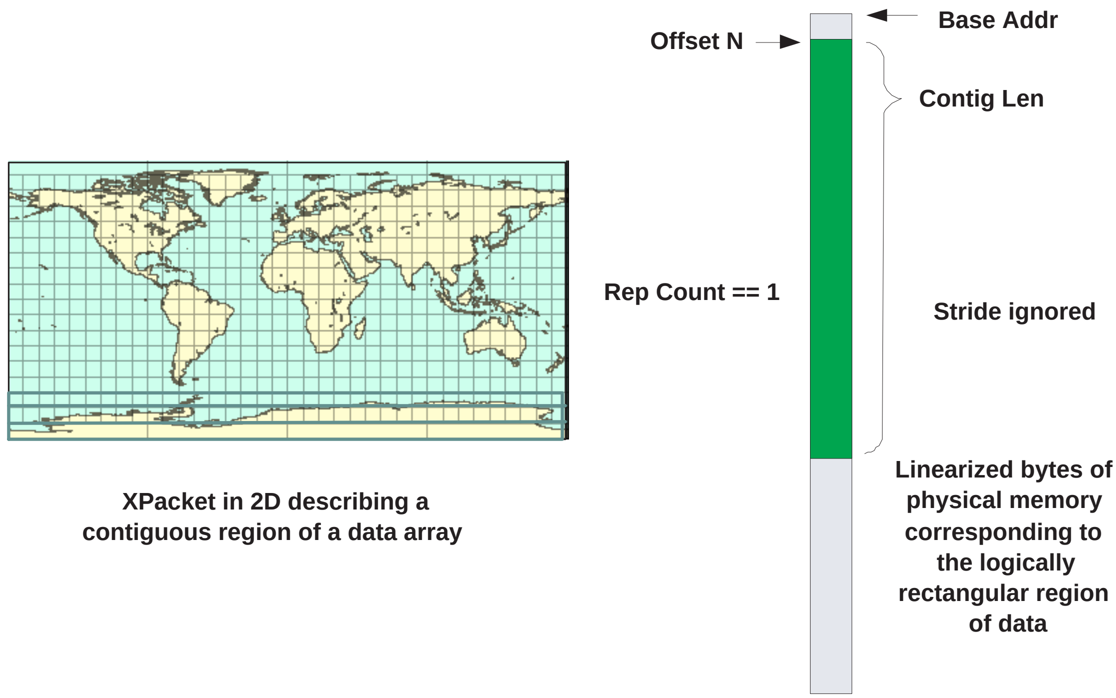


Figure 23: When there are multiple XPackets destined for the same remote PET there are more options for how to order the contiguous pieces into a packed buffer.

Figure 24: When the XPacket describes memory which is physically a single contiguous region, there is no need to copy the data into another buffer; it can be communicated inplace. There is a flag in the XPacket which marks how many of the dimensions are contiguous.



The following options refer to the internal strategy for executing the route and not to whether the user-level API call returns before the route has finished executing. The current system only implements user-synchronous calls; asynchronous calls are on the to-be-written list.

Sync Each pair of processors exchanges data with the VM equivalent of an MPI_SendRecv() call, which does not return until both the send and receive have completed.

Async Each processor executes both an asynchronous send and asychronous receive to the other processor and does not wait for completion before moving on to the next communication in the CommTable. Then in a separate loop through the RTables, each call is waited for in turn and when all outstanding communication calls have completed, then the API call returns to the user.

(Note that in the Async case it makes much more sense to iterate throught the Route table in PET order instead of the complication of computing communication pairs and iterating in a non-sequential order. The code is as it is now for reasons of implementation speed and not for any other design reason. This would require a slightly simpler, but separate, version of the RouteRun() subroutine.)

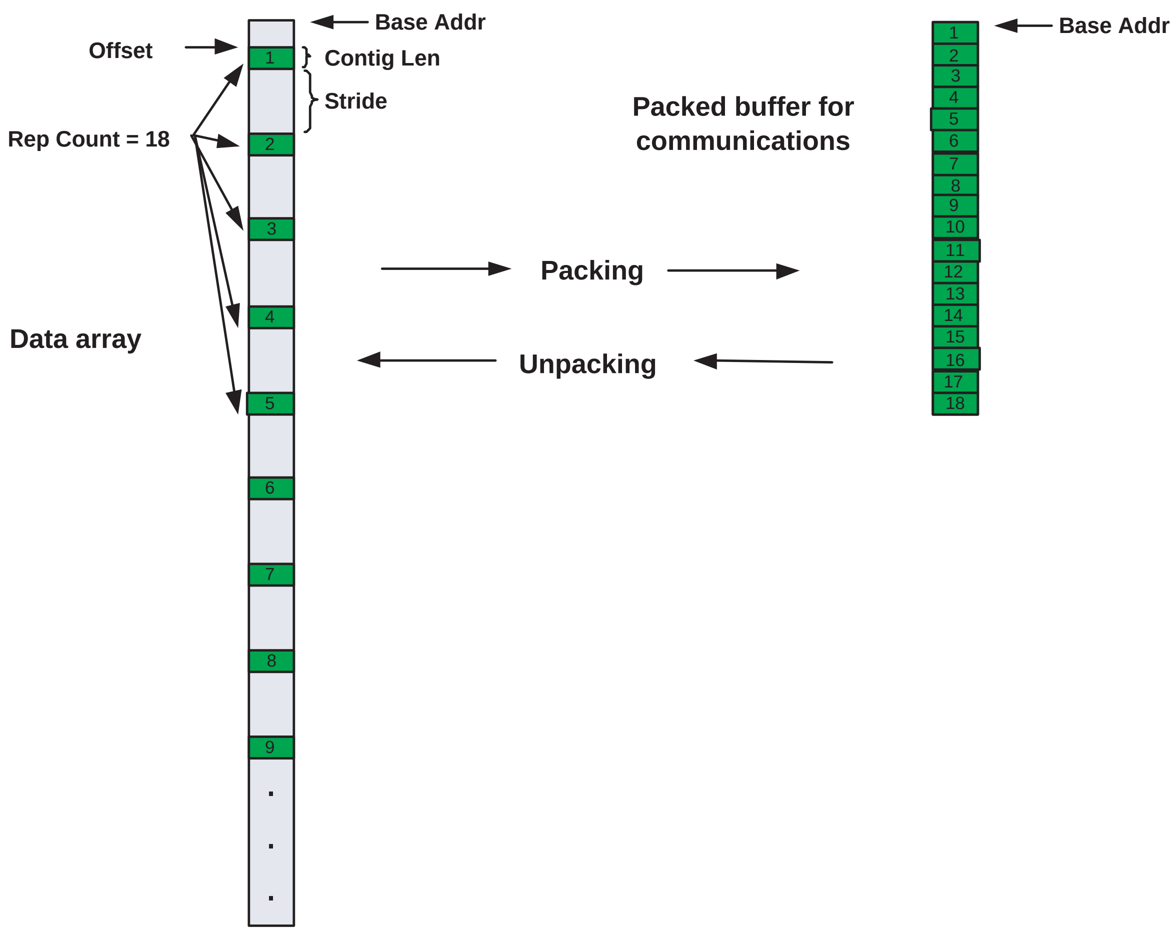


Figure 25: Often the overhead of making multiple communication calls outweighs the cost of copying non-contiguous data into a contiguous buffer, sending it in a single operation, and then copying it to the final memory locations on the receiving side.

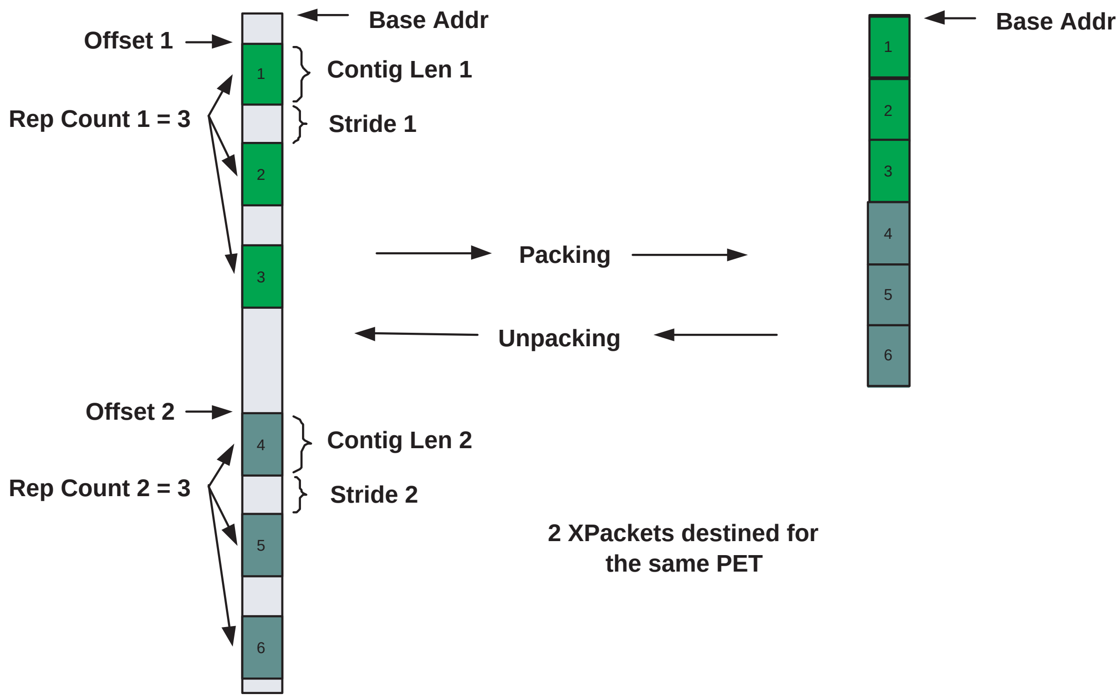


Figure 26: Once there is more than a single XPacket to pack, there are many more interleave options. For example, packing in the order: 1, 4, 2, 5, 3, 6 would also be possible here. However the code becomes more complicated when the XPackets have different repeat counts, and has no real performance advantage over the straightforward packing of each XPacket in sequence. Note that this packing is the same whether it refers to multiple XPackets from the same memory buffer or from multiple buffers.

7. FieldBundle-level communication calls have additional packing options under certain circumstances. FieldBundles are groups of Fields which share the same Grid, but they are not required to share the same data types, data ranks, nor relative data locations. FieldBundles in which these things are the same in all Fields are marked inside the bundle code as being **congruent**. At communication store time FieldBundles which have congruent data in all the Fields have the option of packing all Field data together into fewer communication calls which generally is expected to give better performance. Fields where the data is not of the same type or perhaps not the same number of items (e.g. different rank, vertex-centered data vs. cell centered data) can in theory also be packed but in fact the code becomes more complicated, and in the case of differing data types may cause system errors because of accessing data on non-standard byte offsets or putting mixing integer data with floating data and causing NaN (not a number) exceptions. In this case, the conservative implementation strategy is to construct a separate Route object for each Field, all enclosed in the same RouteHandle. Inside the FieldBundle communication code the execution for both types of FieldBundles is identical for the caller, but inside the congruent FieldBundle code calls the `ESMF_RouteRun()` code once and all communication for all Fields in the FieldBundle is done when it returns. The non-congruent FieldBundles execute a separate `ESMF_RouteRun()` call for each Field and return to the user when all Field data have been sent/received.

   There are comments in the code for an intermediate level of optimization in which the FieldBundle code determines the smallest number of unique types of Fields in the FieldBundle, and all same types share the same Route object, but this has not been implemented at this time. Once the existing code has been in use for a while, whether this is useful or needed may become more clear.

8. The precompute code for all operations must have enough information to compute which parts of the data arrays are expected to be sent to remote PETs and also what remote data is expected to be received by this PET.

   These computations depend heavily on what type of distributed method is being executed. The regridding methods are described in detail separately in the Regrid Design and Implementation Notes section. The halo and redistribution operations are described here.

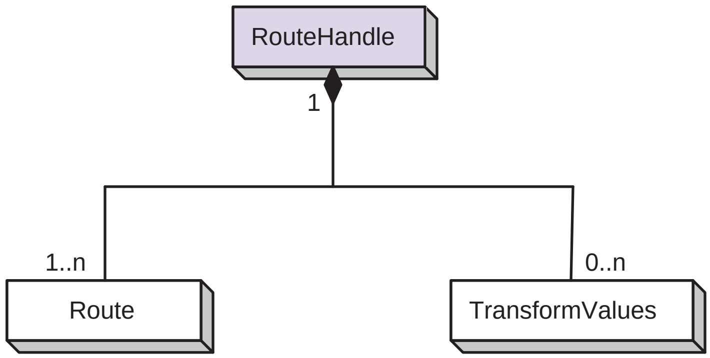
Halo The total array area, which includes any halo regions, are intersected with the computational area of other DEs. The overlap regions are converted from index space into memory space and stored as XPackets in the RTables. This code must be aware of: whether the grid was defined as periodic in any or all of the dimensions since that affects which halo regions overlap at the grid edges; if the data is only decomposed into a single block in any dimension (which means it halos with itself); and if the halo region is large enough that a halo operation may require intersection with the N+1 neighbor in any dimension.

Redistribute Each DE computes the overlap between its own computational region and all DEs in the remote Grid, again only working in computational area. The overlap regions are converted from index space into memory space and stored as XPackets in the RTables. After execution a redistribution, a halo operation may be required to populate any halo regions with consistent data.

(Note: the Redistribution code has been reimplemented to intersect the DEs in index space and then convert the overlap region to an XPacket representation. Halo still converts the regions from AxisIndex to XPackets and then intersects the XPackets, but this code needs to be changed to intersect in AxisIndex space and once the overlap is computed then convert to XPackets. Intersecting AxisIndex objects is very much simpler, both to understand and to execute, and more easily extensible to multiple dimensions than intersecting XPackets.)

33.5 Object Model

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


RouteHandle
1
1..n
Route
0..n
TransformValues

Part V

Infrastructure: Utilities

34 Overview of Infrastructure Utility Classes

The ESMF utilities are a set of tools for quickly assembling modeling applications.

The ESMF Attribute class enables models to be self-describing via metadata, which are instances of Attribute name-value pairs.

The Time Management Library provides utilities for time and time interval representation and calculation, and higher-level utilities that control model time stepping, via clocks, as well as alarming.

The ESMF Config class provides configuration management based on NASA DAO's Inpak package, a collection of methods for accessing files containing input parameters stored in an ASCII format.

The ESMF LogErr class consists of a variety of methods for writing error, warning, and informational messages to log files. A default Log is created during ESMF initialization. Other Logs can be created later in the code by the user.

The DELayout class provides a layer of abstraction on top of the Virtual Machine (VM) layer. DELayout does this by introducing DEs (Decomposition Elements) as logical resource units. The DELayout object keeps track of the relationship between its DEs and the resources of the associated VM object. A DELayout can be shaped by the user at creation time to best match the computational problem or other design criteria.

The ESMF VM (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component. In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods.

The ESMF Fortran I/O utilities provide portable methods to access capabilities which are often implemented in different ways amongst different environments. Currently, two utility methods are implemented: one to find an unopened unit number, and one to flush an I/O buffer.

35 Attribute Class

35.1 Description

The ESMF Attribute class is a metadata utility that supports emerging standards in a flexible way. The Attribute class is useful for documenting data provenance and encourages models to be more self describing. Attributes can also be used to automate some aspects of model execution and coupling.
Metadata, which is data about data, is broken down into name-value pairs by the Attribute class. Attributes can be attached at any level of the ESMF object hierarchy, and in some cases the Attributes of different ESMF objects can be linked together to form a corresponding Attribute hierarchy. Attribute hierarchies are linked up automatically for the most part, with the exception of links between Components and between a Component and a State. Attribute hierarchies can also be unlinked, copied, and moved around as needed.
ESMF Attribute packages are used to aggregate, store, and output model metadata. They can be nested inside each other to make larger organized packages, distributed across processors and updated at runtime, and expanded to suit specific needs. The ESMF-supplied Attribute packages are designed around accepted metadata conventions, such as: climate and forecast (CF) [6], ISO standards [3], and the METAFOR Common Information Model (CIM) [4].
Most of the ESMF deep objects can host Attributes, and every object that can hold individual Attributes can also hold Attribute packages. Attribute hierarchies are supported for a majority of the Attribute bearing classes. More information on the various Attribute capabilities, and the classes for which they are supported appear in the following sections.
Reading and writing Attribute XML files requires the Xerces C++ library, v3.1.0 or better. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, Xerces".

35.1.1 The ESMF approach to Attributes

ESMF's approach to Attributes can be summarized as follows:

- Implement community standards where they exist.
- Associate Attributes with the ESMF object they describe. Currently, the following ESMF objects can have Attributes:
  - CplComp
  - GridComp
  - State
  - FieldBundle
  - Field
  - ArrayBundle
  - Array
  - Grid
  - DistGrid
- Establish pre-defined Attribute packages (see Section 35.2) to make Attribute creation easier for the user.
- Allow for user-defined custom Attribute packages (see Section 35.2.7).
- Enable the nesting of Attribute packages (see Section 35.3) including Custom packages.
- Enable complex Attribute hierarchies (see Section 35.1.2.
- Export Attributes in more than one format (see Section 35.4).
- Ensure that all Attributes are consistent across the entire virtual machine of the object to which they are attached.

35.1.2 Attribute hierarchies

Of the ESMF objects with Attributes, only some can link their Attributes together in an Attribute hierarchy. These objects are:

- CplComp
- GridComp
- State
- FieldBundle
- Field
- ArrayBundle
- Array

Every ESMF deep object is given a `root` Attribute on creation. These `root` Attributes serve as the attachment point for all metadata that is stored on a particular ESMF object, including all Attributes and Attribute packages. The `root` Attributes can also be connected together via the `ESMF_AttributeLink()` functionality. This happens automatically in most cases, such as when a Field is added to a FieldBundle, and results in the formation of an Attribute hierarchy which mirrors the structure of the underlying object hierarchy.
When two Attribute hierarchies are linked together the objects are given read-only access to each other's Attributes. To ensure consistency across a distributed system, there can only ever be one set of Attributes associated with each ESMF object. This implies that a copy operation on an ESMF object Attribute hierarchy *can* use a value copy for all Attributes which are owned by the object being copied, but *must* use a reference copy for all Attributes which the object can access (through links) but does NOT own. See section 35.8.6 for more details on this concept.
The most common use for this hierarchy capability is for linking the Attributes of a Field to the FieldBundle which holds it, which is then linked to the State that is used to transport all of the data for a Component. All of these links, with the exception of the link between the Component and the State, are automatically handled by ESMF. Additionally, the State will automatically set a `VariableIntent` Attribute for Field when that Field is added to the State. `VariableIntent` will be set to either `Export` or `Import`.

35.2 Attribute Packages

At this time, all ESMF objects which are enabled to contain Attributes can also contain Attribute packages, which are groupings of individual Attributes. Every Attribute package is specified by a **convention** and a **purpose**, hereafter called **specifiers**, such as "CF" and "General" (see below). These specifiers are used to validate ESMF Attribute packages against existing metadata conventions. The user can choose to use an ESMF pre-defined Attribute package, specify their own Attribute package, or add customized Attributes to any of the ESMF pre-defined Attribute packages. Currently, the creation and setting of Attribute packages is quite involved, but future development with IO will allow for a more automated approach to populating Attribute packages from a file. This is already possible via `ESMF_AttributeRead()` for the ESMF/ESG/CF Attribute packages supplied by ESMF, as well as for custom individual Attributes not in a package.
The standard Attribute packages supplied by ESMF exist for the following ESMF objects:

- CplComp
- GridComp
- State
- Field
- Array
- Grid

The packages described in this section are grouped by the ESMF object they apply to. The creation of custom attributes and custom attribute packages is also possible and is discussed in Section 35.2.7. In some cases it is possible to nest custom packages on top of ESMF packages. Attribute package nesting is described separately in Section 35.3.
Some Attributes come with a controlled vocabulary. A controlled vocabulary is a list of options that can be selected as the value of the attribute. The controlled vocabularies listed in this documentation represent those chosen by the community. They are not exhaustive and users may set these Attributes to a different value if they so choose. The primary consequence of doing so is that the resulting output may not be recognized by any of the online tools being developed with respect to this controlled vocabulary.

35.2.1 Component Attribute packages

There are many attributes that are used to describe components. There are currently 3 predefined component-level Attribute packages, with sub-packages defined for the 2nd:

1. Earth System Grid (ESG) Basic
2. Common Information Model (CIM) Main
   (a) Common Information Model (CIM) Platform
   (b) International Organization for Standardization (ISO) Responsible Party
   (c) International Organization for Standardization (ISO) Citation
3. Common Information Model (CIM) Component Properties

1. Earth System Grid (ESG) Basic Attribute Package

- Convention: ESG or ESMF
- Purpose: General
- Output Options:
  - Simple XML file. This requires the Xerces 3rd party library, see 35.8.7
- Description: This package contains several Attributes used to describe model components within the Earth System Grid (ESG) ontology.

Name	Definition	Controlled Vocabulary
Agency	An administrative unit of government.	DoD, DOE, DOI, NASA, NOAA, NSF
Author	The person who created the content of a book, article, or other source.	N/A
CodingLanguage	The computer language in which a unit of software is written.	C, C++, F77, F90, Java
ComponentLongName	The name of a model, model component, simulation, experiment, or dataset with all acronyms spelled out.	N/A
ComponentShortName	A version of the component name that contains acronyms.	N/A
Discipline	A subject, theme, category, or general area of interest.	Aerosol, Fisheries, Climate, Carbon Cycle, Hydrology, Land, Ocean, Polar, Sediment, Storm Surge, Turbulence, Weather, Wave, Weather Prediction
Institution	An organization associated with a model component, simulation, or dataset.	N/A
ModelComponentFramework	The software package or mechanism used to transfer and transform data between model components.	CCA, ESMF, Flume, FMS, OASIS, SWMF
PhysicalDomain	A description of the geographic range being simulated.	Atmosphere, Earth System, Ice, Lake, Land Ocean, River
Version	A specific form or variation of an artifact, i.e. a unit of software or metadata.	N/A

2. Common Information Model (CIM) Main Attribute Package

- Convention: CIM 1.5
- Purpose: Model Component Simulation Description
- Includes:
  - CIM Platform
  - ISO Responsible Party (1 or more – user specifiable)
  - ISO Citation (1 or more – user specifiable)
- Output Options:
  - CIM XML
- Description: The CIM is a formal model of the climate modeling process being developed by the European Union's METAFOR project. "It includes descriptions of the experiments being undertaken, the simulations being run in support of these experiments, the software models and tools being used to implement the simulations and the data generated by the software." The CIM divides up the climate modeling process into 6 sections, listed below. The CIM also contains other standards. It is the primary metadata representation for the fifth Climate

Model Intercomparison Project (CMIP5). ESMF is currently implementing only a subset of the CIM. The representation is expected to grow. The CIM Main Package contains several standalone properties used to describe components. It also serves as the anchor to which other CIM packages are nested. Presently, these additional CIM packages (described further below) can only be created if the CIM Main Package is created. In the future, these packages will be decoupled, so that users may select subsections of the CIM to create and use. This package nests three of the packages below within it; this is described in Section 35.3.

- Shared: Contains those elements that are used in many different packages.
- Quality: Contains elements used to express diverse quality metrics for CIM Metadata or for artifacts that the CIM metadata describes.
- Grids: Provides a complete description of the horizontal and vertical discretization of modeling elements. This may refer to grids that output data is mapped onto, software adheres to, as well as activity constraints.
- Activity: Specifies the experimental design including the experimental requirements and description of how simulations conform to these requirements.
- Software: Specifies all the modeling software components used within the experiment process.
- Data: Describes the data output from the climate modeling process as well as for input data.

Name	Definition	Controlled Vocabulary
`Description`	A multi-line description of the component.	N/A
`LongName`	A version of the component name with all acronyms spelled out.	N/A
`MetadataVersion***`	The version number of the simulation metadata.	N/A
`ModelType*`	A short string describing the discipline of a model component.	Advection, Aerosol3D-Sources etc.
`PreviousVersion**`	Name of the previous version of a model or model component.	N/A
`PreviousVersionDescription**`	A short note about the previous version of the model or model component.	N/A
`ReleaseDate`	The date a model component was issued.	N/A
`ShortName*`	A version of the component name that contains acronyms.	N/A
`SimulationDuration`	The length of time a simulation runs.	N/A
`SimulationEndDate`	The date in simulated time of the end of a model simulation.	N/A
`SimulationEnsembleID`	The reference name or number of the ensemble to which a simulation belongs.	N/A
`SimulationLongName`	The name of the simulation with any acronyms spelled out.	N/A
`SimulationNumberOfProcessingElements`	The number of PEs used in the simulation.	N/A
`SimulationProjectName`	A campaign, such as a model intercomparison project, that may involve multiple groups and experiments.	N/A
`SimulationRationale`	The reason for performing a simulation.	N/A
`SimulationShortName`	The name of the simulation.	N/A
`SimulationStartDate*`	The date in simulated time of the start of a model simulation.	N/A
`URL`	A URL associated with a model component.	N/A
`Version`	Version number of the component.	N/A

* Attribute required to be set to produce valid CIM XML output.
* If PreviousVersionDescription is set, PreviousVersion must also be set, to produce valid CIM XML output.
** If not set, defaults to 1.0

2.1. CIM Platform Attribute Package

- Convention: CIM 1.5

- Purpose: Platform Description
- Output Options:
  - CIM XML
- Description: This package describes the platform a particular simulation is run on. It must be created in conjunction with the CIM Main Package (see above). This package is nested within the CIM Main Package (above); see the description in Section 35.3.

Name	Definition	Controlled Vocabulary
`CompilerName**`	The brand of the software that takes source code and turns it into an executable.	Absoft, Default, Intel, Lahey, NAG, Pathscale, PGI, PGIGCC, XLF, XLFGCC
`CompilerVersion**`	The specific configuration value of the software used to take source code and turn it into executable code.	N/A
`MachineCoresPerProcessor`	The number of sub-divided elements or mini-chips on a computer chip.	N/A
`MachineDescription`	A short note about the machine.	N/A
`MachineInterconnectType`	The technology used to associate each node in a supercomputer with every other node.	Cray Interconnect, Fat Tree, Gigabit Ethernet, Infiniband, Mixed, Myrinet, Numalink, Quadrics, SP Switch
`MachineMaximumProcessors`	The highest number of computer chips on a computer system.	N/A
`MachineName*`	The name given to a computer by its system administrators. This is not the brand name of the system.	N/A
`MachineOperatingSystem`	The software that is responsible for the management and coordination of activities and the sharing of resources of a computer.	Aix, Darwin, Irix64, Linux, SUNOS, Unicos
`MachineProcessorType`	The type of computer chip used in a particular computer platform.	Altix, AMD x86-64, Bluegene, G4, G5, Intel EM64T, Intel IA-64, Itanium, NEC, Opteron, Origin3800, Pentium 3, Pentium 4, SP, SPARC, X1, Xeon, XT3-4, ZX6000
`MachineSystem`	The type of computer system (e.g. vector, parallel, cluster, etc.).	Beowulf, Parallel, Vector
`MachineVendor`	The brand name of a computer system.	ACS, Action, Appro International, Bull SA, Cray Inc, Dalco AG Switzerland, Dawning, Dell, Fujitsu, Hitachi, HP, IMB, Intel, Koi Computers, Lenovo, Mac, NEC, NEC SUN, NUDT, PC, Pyramid Computer, Raytheon-Aspen Systems, Self Made, SGI, Sun Microsystems, T-platorms

* Attribute required to be set to produce valid CIM XML output.

* Both CompilerName and CompilerVersion are required to be set, or else neither one, to produce valid CIM XML output; setting one without the other will produce invalid CIM XML output.

2.2. ISO Responsible Party Attribute Package

- Convention: ISO 19115
- Purpose: Responsible Party Description
- Output Options:
  - CIM XML
- Description: This package is used to describe contacts, authors, institutions, and funding agencies. This package is nested, with one or more user-specifiable instances, within the CIM Main Package(above); see the description in Section 35.3.
- Usage: The Responsible Party package is unique in that the user should first select the type of Responsible Party they wish to define. This is done via the ResponsiblePartyRole attribute within the package. Then the package's main value is set using the Name attribute.

Name	Definition	Controlled Vocabulary
`Abbreviation`	The abbreviation of an individual or organization associated with a model component or simulation.	N/A
`EmailAddress`	The email address that others can use to ask questions about a model component.	N/A
`Name`	The name of an author, contact, funder, centre, or principal investigator.	N/A
`NameType`	The type of entity that Name references.	Individual, Organization, Position
`PhysicalAddress`	The address of the person designated to provide information about a model component.	N/A
`ResponsiblePartyRole*`	A flag to define the role of the Responsible Party.	Author, PI, Contact, Center, Funder
`URL`	A URL of an individual or organization.	N/A

* Attribute required to be set, when any other attributes in this package are set, to produce valid CIM XML output. It is valid to set none of the attributes in this package. In that case, no corresponding CIM XML output will appear for that Responsible Party package instance, although there may be other populated instances, which, because they have attributes set, will appear in the output.

2.3. ISO Citation Attribute Package

- Convention: ISO 19115
- Purpose: Citation Description
- Output Options:
  - CIM XML
- Description: This package is used to describe references. Examples include a URL or a scientific reference. This package is nested, with one or more user-specifiable instances, within the CIM Main Package (above); see the description in Section 35.3.

Name	Definition	Controlled Vocabulary
`Date*`	The date of the citation.	N/A
`DOI`	The assigned Digital Object Identifier (DOI) of the citation.	N/A
`LongTitle`	The text of the citation or pointer (e.g. URL) that further describes a model component or simulation.	N/A
`PresentationForm`	A description of the type of citation.	documentDigital, documentHardcopy, imageDigital, imageHardcopy, mapDigital, mapHardcopy, modelDigital, modelHardcopy, profileDigital, profileHardcopy, tableDigital, tableHardcopy, videoDigital, videoHardcopy
`ShortTitle*`	An abbreviation for the citation. This could be the short scientific citation (e.g. Murphy, 2009) or the title of a web page.	N/A
`URL`	Website associated with the citation.	N/A

* Attribute required to be set, when any other attributes in this package are set, to produce valid CIM XML output. It is valid to set none of the attributes in this package. In that case, no corresponding CIM XML output will appear for that Citation package instance, although there may be other populated instances, which, because they have attributes set, will appear in the output.

3. Common Information Model (CIM) Component Properties Package

- Convention: CIM 1.5
- Purpose: General Component Properties Description
- Output Options:
  - CIM XML
- Description: This package is used to specify any number of user-defined attributes of a component and have them output in valid CIM XML format. This differs from a custom attribute package (see Section 35.2.7) in that this package has a standard convention and purpose, which is used to control the output of the user-defined attributes in standard CIM XML format.

Name	Definition	Controlled Vocabulary
`<User-defined name>`	<User-defined definition>.	N/A
`...`	...	N/A

35.2.2 State Attribute packages

There is currently only 1 predefined State-level Attribute package:

1. ESMF Basic

1. ESMF Basic State Attribute Package

- Convention: ESMF
- Purpose: General
- Output Options:
  - Tab-delimited
  - Simple XML
- Description: This package is used to define whether an ESMF State object is an Import State or Export State.

Name	Definition	Controlled Vocabulary
`Intent`	An indication of whether a state is imported into or exported from a particular model component. This refers to coupling, and not history output.	Export,Import

35.2.3 Field Attribute packages

Several standards exist to describe fields. There are currently 4 predefined Field-level Attribute packages:

1. Common Information Model (CIM) Inputs
2. Earth System Grid General
3. Climate Forecast (CF) Convention Extended
4. Climate Forecast (CF) Convention General

1. Common Information Model (CIM) Inputs

- Convention: CIM 1.5
- Purpose: Inputs Description
- Includes:
  - ESG (or ESMF) General
  - CF Extended
  - CF General
- Output Options:
  - CIM XML
- Description: This package is used to describe a simulation and the input (initial and boundary) conditions used in that simulation. It is also used to describe any ancillary data sets that contain input condition variables. This package should not be used to describe the variables in an unconfigured model component. A pre-defined Attribute package for that case will be implemented in a future release of ESMF. This package nests the ESG (or ESMF) General, CF Extended, and CF General Field packages (below) within it; this is described in Section 35.3. The attribute values within these ESG and CF nested packages currently appear in the Component Properties section of the CIM output file. A separate Component Properties package may be developed for this purpose in a future ESMF release.

Name	Definition	Controlled Vocabulary
CouplingPurpose*	The form of the input condition (e.g. initial condition or boundary condition).	Ancillary, Boundary, Initial
CouplingSource*	The component the input condition is coming from.	N/A
CouplingTarget*	The component the input condition is going to.	N/A
Description	A multi-line description of the input.	N/A
Frequency	The frequency (e.g. 2 months or 5 days) that a field from one component is input to another.	n Seconds, n Minutes, n Hours, n Days, n Months, n Years, n Decades, n Centuries
SpatialRegriddingMethod	Method used to interpolate a field from one grid (source grid) to another (target grid).	Linear, Near-Neighbor, Cubic, Conservative-First-Order, Conservative-Second-Order, Conservative, Non-Conservative
SpatialRegriddingDimension	Dimension of the regridding method.	1D, 2D, 3D
Technique	The software package or mechanism used to transfer and transform data between model components.	CCSM Flux Coupler, ESMF, Files, FMS, MCT, OASIS3, Shared
TimeTransformationType	Temporal transformation performed on the input field before or after regridding onto the target grid.	Exact, None, Time Accumulation, Time Average, Time Interpolation

* Attribute required to be set, when any other attributes in this package are set, to produce valid CIM XML output. It is valid to set none of the attributes in this package. In that case, no corresponding CIM XML output will appear for that Inputs package.

2. Earth System Grid (ESG) Field

- Convention: ESG or ESMF
- Purpose : General
- Includes:
  - CF Extended
  - CF General
- Output Options:
  - Tab-delimited
  - Simple XML

- CIM XML (when part of the CIM Inputs package)

- Description: ESG has the ability to list variables as either import or export variables. This should not be confused with the ESMF State Attribute Package, which has similiar attributes. This attribute is assigned to individual variables. This package nests the CF Extended and CF General packages (below) within it; this is described in Section 35.3.

Name	**Definition**	**Controlled Vocabulary**
`Intent`	An indication of whether a variable is exported or imported. This refers to coupling and not history output.	Export,Import

3. Climate Forecast (CF) Convention Extended

- Convention: CF
- Purpose: Extended
- Includes:
  - CF General
- Output Options:
  - Tab-delimited
  - Simple XML
  - CIM XML (when part of the CIM Inputs package)
- Description: The CF standard for fields contains an optional standard_name Attribute. Standard names are controlled vocabularies and not every variable in the earth system sciences contains a standard name. Because of this, ESMF implemented this optional Attribute in its own package. This package nests the CF General package (below) within it; this is described in Section 35.3.

Name	**Definition**	**Controlled Vocabulary**
`StandardName`	The approved CF standard name for a variable if it exists.	N/A

4. Climate Forecast (CF) Convention General

- Convention: CF
- Purpose: General
- Output Options:
  - Tab-delimited
  - Simple XML
  - CIM XML (when part of the CIM Inputs package)
- Description: The climate and forecast (CF) convention contains metadata that is designed to promote the processing and sharing of files created with the NetCDF API. The CF conventions are increasingly gaining acceptance and have been adopted by a number of projects and groups as a primary standard. The conventions define metadata that provide a definitive description of what the data in each variable represents, and the spatial and temporal properties of the data. This enables users of data from different sources to decide which quantities are comparable, and facilitates building applications with powerful extraction, regridding, and display capabilities. The ESMF CF Attribute package contains the three mandatory Attributes required to describe fields.

Name	Definition	Controlled Vocabulary
`LongName`	An ad-hoc long descriptive name which may, for example, be used for labeling plots	N/A
`ShortName*`	The short_name is technically not part of the CF stanard but is commonly the name of the variable on the output file and so is distinct from the long_name	N/A
`Units`	The value of the units attribute is a string that can be recognized by UNIDATA's Udunits package	N/A

* Attribute required to be set, if any attributes are set within this package, the CF/Extended, or ESMF/General package, to produce valid CIM XML output. It is valid to set none of the attributes in this package, the CF/Extended, or ESMF/General package, in which case no field CIM output will be produced.

35.2.4 Array Attribute packages

At this time the Array packages are the same as the Field packages.

35.2.5 Grid Attribute packages

There are 2 grid attribute packages in ESMF.

1. GFDL Gridspec
2. ESMF Grid

1. Gridspec

- Convention: GridSpec
- Purpose: General
- Output Options:
  - Simple XML
  - If combined with the ESMF Grid package, this package can be used to create a Grid object by reading in an XML file with these attributes. This is currently limited to 2D regularly distributed rectinlinear grids. See Section 27.3.24 for details.
- Description: This package contains the Attributes developed as part of GFDL's Gridspec standard.

Name	Definition	Controlled Vocabulary

`CongruentTiles`	Indicates whether or not all the tiles contained within a grid mosaic are of the same size and shape.	True, False
`DimensionOrder`	The order in which latitude and longitude appear within the two dimensional grid array.	N/A
`DiscretizationType`	Specifies the method by which a two-dimensional coordinate system is sampled to form a computational grid.	Logically rectangular, Pixel-based catchment, Structured triangular, Unstructured triangular
`GeometryType`	Indicates the geometric figure used to approximate the shape of the Earth, e.g. "sphere".	Ellipsoid, Plane, Sphere
`GridType`	A text description of the grid that uses common terminology.	Gnomonic,Cubed Sphere,Displaced Pole, Icosahedral geodesic, Reduced gaussian, Regular lat lon, Spectral gaussian, Tripolar, Yin Yang
`HorizontalResolution`	The number of points within the horizontal domain of a model or grid.	N/A
`IsConformal`	Indicates if the grid tile is angle-preserving. If so, angles measured on the grid are equal to the equivalent angles on the Earth.	True, False
`IsRegular`	Indicates whether or not the horizontal coordinates of the grid can be defined using 1D arrays (vectors). This means that grid node locations are defined by the cartesian product of the X/Lon and Y/Lat coordinate vectors. It also means that grid cells are logically rectangular (they may also be physically rectangular in the case of projected coordinates).	True, False
`IsUniform`	Indicates whether or not the horizontal coordinates of a grid have fixed offsets in the X and Y directions. If the offset is the same in both directions then the grid is logically square, otherwise it is logically rectangular.	True, False
`NorthPoleLocation`	Defines the lat-long position of the 'north pole' used by the grid tile in the case of rotated/displaced pole grids.	N/A
`NumberOfCells`	The number of cells in an unstructured grid.	N/A
`NumberOfGridTiles`	The number of tiles in a mosaic.	N/A

NX	Specifies the length of the X, or longitude, dimension of the grid tile.	N/A
NY	Specifies the length of the Y, or latitude, dimension of the grid tile.	N/A

2. ESMF Grid

- Convention: ESMF
- Purpose: General
- Includes:
  - GridSpec General
- Output Options:
  - Simple XML
  - If combined with the Gridspec Package, this package can be used to create a Grid object by reading in an XML file with these attributes. This is currently limited to 2D regularly distributed logically-rectangular grids. See Section 27.3.24 for details.
- Description: This package contains two Attributes required by ESMF to generate grids from a file. They describe the decomposition of a grid across ESMF DEs. This package nests the GridSpec General package (above) within it; this is described in Section 35.3.

Name	Definition	Controlled Vocabulary
RegDecompX	The number of DEs in X a particular grid is decomposed into.	N/A
RegDecompY	The number of DEs in Y a particular grid is decomposed into.	N/A

35.2.6 Table of available Attributes

The following is an alphabetical list of all the attributes implemented in ESMF, their definitions, and which packages they are contained within.

Name	Definition	Attribute Package
Agency	An administrative unit of government.	ESG Basic Component
CodingLanguage	The computer language in which a unit of software is written.	ESG Basic Component
CompilerName	The brand of the software that takes source code and turns it into an executable.	CIM Platform

CompilerVersion	The specific configuration value of the software used to take source code and turn it into executable code.	CIM Platform
CouplingPurpose	The form of the input condition (e.g. initial condition or boundary condition).	CIM Inputs
CouplingSource	The component the input condition is coming from.	CIM Inputs
CouplingTarget	The component the input condition is going to.	CIM Inputs
Description	A multi-line description of a component or input.	CIM Main, CIM Inputs
Date	The date of the citation.	ISO Citation
DimensionOrder	The order in which latitude and longitude appear within the two dimensional grid array.	Gridspec
DiscretizationType	Specifies the method by which a two-dimensional coordinate system is sampled to form a computational grid.	Gridspec
DOI	The assigned Digital Object Identifier (DOI) of the citation.	ISO Citation
EmailAddress	The email address that others can use to ask questions about a model component.	ISO Responsible Party
Frequency	The frequency (e.g. months, days) that a field from one component is input to another.	CIM Inputs
FullName	The name of a model, model component, simulation, experiment, or dataset with all acronyms spelled out.	ESG Basic Component
GeometryType	Indicates the geometric figure used to approximate the shape of the Earth, e.g. "sphere".	Gridspec
GridType	A text description of the grid that uses common terminology.	Gridspec
IsConformal	Indicates if the grid tile is angle-preserving. If so, angles measured on the grid are equal to the equivalent angles on the Earth.	Gridspec

IsRegular	Indicates whether or not the horizontal coordinates of the grid can be defined using 1D arrays (vectors). This means that grid node locations are defined by the cartesian product of the X/Lon and Y/Lat coordinate vectors. It also means that grid cells are logically rectangular (they may also be physically rectangular in the case of projected coordinates).	Gridspec
Institution	An organization associated with a model component, simulation, or dataset.	ESG Basic Component
Intent	An indication of whether a field or state is imported into or exported from a particular model component. This refers to coupling, and not history output.	ESMF State, ESMF Field
IsUniform	Indicates whether or not the horizontal coordinates of a grid have fixed offsets in the X and Y directions. If the offset is the same in both directions then the grid is logically square, otherwise it is logically rectangular.	Gridspec
LongName	The name of an object with all acronyms spelled out. For fields, it is an ad-hoc long descriptive name which may, for example, be used for labeling plots.	CIM Main, CF General
LongTitle	The text of the citation or pointer (e.g. URL) that further describes a model component or simulation.	ISO Citation
MachineCoresPerProcessor	The number of sub-divided elements or mini-chips on a computer chip.	CIM Platform
MachineDescription	A short note about the machine.	CIM Platform
MachineInterconnectType	The technology used to associate each node in a supercomputer with every other node.	CIM Platform
MachineMaximumProcessors	The highest number of computer chips on a computer system.	CIM Platform
MachineName	The name given to a computer by its system administrators. This is not the brand name of the system.	CIM Platform
MachineOperatingSystem	The software that is responsible for the management and coordination of activities and the sharing of resources of a computer.	CIM Platform
MachineProcessorType	The type of computer chip used in a particular computer platform.	CIM Platform

`MachineSystem`	The type of computer system (e.g. vector, parallel, cluster, etc.).	CIM Platform
`MachineVendor`	The brand name of a computer system.	CIM Platform
`MetadataVersion`	The version number of the simulation metadata.	CIM Main
`ModelComponentFramework`	The software package or mechanism used to transfer and transform data between model components.	ESG Basic Component
`ModelType`	A short string describing the discipline of a model component.	CIM Main
`Name`	The name of an author, contact, funder, centre, or principal investigator.	ISO Responsible Party
`NameType`	The type of entity that Name references.	ISO Responsible Party
`NorthPoleLocation`	Defines the lat-long position of the 'north pole' used by the grid tile in the case of rotated/displaced pole grids.	Gridspec
`NumberOfCells`	The number of cells in an unstructured grid.	Gridspec
`NX`	Specifies the length of the X, or longitude, dimension of the grid tile.	Gridspec
`NY`	Specifies the length of the Y, or latitude, dimension of the grid tile.	Gridspec
`PhysicalAddress`	The address of the person designated to provide information about a model component.	ISO Responsible Party
`PhysicalDomain`	A description of the geographic range being simulated.	ESG Basic Component
`PresentationForm`	A description of the type of citation.	ISO Citation
`PreviousVersion`	Name of the previous version of a model or model component.	CIM Main
`PreviousVersionDescription`	A short note about the previous version of the model or model component.	CIM Main
`ReleaseDate`	The year a model component was issued.	CIM Main
`RegDecompX`	The number of DEs in X a particular grid is decomposed into.	ESMF Grid
`RegDecompY`	The number of DEs in Y a particular grid is decomposed into.	ESMF Grid
`ResponsiblePartyRole`	A flag to define the role of the Responsible Party.	ISO Responsible Party

ShortName	For component: a version of the component name that contains acronyms. For field: The short_name is technically not part of the CF stanard but is commonly the name of the variable on the output file and so is distinct from the long_name.	CIM Main, CF General
ShortTitle	An abbreviation for the citation. This could be the short scientific citation (e.g. Murphy, 2009) or the title of a web page.	ISO Citation
SimulationDuration	The length of time a simulation runs.	CIM Main
SimulationEndDate	The date in simulated time of the end of a model simulation.	CIM Main
SimulationEnsembleID	The reference name or number of the ensemble to which a simulation belongs.	CIM Main
SimulationLongName	The name of the simulation with any acronyms spelled out.	CIM Main
SimulationNumberOfProcessingElements	The number of PEs used in the simulation.	CIM Main
SimulationProjectName	A campaign, such as a model intercomparison project, that may involve multiple groups and experiments.	CIM Main
SimulationRationale	The reason for performing a simulation.	CIM Main
SimulationStartDate	The date in simulated time of the start of a model simulation.	CIM Main
SimulationShortName	The name of the simulation.	CIM Main
SpatialRegriddingMethod	Method used to interpolate a field from one grid (source grid) to another (target grid).	CIM Inputs
SpatialRegriddingDimension	Dimension of the regridding method.	CIM Inputs
StandardName	The approved CF standard name for a variable if it exists.	CF Extended
TimeTransformationType	Temporal transformation performed on the input field before or after regridding onto the target grid.	CIM Inputs
Technique	The software package or mechanism used to transfer and transform data between model components.	CIM Inputs
URL	URL of the object being described. Exists in multple packages.	CIM Main, ISO Responsible Party, ISO Citation

`Units`	The value of the units attribute is a string that can be recognized by UNIDATA's Udunits package.	CF General
`Version`	A specific form or variation of an artifact i.e. a unit of software or metadata.	CIM Main, ESG Basic Component

35.2.7 Custom Attribute packages

ESMF allows for the creation of custom attribute packages, each of which has a user-defined convention and purpose, as well as a set of user-defined attributes. This can be done to augment one of the pre-defined packages (via package nesting 35.3) or to create a suite of attributes unique to the user. Examples of how to create such custom packages are contained in Sections 35.6.2 and 35.6.3.

35.3 Attribute Packages Nesting

Nesting is a way of creating larger Attribute packages out of smaller ones and allows users to add the attributes they want to an existing package. It is very useful when combining a custom package with a pre-defined package. One or more child Attribute packages can be nested within a parent package, and this can be repeated multiple times, allowing a full Attribute tree (hierarchical) structure to be created. Breaking Attributes up into smaller packages that are then nested also allows for the construction of complex attribute trees where certain structures repeat themselves, allowing for Attribute package reusability.
Several of the ESMF pre-defined packages, when added to an ESMF object, are created with nested packages:
CIM Main – Component package – is a nest with three child packages:

1. CIM Platform
2. CIM Responsible Party (one or more – user specifiable)
3. CIM Citation (one or more – user specifiable)

CIM Inputs – Field package – is a nest with one child package:

1. ESG (or ESMF) General (with CF Extended and CF General packages nested within it)

ESG (or ESMF) General – Field package – is a nest with one child package:

1. CF Extended (with a CF General package nested within it)

CF Extended – Field package – is a nest with one child package:

1. CF General

ESMF General – Grid package – is a nest with one child package:

1. GridSpec General

An explanation of the Attribute packages specifiers is in order at this point. The purpose specifier is really just meant as an additional means, beyond the use of "convention", to specify Attribute packages. One could imagine that the CF convention would want to be able to have Attribute packages divided up in some fashion, which ESMF could then keep track of with the purpose specifier. It was added with the intention of allowing Attributes, and packages, maximum flexibility. Take the Field's ESMF standard Attribute package for example. This package is made up of three nested Attribute packages. The lowest one is made up of three Attributes with convention=CF and purpose=General. The next level contains one Attribute with convention=CF but purpose=Extended. On top of this is the convention=ESG package, also with purpose=General.

35.4 Export Formats

The `ESMF_AttributeWrite()` interface is used to write the contents of an Attribute package to a file. This routine can be called on any ESMF object that is capable of holding Attribute packages. It can also write out all Attributes in Attribute packages with the same specifiers throughout an entire ESMF object hierarchy.
There are three primary ways of exporting Attributes:

1. Tab-delimited ASCII
2. Simple XML
3. CIM XML

The flag that is used in the `ESMF_AttributeWrite()` interface to determine which format for writing the Attribute packages is called the `ESMF_AttWriteFlag`, with values as described below. The resulting file will be placed in the execution directory after it is written and closed.

35.4.1 Tab-delimited ASCII

When `ESMF_AttWriteFlag` is set to `ESMF_ATTWRITE_TAB` (the default), a tab-delimited ascii file containing name-value pairs of attributes in the packages will be written. The file will be named for the name of the ESMF object from which `ESMF_AttributeWrite()` is called. The suffix will be .stdout.

35.4.2 Simple XML

When `ESMF_AttWriteFlag` is set to `ESMF_ATTWRITE_XML`, an XML file containing name-value pairs of attributes in the packages will be written. The file will be named for the name of the ESMF object from which `ESMF_AttributeWrite()` is called. The suffix will be .xml.

35.4.3 CIM XML

When the ESMF object from which `ESMF_AttributeWrite()` is called is a Component, and the Attribute package convention="CIM 1.5", and the purpose="Model Component Simulation Description", and `ESMF_AttWriteFlag` is set to `ESMF_ATTWRITE_XML`, an XML file conforming to the CIM standard will be written. The file will contain Attributes from the entire Component tree and their contained Fields. The file will be named for the name of the ESMF Component object from which `ESMF_AttributeWrite()` is called, and the suffix will be .xml.

35.5 Constants

35.5.1 ESMF_ATTGETCOUNT

DESCRIPTION:
Indicates whether or not to descend the Attribute hierarchy for the present operation.
The type of this flag is:
`type(ESMF_AttGetCountFlag)`
The valid values are:

ESMF_ATTGETCOUNT_ATTRIBUTE This option will allow the routine to return the number of single Attributes.

ESMF_ATTGETCOUNT_ATTPACK This option will allow the routine to return the number of Attribute packages.

ESMF_ATTGETCOUNT_ATTLINK This option will allow the routine to return the number of Attribute links.

ESMF_ATTGETCOUNT_TOTAL This option will allow the routine to return the total number of Attributes.

35.5.2 ESMF_ATTTREE

DESCRIPTION:
Indicates whether or not to descend the Attribute hierarchy for the present operation.
The type of this flag is:
`type(ESMF_AttTreeFlag)`
The valid values are:

ESMF_ATTTREE_OFF This option will allow the routine to only descend the first base level of the Attribute hierarchy.

ESMF_ATTTREE_ON This option will allow the routine to descend the entire Attribute hierarchy.

35.5.3 ESMF_ATTWRITE

DESCRIPTION:
Indicates which file format to use in the write operation.
The type of this flag is:
`type(ESMF_AttWriteFlag)`
The valid values are:

ESMF_ATTWRITE_XML This option will allow the routine to write in xml format.

ESMF_ATTWRITE_TAB This option will allow the routine to write in tab-delimited format.

35.6 Use and Examples

This section describes the use of the Attribute class. There are eight examples that follow, which outline the use of Attributes at three increasing levels of difficulty. The first example covers basic Attribute manipulations on the gridded Component. The second example covers the Attribute package capabilities, including Attribute package nesting and Attribute hierarchy linking. The third example covers Attribute management in a distributed environment and the I/O utilities. These examples will be best understood if followed in an ascending order from basic to advanced. The fourth example shows how to use the CIM Attribute packages. The last four examples cover setting of Attribute packages and custom Attributes from an XML file.

35.6.1 Basic Attribute usage

This example illustrates the most basic usage of the Attribute class. This demonstration of Attribute manipulation is limited to the gridded Component, but the same principles apply to the coupler Component, State, Grid, FieldBundle, Field, ArrayBundle and Array. The functionality that is demonstrated includes setting and getting Attributes, working with Attributes with different types and lists, removing Attributes, and getting default Attributes. Various other uses of `ESMF_AttributeGet()` is covered in detail in the last section. The first thing we must do is declare variables and initialize ESMF.

```
! Use ESMF framework module
use ESMF
implicit none

! Local variables
integer                 :: rc, finalrc, petCount, localPet, &
                           itemCount, count
type(ESMF_VM)           :: vm
type(ESMF_GridComp)     :: gridcomp
character(ESMF_MAXSTR)  :: name
type(ESMF_TypeKind_Flag)     :: tk

integer(ESMF_KIND_I4)                :: inI4
```

```
integer(ESMF_KIND_I4), dimension(3)  :: inI4l
integer(ESMF_KIND_I8)                :: inI8
integer(ESMF_KIND_I8), dimension(3)  :: inI8l
real(ESMF_KIND_I4)                   :: inR4
real(ESMF_KIND_I4), dimension(3)     :: inR4l
real(ESMF_KIND_I8)                   :: inR8
real(ESMF_KIND_I8), dimension(3)     :: inR8l
character(ESMF_MAXSTR)               :: inChar
character(ESMF_MAXSTR), dimension(3) :: inCharl, &
                                     defaultCharl, dfltoutCharl
character(ESMF_MAXSTR), dimension(8) :: outCharl
logical                              :: inLog
logical, dimension(3)                :: inLogl, value

! initialize ESMF
finalrc = ESMF_SUCCESS
call ESMF_Initialize(vm=vm, defaultlogfilename="AttributeEx.Log", &
              logkindflag=ESMF_LOGKIND_MULTI, rc=rc)

! get the vm
call ESMF_VMGet(vm, petCount=petCount, localPet=localPet, rc=rc)
```

We will construct the gridded Component which will be responsible for all of the Attributes we will be manipulating.

```
if (petCount<4) then
  gridcomp = ESMF_GridCompCreate(name="gridcomp", &
    petList=(/0/), rc=rc)
else
  gridcomp = ESMF_GridCompCreate(name="gridcomp", &
    petList=(/0,1,2,3/), rc=rc)
endif
```

We can set Attributes using the `ESMF_AttributeSet()` command. Attributes can be any of several different types, all of which are demonstrated here.

```
inI4 = 4
inI4l = (/1,2,3/)
inI8 = 4
inI8l = (/1,2,3/)
inR4 = 4
inR4l = (/1,2,3/)
inR8 = 4
inR8l = (/1,2,3/)
inChar = "Character string 4"
inCharl = (/ "Character string 1", &
             "Character string 2", &
             "Character string 3" /)
inLog = .true.
inLogl = (/.true., .false., .true. /)

call ESMF_AttributeSet(gridcomp, name="ESMF_I4name", value=inI4, rc=rc)
call ESMF_AttributeSet(gridcomp, name="ESMF_I4namelist", &
  valueList=inI4l, rc=rc)
call ESMF_AttributeSet(gridcomp, name="ESMF_I8name", value=inI8,  rc=rc)
```

```
call ESMF_AttributeSet(gridcomp, name="ESMF_I8namelist", &
  valueList=inI8l, rc=rc)
call ESMF_AttributeSet(gridcomp, name="ESMF_R4name", value=inR4, rc=rc)
call ESMF_AttributeSet(gridcomp, name="ESMF_R4namelist", &
  valueList=inR4l, rc=rc)
call ESMF_AttributeSet(gridcomp, name="ESMF_R8name", value=inR8, rc=rc)
call ESMF_AttributeSet(gridcomp, name="ESMF_R8namelist", &
  valueList=inR8l, rc=rc)
call ESMF_AttributeSet(gridcomp, name="Character_name", &
  value=inChar, rc=rc)
call ESMF_AttributeSet(gridcomp, name="Character_namelist", &
  valueList=inCharl, rc=rc)
call ESMF_AttributeSet(gridcomp, name="Logical_name", value=inLog, rc=rc)
call ESMF_AttributeSet(gridcomp, name="Logical_namelist", &
  valueList=inLogl, rc=rc)
```

We can retrieve Attributes by issuing the `ESMF_AttributeGet()` command. This command can also be used with an optional default value (or value list) so that if the Attribute is not found a value is returned without an error code. Removal of Attributes is also possible, and is demonstrated here as well. One of the Attributes previously created will be retrieved, then removed, then retrieved again using a default return value. In order to use the default return value capabilites, we must first set up a default parameter.

```
defaultCharl = (/ "Character string 4", &
                  "Character string 5", &
                  "Character string 6" /)

itemCount=3
call ESMF_AttributeGet(gridcomp, name="Character_namelist", &
  valueList=outCharl(1:5), itemCount=itemCount, rc=rc)

call ESMF_AttributeRemove(gridcomp, name="Character_namelist", rc=rc)

call ESMF_AttributeGet(gridcomp, name="Character_namelist", &
  valueList=dfltoutCharl, defaultvalueList=defaultCharl,rc=rc)
```

There are more overloaded instances of `ESMF_AttributeGet()` which allow the retrieval of Attribute information by name or index number, or a query for the count of the Attributes on a certain object. These capabilities are demonstrated here by first retrieving the name of an Attribute using the index number, keep in mind that these index numbers start from 1. Then the name that is retrieved is used to get other information about the Attribute, such as the typekind, and the number of items in the value of the Attribute. This information is then used to actually retreive the Attribute value. Then the count of the number of Attributes on the object will be retrieved.

```
call ESMF_AttributeGet(gridcomp, attributeIndex=11 , name=name, rc=rc)

call ESMF_AttributeGet(gridcomp, name=name, typekind=tk, &
  itemCount=itemCount, rc=rc)

if (tk==ESMF_TYPEKIND_Logical .AND. itemCount==3) then
  call ESMF_AttributeGet(gridcomp, name=name, valueList=value, rc=rc)
endif

call ESMF_AttributeGet(gridcomp, count=count, rc=rc)
```

35.6.2 Attribute packages

This example is slightly more complex than the example presented in section 35.6.1 and illustrates the use of the Attribute class to create Attribute hierarchies using Attribute packages. A gridded Component is used in conjunction with two States, a FieldBundle, and various realistic Fields to create an Attribute hierarchy and copy it from one State to another. Attribute packages are created on the Component and Fields, and the standard Attributes in each package are used in the Attribute hierarchy. The Attribute package nesting capability is demonstrated by nesting the standard ESMF supplied packages for the Fields inside a user specified Attribute package with a customized convention.
We must construct the ESMF objects that will be responsible for the Attributes we will be manipulating. These objects include the gridded Component, two States, a FieldBundle, and 10 Fields. In this trivial example we are constructing empty Fields with no underlying Grid.

```
if (petCount<4) then
  gridcomp = ESMF_GridCompCreate(name="gridded_comp_ex2", &
    petList=(/0/), rc=rc)
else
  gridcomp = ESMF_GridCompCreate(name="gridded_comp_ex2", &
    petList=(/0,1,2,3/), rc=rc)
endif
importState = ESMF_StateCreate(name="importState",  &
                               stateintent=ESMF_STATEINTENT_IMPORT, rc=rc)
exportState = ESMF_StateCreate(name="exportState",  &
                               stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)

DPEDT = ESMF_FieldEmptyCreate(name='DPEDT', rc=rc)
DTDT = ESMF_FieldEmptyCreate(name='DTDT', rc=rc)
DUDT = ESMF_FieldEmptyCreate(name='DUDT', rc=rc)
DVDT = ESMF_FieldEmptyCreate(name='DVDT', rc=rc)
PHIS = ESMF_FieldEmptyCreate(name='PHIS', rc=rc)
QTR = ESMF_FieldEmptyCreate(name='QTR', rc=rc)
CNV = ESMF_FieldEmptyCreate(name='CNV', rc=rc)
CONVCPT = ESMF_FieldEmptyCreate(name='CONVCPT', rc=rc)
CONVKE = ESMF_FieldEmptyCreate(name='CONVKE', rc=rc)
CONVPHI = ESMF_FieldEmptyCreate(name='CONVPHI', rc=rc)

fbundle = ESMF_FieldBundleCreate(name="fbundle", rc=rc)
```

Now we can add Attribute packages to all of the appropriate objects. We will use the ESMF supplied Attribute packages for the Fields and the Component. On the Fields, we will first use `ESMF_AttributeAdd()` to create standard Attribute packages, then we will nest customized Attribute packages around the ESMF standard Attribute packages. In this simple example the purpose for the Attribute packages will be specified as "General" in all cases.

```
convESMF = 'ESMF'
convCC = 'CustomConvention'
purpGen = 'General'

attrList(1) = 'Coordinates'
attrList(2) = 'Mask'

! DPEDT
call ESMF_AttributeAdd(DPEDT, convention=convESMF, purpose=purpGen, &
  rc=rc)
call ESMF_AttributeAdd(DPEDT, convention=convCC, purpose=purpGen,   &
  attrList=attrList, nestConvention=convESMF, nestPurpose=purpGen,  &
  rc=rc)
```

... and so on for the other 9 Fields.
The standard Attribute package currently supplied by ESMF for Field contains 6 Attributes, 2 of which are set automatically. The remaining 4 Attributes in the standard Field Attribute package must be set manually by the user. We must also set the Attributes of our own custom Attribute package, which is built around the ESMF standard Attribute package.

```
      name1 = 'ShortName'
      name2 = 'StandardName'
      name3 = 'LongName'
      name4 = 'Units'

      ! DPEDT
      value1 = 'DPEDT'
      value2 = 'tendency_of_air_pressure'
      value3 = 'Edge pressure tendency'
      value4 = 'Pa s-1'
      ! Custom Attributes
      call ESMF_AttributeSet(DPEDT, name='Coordinates', value='latlon', &
        convention=convCC, purpose=purpGen, rc=rc)
      call ESMF_AttributeSet(DPEDT, name='Mask', value='yes', &
        convention=convCC, purpose=purpGen, rc=rc)
      ! ESMF Attributes
      call ESMF_AttributeSet(DPEDT, name1, value1, convention=convESMF, &
        purpose=purpGen, rc=rc)
      call ESMF_AttributeSet(DPEDT, name2, value2, convention=convESMF, &
        purpose=purpGen, rc=rc)
      call ESMF_AttributeSet(DPEDT, name3, value3, convention=convESMF, &
        purpose=purpGen, rc=rc)
      call ESMF_AttributeSet(DPEDT, name4, value4, convention=convESMF, &
        purpose=purpGen, rc=rc)
```

... and so on for the other 9 Fields.
The standard Attribute package currently supplied by ESMF for Component contains 10 Attributes. These Attributes conform to both the ESG and CF conventions, and must be set manually.

```
    call ESMF_AttributeSet(gridcomp, 'Agency', 'NASA', &
      convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'Author', 'Max Suarez', &
      convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'CodingLanguage', &
      'Fortran 90', convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'Discipline', &
      'Atmosphere', convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'ComponentLongName', &
      'Goddard Earth Observing System Version 5 Finite Volume Dynamical Core', &
        convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'ModelComponentFramework', &
      'ESMF', convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'ComponentShortName', &
      'GEOS-5 FV dynamical core', &
      convention=convESMF, purpose=purpGen, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'PhysicalDomain', &
```

```
    'Earth system', convention=convESMF, purpose=purpGen, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'Version', &
    'GEOSagcm-EROS-beta7p12', convention=convESMF, purpose=purpGen, rc=rc)
```

Adding the Fields to the FieldBundle will automatically "link" the Attribute hierarchies. The same type of link will be generated when adding a FieldBundle to a State.

```
call ESMF_FieldBundleAdd(fbundle, (/DPEDT/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/DTDT/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/DUDT/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/DVDT/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/PHIS/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/QTR/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/CNV/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/CONVCPT/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/CONVKE/), rc=rc)
call ESMF_FieldBundleAdd(fbundle, (/CONVPHI/), rc=rc)

call ESMF_StateAdd(exportState, fieldbundleList=(/fbundle/), rc=rc)
```

The link between a State and the Component of interest must be set manually.

```
call ESMF_AttributeLink(gridcomp, exportState, rc=rc)
```

There are currently two different formats available for writing the contents of the Attribute packages in an Attribute hierarchy. There is an XML formatted write, which generates an .xml file in the execution directory with the contents of the write. There is also a tab-delimited write which writes to standard out, a file generated in the execution directory with the extension .stdout. Either of the `ESMF_AttributeWrite()` formats can be called on any of the objects which are capable of manipulating Attributes, but only from objects in an Attribute hierarchy which contain ESMF standard Attribute packages can it be confirmed that any relevant information be written. The `ESMF_AttributeWrite()` capability is only functional for single-item Attributes at this point, it will be more robust in future releases. A flag is used to specify which format to write, the default is tab-delimited.

```
call ESMF_AttributeWrite(gridcomp,convESMF,purpGen, &
  attwriteflag=ESMF_ATTWRITE_XML,rc=rc)
call ESMF_AttributeWrite(gridcomp,convESMF,purpGen,rc=rc)
```

35.6.3 Custom Attribute package

This example illustrates how to create a user-defined, custom Attribute package. The package is created on a gridded Component with three custom Attributes.
We must construct the ESMF gridded Component object that will be responsible for the custom Attribute package we will be manipulating.

```
if (petCount<4) then
  gridcomp = ESMF_GridCompCreate(name="gridded_comp_ex3", &
    petList=(/0/), rc=rc)
else
  gridcomp = ESMF_GridCompCreate(name="gridded_comp_ex3", &
    petList=(/0,1,2,3/), rc=rc)
endif
```

Now we can add a custom Attribute package to the gridded Component object.

```
      customConv = 'CustomConvention'
      customPurp = 'CustomPurpose'

      customAttrList(1) = 'CustomAttrName1'
      customAttrList(2) = 'CustomAttrName2'
      customAttrList(3) = 'CustomAttrName3'

      call ESMF_AttributeAdd(gridcomp, convention=customConv, &
        purpose=customPurp, attrList=customAttrList, rc=rc)
```

We must set the Attribute values of our custom Attribute package.

```
    call ESMF_AttributeSet(gridcomp, 'CustomAttrName1', 'CustomAttrValue1', &
      convention=customConv, purpose=customPurp, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'CustomAttrName2', 'CustomAttrValue2', &
      convention=customConv, purpose=customPurp, rc=rc)
    call ESMF_AttributeSet(gridcomp, 'CustomAttrName3', 'CustomAttrValue3', &
      convention=customConv, purpose=customPurp, rc=rc)
```

Write out the contents of our custom Attribute package to an XML file, which is generated with a .xml file extension in the execution directory.

```
      call ESMF_AttributeWrite(gridcomp,customConv,customPurp, &
        attwriteflag=ESMF_ATTWRITE_XML,rc=rc)
```

35.6.4 Updating Attributes in a distributed environment

This advanced example illustrates the proper methods of Attribute manipulation in a distributed environment to ensure consistency of metadata across the VM. This example is much more complicated than the previous two because we will be following the flow of control of a typical model run with two gridded Components and one coupling Component. We will start out in the application driver, declaring Components, States, and the routines used to initialize, run and finalize the user's model Components. Then we will follow the control flow into the actual Component level through initialize, run, and finalize examining how Attributes are used to organize the metadata.

This example follows a simple user model with two gridded Components and one coupling Component. The initialize routines are used to set up the application data and the run routines are used to manipulate the data. Accordingly, most of the Attribute manipulation will take place in the initialize phase of each of the three Components. The two gridded Components will be running on exclusive pieces of the VM and the coupler Component will encompass the entire VM so that it can handle the Attribute communications.

The control flow of this example will start in the application driver, after which it will complete three cycles through the three Components. The first cycle will be through the initialize routines, from the first gridded Component to the second gridded Component to the coupler Component. The second cycle will go through the run routines, from the first gridded Component to the coupler Component to the second Gridded component. The third cycle will be through the finalize routines in the same order as the first cycle.

In the application driver, we must now construct some ESMF objects, such as the gridded Components, the coupler Component, and the States. This is also where it is determined which subsets of the PETs of the VM the Components will be using to run their initialize, run, and finalize routines.

```
        gridcomp1 = ESMF_GridCompCreate(name="gridcomp1", &
          petList=(/0,1/), rc=rc)
        gridcomp2 = ESMF_GridCompCreate(name="gridcomp2", &
```

```
      petList=(/2,3/), rc=rc)
    cplcomp = ESMF_CplCompCreate(name="cplcomp", &
      petList=(/0,1,2,3/), rc=rc)

  c1exp = ESMF_StateCreate(name="Comp1 exportState", &
                           stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)
  c2imp = ESMF_StateCreate(name="Comp2 importState", &
                           stateintent=ESMF_STATEINTENT_IMPORT, rc=rc)
```

Before the individual components are initialized, run, and finalized Attributes should be set at the Component level. Here we are going to use the ESG Attribute package on the first gridded Component. The Attribute package is added, and then each of the Attributes is set. The Attribute hierarchy of the Component is then linked to the Attribute hierarchy of the export State in a manual fashion.

```
  convESMF = 'ESMF'
  purpGen = 'General'
call ESMF_AttributeAdd(gridcomp1, convention=convESMF, purpose=purpGen, &
  rc=rc)
call ESMF_AttributeSet(gridcomp1, 'Agency', 'NASA', &
  convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'Author', 'Max Suarez', &
  convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'CodingLanguage', &
  'Fortran 90', convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'Discipline', &
  'Atmosphere', convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'ComponentLongName', &
'Goddard Earth Observing System Version 5 Finite Volume Dynamical Core', &
    convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'ModelComponentFramework', &
  'ESMF', &
  convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'ComponentShortName', &
  'GEOS-5 FV dynamical core', convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'PhysicalDomain', &
  'Earth system', convention=convESMF, purpose=purpGen, rc=rc)
call ESMF_AttributeSet(gridcomp1, 'Version', &
  'GEOSagcm-EROS-beta7p12', convention=convESMF, purpose=purpGen, rc=rc)

  call ESMF_AttributeLink(gridcomp1, c1exp, rc=rc)
```

Now the individual Components will be run. First we will initialize the two gridded Components, then we will initialize the coupler Component. During each of these Component initialize routines Attribute packages will be added, and the Attributes set. The Attribute hierarchies will also be linked. As the gridded Components will be running on exclusive portions of the VM, the Attributes will need to be made available across the VM using an `ESMF_StateReconcile()` call in the coupler Component. The majority of the work with Attributes will take place in this portion of the model run, as metadata rarely needs to be changed during run time.
What follows are the calls from the driver code that run the initialize, run, and finalize routines for each of the Components. After these calls we will step through the first cycle as explained in the introduction, through the intialize routines of gridded Component 1 to gridded Component 2 to the coupler Component.

```
  call ESMF_GridCompInitialize(gridcomp1, exportState=c1exp, rc=rc)
  call ESMF_GridCompInitialize(gridcomp2, importState=c2imp, rc=rc)
  call ESMF_CplCompInitialize(cplcomp, importState=c1exp, &
```

```
    exportState=c2imp, rc=rc)

  call ESMF_GridCompRun(gridcomp1, exportState=c1exp, rc=rc)
  call ESMF_CplCompRun(cplcomp, importState=c1exp, &
    exportState=c2imp, userRc=urc, rc=rc)

  call ESMF_GridCompRun(gridcomp2, importState=c2imp, rc=rc)

  call ESMF_GridCompFinalize(gridcomp1, exportState=c1exp, rc=rc)
  call ESMF_GridCompFinalize(gridcomp2, importState=c2imp, rc=rc)
  call ESMF_CplCompFinalize(cplcomp, importState=c1exp, &
    exportState=c2imp, rc=rc)
```

In the first gridded Component initialize routine we need to create some Attribute packages and set all of the Attributes. These Attributes will be attached to realistic Fields, containing a Grid, which are contained in a FieldBundle. The first thing to do is declare variables and make the Grid.

```
type(ESMF_VM)               :: vm
integer                     :: petCount, status, myPet
character(ESMF_MAXSTR)      :: name1,name2,name3,name4,value1,value2, &
                               value3,value4,convESMF,purpGen,convCC
type(ESMF_ArraySpec)        :: arrayspec
type(ESMF_Grid)             :: grid
type(ESMF_Field)            :: DPEDT,DTDT,DUDT,DVDT,PHIS,QTR,CNV,CONVCPT, &
                               CONVKE,CONVPHI
type(ESMF_FieldBundle)      :: fieldbundle
character(ESMF_MAXSTR),dimension(2)   :: attrList

rc = ESMF_SUCCESS

call ESMF_GridCompGet(comp, vm=vm, rc=status)
call ESMF_VMGet(vm, petCount=petCount, localPet=myPet, rc=status)

call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, &
       rc=rc)
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/100,150/), &
  regDecomp=(/1,petCount/), &
  gridEdgeLWidth=(/0,0/), gridEdgeUWidth=(/0,0/), &
  indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

At this point the Fields will need to have Attribute packages attached to them, and the Attributes will be set with appropriate values.

```
convCC = 'CustomConvention'
convESMF = 'ESMF'
purpGen = 'General'
name1 = 'ShortName'
name2 = 'StandardName'
name3 = 'LongName'
name4 = 'Units'

value1 = 'DPEDT'
value2 = 'tendency_of_air_pressure'
value3 = 'Edge pressure tendency'
```

```
value4 = 'Pa s-1'

DPEDT = ESMF_FieldCreate(grid, arrayspec=arrayspec, &
          staggerloc=ESMF_STAGGERLOC_CENTER, rc=status)
call ESMF_AttributeAdd(DPEDT, convention=convESMF, purpose=purpGen, &
  rc=status)
call ESMF_AttributeSet(DPEDT, name1, value1, convention=convESMF, &
  purpose=purpGen, rc=status)
call ESMF_AttributeSet(DPEDT, name2, value2, convention=convESMF, &
  purpose=purpGen, rc=status)
call ESMF_AttributeSet(DPEDT, name3, value3, convention=convESMF, &
  purpose=purpGen, rc=status)
call ESMF_AttributeSet(DPEDT, name4, value4, convention=convESMF, &
  purpose=purpGen, rc=status)
```

... and so on for the other 9 Fields.
Now the Fields will be added to the FieldBundle, at which point the Attribute hierarchies of the Fields will also be attached to the Attribute hierarchy of the FieldBundle. After that, the FieldBundle will be attached to the export State, again at which time the Attribute hierarchy of the FieldBundle will be attached to the Attribute hierarchy of the export State.

```
fieldbundle = ESMF_FieldBundleCreate(name="fieldbundle", rc=status)
call ESMF_FieldBundleSet(fieldbundle, grid=grid, rc=status)

call ESMF_FieldBundleAdd(fieldbundle, (/DPEDT/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/DTDT/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/DUDT/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/DVDT/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/PHIS/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/QTR/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/CNV/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/CONVCPT/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/CONVKE/), rc=status)
call ESMF_FieldBundleAdd(fieldbundle, (/CONVPHI/), rc=status)

call ESMF_StateAdd(exportState, fieldbundleList=(/fieldbundle/), rc=status)
```

At this point, the driver of the model run will transfer control to the initialize phase of the second gridded Component. In the second gridded Component initialize routine we don't have anything to do. The data that was created in the initialize routine of the first gridded Component will be passed to this Component through the coupler Component. The data will not be used in this Component until the run phase of the model. So now the application driver transfers control to the initialize phase of the coupler Component.
In the coupler Component initialize routine all that is required is to ensure consistent data across the VM. The data created in the first gridded Component on one set of the PETs in the VM is intended to be read and manipulated by the second gridded Component which runs on an exclusive set of the PETs of the VM for this application. We need to first make that data consistent across the entire VM with the `ESMF_StateReconcile()` call. This State level call handles both the data – Fields and FieldBundles, and the metadata – Attribute and Attribute packages. There is a flag in this call to allow the user to specify whether they want the metadata to be reconciled or not.

```
type(ESMF_VM)         :: vm

rc = ESMF_SUCCESS
```

```
call ESMF_CplCompGet(comp, vm=vm, rc=rc)
call ESMF_StateReconcile(importState, vm=vm, &
           attreconflag=ESMF_ATTRECONCILE_ON, rc=rc)
call ESMF_StateReconcile(exportState, vm=vm, &
           attreconflag=ESMF_ATTRECONCILE_ON, rc=rc)
```

At this point, the driver of the model run will transfer control to the run phase of the first gridded Component. In the run phase of the first gridded Component is typically where the data contained in the Fields is manipulated. For this simple example we will do no actual data manipulation because all we are interested in at this point is the metadata. What we will do is add a nested Attribute package inside the currently existing Attribute package on each Field. We will also change the value of one of the Attributes in the original Attribute package, and remove another of the Attributes from the original Attribute package on each of the Fields. The first thing is to declare variables and get the Component, VM, State, and FieldBundle.

```
type(ESMF_VM)                :: vm
integer                      :: petCount, status, myPet, k
character(ESMF_MAXSTR)       :: name2,value2,convESMF,purpGen,purp2,name3
character(ESMF_MAXSTR),dimension(2) :: attrList
type(ESMF_Field)             :: field
type(ESMF_FieldBundle)       :: fieldbundle
type(ESMF_Grid)              :: grid

rc = ESMF_SUCCESS

convESMF = 'ESMF'
purpGen = 'General'
name2 = 'StandardName'
value2 = 'default_standard_name'
name3 = 'LongName'

purp2 = 'Extended'
attrList(1) = 'Coordinates'
attrList(2) = 'Mask'

call ESMF_GridCompGet(comp, vm=vm, rc=status)
call ESMF_VMGet(vm, petCount=petCount, localPet=myPet, rc=status)

call ESMF_StateGet(exportState, "fieldbundle", fieldbundle, rc=rc)
call ESMF_FieldBundleGet(fieldbundle, grid=grid, rc=rc)
```

At this point we will extract each of the Fields in the FieldBundle in turn and change the value of one Attribute in the original Attribute package, add a nested Attribute package, and delete one other of the Attributes in the original Attribute package. These three changes represent, respectively, a value change and two structural changes to the Attribute hierarchy during run time, which must be reconciled across the VM before the second gridded Component can be allowed to further manipulate the Attribute hierarchy.

```
do k = 1, 10
    call ESMF_FieldBundleGet(fieldbundle, fieldIndex=k, field=field, rc=rc)
    call ESMF_AttributeSet(field, name2, value2, convention=convESMF, &
      purpose=purpGen, rc=status)
    call ESMF_AttributeAdd(field, convention=convESMF, purpose=purp2, &
      attrList=attrList, nestConvention=convESMF, nestPurpose=purpGen, &
      rc=rc)
    call ESMF_AttributeSet(field, name='Coordinates', value='Latlon', &
```

```
      convention=convESMF, purpose=purp2, rc=rc)
    call ESMF_AttributeSet(field, name='Mask', value='Yes', &
      convention=convESMF, purpose=purp2, rc=rc)
    call ESMF_AttributeRemove(field, name=name3, convention=convESMF, &
      purpose=purpGen, rc=status)
enddo
```

At this point, the driver of the model run will transfer control to the run phase of the coupler Component.
In the run phase of the coupler Component we must now ensure that the entire VM again has a consistent view of the Attribute hierarchy. This is different from the communication done in the initialize phase of the model run because the only structural change that has occurred is in the Attribute hierarchy. Therefore an `ESMF_AttributeUpdate()` call can be used at this point to reconcile these changes. It should be noted that the `ESMF_AttributeUpdate()` call will reconcile value changes to the Attribute hierarchy as well as structural changes.
The first thing to do is to retrieve the Component, VM, and States. Then `ESMF_AttributeUpdate()` will be called on the import State to accomplish a VM wide communication. Afterwards, the Attribute hierarchy can be transfered, in a local sense, from the import State to the export State using an `ESMF_AttributeCopy()` call.

```
type(ESMF_VM)              :: vm
integer                    :: myPet

integer, dimension(2)      :: rootList

rc = ESMF_SUCCESS

call ESMF_CplCompGet(comp, vm=vm, rc=rc)

call ESMF_VMGet(vm, localPet=myPet, rc=rc)

call ESMF_StateGet(importState, rc=rc)
call ESMF_StateGet(exportState, rc=rc)

rootList = (/0,1/)
call ESMF_AttributeUpdate(importState, vm, rootList=rootList, rc=rc)

call ESMF_AttributeCopy(importState, exportState, &
  ESMF_COPY_ALIAS, ESMF_ATTTREE_ON, rc=rc)
```

At this point the entire VM has a consistent view of the Attribute hierarchy that was recently modified during *run time* in the first gridded component and the driver of the model run will transfer control to the run phase of the second gridded Component.
In the run phase of the second gridded Component is normally where a user model would again manipulate the data it was given. In this simple example we are only dealing with the metadata, which has already been ensured for consistency across the VM, including the exclusive piece of which is being used in this Component. Therefore we are free to use the metadata as we wish, considering only that any changes we make to it during run time will have to first be reconciled before other parts of the VM can use them. However, this is not our concern at this point because we will now explore the capabilities of `ESMF_AttributeWrite()`.
First we will get the Component and VM. Then we will write out the Attribute hierarchy to an .xml file, after which we will write out the Attribute hierarchy to a more reader friendly tab-delimited format. Both of these write calls will output their respective data into files in the execution directory, in either a .xml or .stdout file.

```
type(ESMF_VM)              :: vm
integer                    :: petCount, status, myPet
character(ESMF_MAXSTR)     :: convESMF,purpGen
```

```
    rc = ESMF_SUCCESS

    call ESMF_GridCompGet(comp, vm=vm, rc=status)
    if (status .ne. ESMF_SUCCESS) return
    call ESMF_VMGet(vm, petCount=petCount, localPet=myPet, rc=status)
    if (status .ne. ESMF_SUCCESS) return

    convESMF = 'ESMF'
    purpGen = 'General'

    if (myPet .eq. 2) then
      call ESMF_AttributeWrite(importState,convESMF,purpGen, &
        attwriteflag=ESMF_ATTWRITE_XML, rc=rc)
      call ESMF_AttributeWrite(importState,convESMF,purpGen,rc=rc)
      if (rc .ne. ESMF_SUCCESS) return
    endif
```

At this point the driver of the model run would normally transfer control to the finalize phase of the first gridded Component. However, there is not much of interest as far as metadata is concerned in this portion of the model run. So with that we will conclude this example.

35.6.5 CIM Attribute packages

This example illustrates the use of the Metafor CIM Attribute packages, supplied by ESMF, to create an Attribute hierarchy on an ESMF object tree. A gridded Component is used together with a State and a realistic Field to create a simple ESMF object tree. CIM Attributes packages are created on the Component and Field, and then the individual Attributes within the packages are populated with values. Finally, all the Attributes are written to a CIM-formatted XML file. For a more comprehensive example, see the ESMF_AttributeCIM system test.

```
      ! Use ESMF framework module
      use ESMF
      implicit none

      ! Local variables
      integer                 :: rc, finalrc, petCount, localPet
      type(ESMF_VM)           :: vm
      type(ESMF_Field)        :: ozone
      type(ESMF_State)        :: exportState
      type(ESMF_GridComp)     :: gridcomp
      character(ESMF_MAXSTR)  :: convCIM, purpComp, purpProp
      character(ESMF_MAXSTR)  :: purpField, purpPlatform
      character(ESMF_MAXSTR)  :: convISO, purpRP, purpCitation
      character(ESMF_MAXSTR), dimension(2)  :: compPropAtt

      ! initialize ESMF
      finalrc = ESMF_SUCCESS
      call ESMF_Initialize(vm=vm, defaultlogfilename="AttributeCIMEx.Log", &
        logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
      if (rc/=ESMF_SUCCESS) goto 10

      ! get the vm
      call ESMF_VMGet(vm, petCount=petCount, localPet=localPet, rc=rc)
      if (rc/=ESMF_SUCCESS) goto 10
```

Create the ESMF objects that will hold the CIM Attributes. These objects include a gridded Component, a State, and a Field. In this example we are constructing empty Fields without an underlying Grid.

```
! Create Component
gridcomp = ESMF_GridCompCreate(name="gridded_component", &
  petList=(/0/), rc=rc)

! Create State
exportState = ESMF_StateCreate(name="exportState",  &
  stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)

! Create Field
ozone = ESMF_FieldEmptyCreate(name='ozone', rc=rc)
```

Now add CIM Attribute packages to the Component and Field. Also, add a CIM Component Properties package, to contain two custom attributes.

```
convCIM = 'CIM 1.5'
purpComp = 'Model Component Simulation Description'
purpProp = 'General Component Properties Description'
purpField = 'Inputs Description'
purpPlatform = 'Platform Description'

convISO = 'ISO 19115'
purpRP = 'Responsible Party Description'
purpCitation = 'Citation Description'

! Add CIM Attribute package to the gridded Component
call ESMF_AttributeAdd(gridcomp, convention=convCIM, &
  purpose=purpComp, rc=rc)

! Specify the gridded Component to have a Component Properties
! package with two custom attributes, with user-specified names
compPropAtt(1) = 'SimulationType'
compPropAtt(2) = 'SimulationURL'
call ESMF_AttributeAdd(gridcomp, convention=convCIM, purpose=purpProp, &
  attrList=compPropAtt, rc=rc)

! Add CIM Attribute package to the Field
call ESMF_AttributeAdd(ozone, convention=convCIM, purpose=purpField, &
  rc=rc)
```

The standard Attribute package supplied by ESMF for a CIM Component contains several Attributes, grouped into sub-packages. These Attributes conform to the CIM convention as defined by Metafor and their values are set individually.

```
!
! Top-level model component attributes, set on gridded component
!
call ESMF_AttributeSet(gridcomp, 'ShortName', 'EarthSys_Atmos', &
  convention=convCIM, purpose=purpComp, rc=rc)
call ESMF_AttributeSet(gridcomp, 'LongName', &
  'Earth System High Resolution Global Atmosphere Model', &
  convention=convCIM, purpose=purpComp, rc=rc)
call ESMF_AttributeSet(gridcomp, '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.', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'Version', '2.0', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'ReleaseDate', '2009-01-01T00:00:00Z', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'ModelType', 'aerosol', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'URL', &
    'www.earthsys.org', convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MetadataVersion', '1.1', &
    convention=convCIM, purpose=purpComp, rc=rc)

  ! Simulation run attributes
  call ESMF_AttributeSet(gridcomp, 'SimulationShortName', &
                                   'SMS.f09_g16.X.hector', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'SimulationLongName', &
    'EarthSys - Earth System Modeling Framework Earth System Model 1.0', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'SimulationRationale', &
'EarthSys-ESMF simulation run in repsect to CMIP5 core experiment 1.1 ()', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'SimulationStartDate', &
                                   '1960-01-01T00:00:00Z', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'SimulationDuration', 'P10Y', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'SimulationNumberOfProcessingElements', &
                                   '16', &
    convention=convCIM, purpose=purpComp, rc=rc)

  ! Document genealogy
  call ESMF_AttributeSet(gridcomp, 'PreviousVersion', &
                                   'EarthSys1 Atmosphere', &
    convention=convCIM, purpose=purpComp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'PreviousVersionDescription', &
   'Horizontal resolution increased to 1.20 x 0.80 degrees; ' // &
   'Timestep reduced from 30 minutes to 15 minutes.', &
    convention=convCIM, purpose=purpComp, rc=rc)

  ! Platform description attributes
  call ESMF_AttributeSet(gridcomp, 'CompilerName', 'Pathscale', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'CompilerVersion', '3.0', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineName', 'HECToR', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineDescription', &
    'HECToR (Phase 2a) is currently an integrated system known ' // &
    'as Rainier, which includes a scalar MPP XT4 system, a vector ' // &
```

```
    'system known as BlackWidow, and storage systems.', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineSystem', 'Parallel', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineOperatingSystem', 'Unicos', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineVendor', 'Cray Inc', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineInterconnectType', &
                                   'Cray Interconnect', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineMaximumProcessors', '22656', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineCoresPerProcessor', '4', &
    convention=convCIM, purpose=purpPlatform, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'MachineProcessorType', 'AMD X86_64', &
    convention=convCIM, purpose=purpPlatform, rc=rc)

  ! Component Properties: custom attributes
  call ESMF_AttributeSet(gridcomp, 'SimulationType', 'branch', &
    convention=convCIM, purpose=purpProp, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'SimulationURL', &
                                   'http://earthsys.org/simulations', &
    convention=convCIM, purpose=purpProp, rc=rc)
```

Set the attribute values of the Responsible Party sub-package, created above for the gridded Component in the ESMF_AttributeAdd(gridcomp, ...) call.

```
  ! Responsible party attributes (for Principal Investigator)
  call ESMF_AttributeSet(gridcomp, 'Name', 'John Doe', &
    convention=convISO, purpose=purpRP, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'Abbreviation', 'JD', &
    convention=convISO, purpose=purpRP, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'PhysicalAddress', &
      'Department of Meteorology, University of ABC', &
    convention=convISO, purpose=purpRP, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'EmailAddress', &
                                   'john.doe@earthsys.org', &
    convention=convISO, purpose=purpRP, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'ResponsiblePartyRole', 'PI', &
    convention=convISO, purpose=purpRP, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'URL', 'www.earthsys.org', &
    convention=convISO, purpose=purpRP, rc=rc)
```

Set the attribute values of the Citation sub-package, created above for the gridded Component in the ESMF_AttributeAdd(gridcomp, ...) call.

```
  ! Citation attributes
  call ESMF_AttributeSet(gridcomp, 'ShortTitle', 'Doe_2009', &
    convention=convISO, purpose=purpCitation, rc=rc)
  call ESMF_AttributeSet(gridcomp, 'LongTitle', &
   'Doe, J.A.; Norton, A.B.; ' // &
   'Clark, G.H.; Davies, I.J.. 2009 EarthSys: ' // &
   'The Earth System High Resolution Global Atmosphere Model - Model ' // &
```

```
 'description and basic evaluation. Journal of Climate, 15 (2). ' // &
 '1261-1296.', &
  convention=convISO, purpose=purpCitation, rc=rc)
call ESMF_AttributeSet(gridcomp, 'Date', '2010-03-15', &
  convention=convISO, purpose=purpCitation, rc=rc)
call ESMF_AttributeSet(gridcomp, 'PresentationForm', 'Online Refereed', &
  convention=convISO, purpose=purpCitation, rc=rc)
call ESMF_AttributeSet(gridcomp, 'DOI', 'doi:17.1035/2009JCLI4508.1', &
  convention=convISO, purpose=purpCitation, rc=rc)
call ESMF_AttributeSet(gridcomp, 'URL', &
                       'http://www.earthsys.org/publications', &
  convention=convISO, purpose=purpCitation, rc=rc)
```

The standard Attribute package currently supplied by ESMF for CIM Fields contains a standard CF-Extended package nested within it.

```
! ozone CF-Extended Attributes
call ESMF_AttributeSet(ozone, 'ShortName', 'Global_O3_mon', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'StandardName', 'ozone', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'LongName', 'ozone', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'Units', 'unknown', &
 convention=convCIM, purpose=purpField, rc=rc)

! ozone CIM Attributes
call ESMF_AttributeSet(ozone, 'CouplingPurpose', 'Boundary', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'CouplingSource', 'EarthSys_Atmos', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'CouplingTarget', 'EarthSys_AtmosDynCore', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'Description', &
                              'Global Ozone concentration ' // &
                              'monitoring in the atmosphere.', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'SpatialRegriddingMethod', &
                              'Conservative-First-Order', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'SpatialRegriddingDimension', '3D', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'Frequency', '15 Minutes', &
 convention=convCIM, purpose=purpField, rc=rc)
call ESMF_AttributeSet(ozone, 'TimeTransformationType', &
                              'TimeInterpolation', &
 convention=convCIM, purpose=purpField, rc=rc)
```

Adding the Field to the State will automatically link the Attribute hierarchies from the State to the Field

```
! Add the Field directly to the State
call ESMF_StateAdd(exportState, fieldList=(/ozone/), rc=rc)
```

The Attribute link between a Component and a State must be set manually.

```
! Link the State to the gridded Component
call ESMF_AttributeLink(gridcomp, exportState, rc=rc)
```

Write the entire CIM Attribute hierarchy, beginning at the gridded Component (the top), to an XML file formatted to conform to CIM specifications. The CIM output tree structure differs from the internal Attribute hierarchy in that it has all the attributes of the fields within its top-level <modelComponent> record. The filename used, gridded_component.xml, is derived from the name of the gridded Component, given as an input argument in the ESMF_GridCompCreate() call above. The file is written to the examples execution directory.

```
call ESMF_AttributeWrite(gridcomp, convCIM, purpComp, &
  attwriteflag=ESMF_ATTWRITE_XML,rc=rc)
```

35.6.6 Read an XML file-based ESG Attribute package for a Gridded Component

This example shows how to read an ESG Attribute Package for a Gridded Component from an XML file. The XML file contains Attribute values filled-in by the user. The standard ESG Component Attribute Package is supplied with ESMF and is defined in an XSD file, which is used to validate the XML file. See

ESMF_DIR/src/Superstructure/Component/etc/esmf_gridcomp.xml (Attribute Package values) and

ESMF_DIR/src/Superstructure/Component/etc/esmf_comp.xsd (Attribute Package definition).

```
! ESMF Framework module
use ESMF
implicit none

! local variables
type(ESMF_GridComp)    :: gridcomp
character(ESMF_MAXSTR) :: attrvalue
type(ESMF_VM)          :: vm
integer                :: rc, petCount, localPet


! initialize ESMF
call ESMF_Initialize(vm=vm, defaultlogfilename="AttReadGridCompEx.Log", &
              logkindflag=ESMF_LOGKIND_MULTI, rc=rc)


! get the vm
call ESMF_VMGet(vm, petCount=petCount, localPet=localPet, rc=rc)


if (petCount<4) then
  gridcomp = ESMF_GridCompCreate(name="gridcomp", &
    petList=(/0/), rc=rc)
else
  gridcomp = ESMF_GridCompCreate(name="gridcomp", &
    petList=(/0,1,2,3/), rc=rc)
endif


! Read an XML file to populate the ESG Attribute package of a GridComp.
! The file is validated against an internal, ESMF-supplied XSD file
! defining the standard ESG Component Attribute package (see file
```

```
! pathnames above).
call ESMF_AttributeRead(comp=gridcomp, fileName="esmf_gridcomp.xml", &
    rc=rc)


! Get ESG "ComponentShortName" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='ComponentShortName', &
                       value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "ComponentLongName" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='ComponentLongName', &
                       value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "Agency" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='Agency', value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "Institution" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='Institution', value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "Version" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='Version', value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "Author" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='Author', value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "Discipline" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='Discipline', value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)


! Get ESG "PhysicalDomain" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='PhysicalDomain', &
                       value=attrValue, convention='ESG', &
                       purpose='General', rc=rc)


! Get ESG "CodingLanguage" Attribute from a GridComp Test
call ESMF_AttributeGet(gridcomp, name='CodingLanguage', &
                       value=attrValue,  convention='ESG', &
                       purpose='General', rc=rc)


! Get ESG "ModelComponentFramework" Attribute from a GridComp
call ESMF_AttributeGet(gridcomp, name='ModelComponentFramework', &
                       value=attrValue, &
                       convention='ESG', purpose='General', rc=rc)
```

```
call ESMF_GridCompDestroy(gridcomp, rc=rc)


! finalize ESMF framework
call ESMF_Finalize(rc=rc)
```

35.6.7 Read an XML file-based CF Attribute package for a Field

This example shows how to read a CF Attribute Package for a Field from an XML file. The XML file contains Attribute values filled-in by the user. The standard CF Attribute Package is supplied with ESMF and is defined in an XSD file, which is used to validate the XML file. See

ESMF_DIR/src/Infrastructure/Field/etc/esmf_field.xml (Attribute Package values) and

ESMF_DIR/src/Infrastructure/Field/etc/esmf_field.xsd (Attribute Package definition).

```
! ESMF Framework module
use ESMF
implicit none

! local variables
type(ESMF_Field)        :: field
character(ESMF_MAXSTR) :: attrvalue
type(ESMF_VM)           :: vm
integer                 :: rc


! initialize ESMF
call ESMF_Initialize(vm=vm, defaultlogfilename="AttReadFieldEx.Log", &
              logkindflag=ESMF_LOGKIND_MULTI, rc=rc)


! Create a field
field = ESMF_FieldEmptyCreate(name="field", rc=rc)


! Read an XML file to populate the CF Attribute package of a Field.
! The file is validated against an internal, ESMF-supplied XSD file
! defining the standard CF Attribute package (see file pathnames above).
call ESMF_AttributeRead(field=field, fileName="esmf_field.xml", rc=rc)


! Get CF "ShortName" Attribute from a Field
call ESMF_AttributeGet(field, name='ShortName', value=attrValue, &
                       convention='CF', purpose='General', rc=rc)


! Get CF "StandardName" Attribute from a Field
call ESMF_AttributeGet(field, name='StandardName', &
                       value=attrValue, &
                       convention='CF', purpose='Extended', rc=rc)


! Get CF "LongName" Attribute from a Field
call ESMF_AttributeGet(field, name='LongName', value=attrValue, &
                       convention='CF', purpose='General', rc=rc)
```

```
! Get CF "Units" Attribute from a Field
call ESMF_AttributeGet(field, name='Units', value=attrValue, &
                       convention='CF', purpose='General', rc=rc)


call ESMF_FieldDestroy(field, rc=rc)


! finalize ESMF framework
call ESMF_Finalize(rc=rc)
```

35.6.8 Read an XML file-based GridSpec Attribute package for a Grid

This example shows how to read a GridSpec Attribute Package from an XML file. The XML file contains Attribute values filled-in by the user. The standard GridSpec Attribute Package is supplied with ESMF and is defined in an XSD file, which is used to validate the XML file. See

ESMF_DIR/src/Infrastructure/Grid/etc/esmf_grid.xml (Attribute Package values) and

ESMF_DIR/src/Infrastructure/Grid/etc/esmf_grid.xsd (Attribute Package definition)

```
! ESMF Framework module
use ESMF
implicit none

! local variables
type(ESMF_Grid)        :: grid
character(ESMF_MAXSTR) :: attrvalue
type(ESMF_VM)          :: vm
integer                :: rc


! initialize ESMF
call ESMF_Initialize(vm=vm, &
              defaultlogfilename="AttReadGridEx.Log", &
              logkindflag=ESMF_LOGKIND_MULTI, rc=rc)


! Create a grid
grid = ESMF_GridEmptyCreate(rc=rc)


! Read an XML file to populate the GridSpec Attribute package of a Grid.
! The file is validated against an internal, ESMF-supplied XSD file
! defining the standard GridSpec Attribute package (see file pathnames
! above).
call ESMF_AttributeRead(grid=grid, fileName="esmf_grid.xml", rc=rc)


! Get GridSpec "GridType" Attribute from a Grid
call ESMF_AttributeGet(grid, name='GridType', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "CongruentTiles" Attribute from a Grid
call ESMF_AttributeGet(grid, name='CongruentTiles', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)
```

```
! Get GridSpec "NumberOfGridTiles" Attribute from a Grid
call ESMF_AttributeGet(grid, name='NumberOfGridTiles', &
                       value=attrValue, convention='GridSpec', &
                       purpose='General', rc=rc)


! Get GridSpec "DimensionOrder" Attribute from a Grid
call ESMF_AttributeGet(grid, name='DimensionOrder', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "DiscretizationType" Attribute from a Grid
call ESMF_AttributeGet(grid, name='DiscretizationType', &
                       value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "GeometryType" Attribute from a Grid
call ESMF_AttributeGet(grid, name='GeometryType', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "IsConformal" Attribute from a Grid
call ESMF_AttributeGet(grid, name='IsConformal', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "IsRegular" Attribute from a Grid
call ESMF_AttributeGet(grid, name='IsRegular', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "IsUniform" Attribute from a Grid
call ESMF_AttributeGet(grid, name='IsUniform', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "NorthPoleLocation" Attribute from a Grid
call ESMF_AttributeGet(grid, name='NorthPoleLocation', &
                       value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "NumberOfCells" Attribute from a Grid
call ESMF_AttributeGet(grid, name='NumberOfCells', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "NX" Attribute from a Grid
call ESMF_AttributeGet(grid, name='NX', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)


! Get GridSpec "NY" Attribute from a Grid
call ESMF_AttributeGet(grid, name='NY', value=attrValue, &
                       convention='GridSpec', purpose='General', rc=rc)
```

```
! Get GridSpec "HorizontalResolution" Attribute from a Grid
call ESMF_AttributeGet(grid, name='HorizontalResolution', &
                       value=attrValue, convention='GridSpec', &
                       purpose='General', rc=rc)


call ESMF_GridDestroy(grid, rc=rc)


! finalize ESMF framework
call ESMF_Finalize(rc=rc)
```

35.6.9 Read and validate an XML file-based set of user-defined Attributes for a Coupler Component

This example shows how to read and validate, from an XML and XSD file, respectively, a set of user-defined custom Attributes for a Coupler Component. See

ESMF_DIR/src/Superstructure/Component/etc/custom_cplcomp.xml (Attribute values) and

ESMF_DIR/src/Superstructure/Component/etc/custom_cplcomp.xsd (Attribute definitions)

```
! ESMF Framework module
use ESMF
implicit none

! local variables
type(ESMF_CplComp)      :: cplcomp
character(ESMF_MAXSTR) :: attrvalue
type(ESMF_VM)           :: vm
integer                 :: rc, petCount, localPet


! initialize ESMF
call ESMF_Initialize(vm=vm, &
              defaultlogfilename="AttReadCustCplCompEx.Log", &
              logkindflag=ESMF_LOGKIND_MULTI, rc=rc)


! get the vm
call ESMF_VMGet(vm, petCount=petCount, localPet=localPet, rc=rc)


if (petCount<4) then
  cplcomp = ESMF_CplCompCreate(name="cplcomp", &
    petList=(/0/), rc=rc)
else
  cplcomp = ESMF_CplCompCreate(name="cplcomp", &
    petList=(/0,1,2,3/), rc=rc)
endif


! Read an XML file to decorate a Coupler Component with custom,
! user-defined attributes, and validate them against a corresponding
! XSD schema file (see file pathnames above).
call ESMF_AttributeRead(comp=cplcomp, fileName="custom_cplcomp.xml", &
                        schemaFileName="custom_cplcomp.xsd", rc=rc)
```

```
! Get custom "MyAttribute1" from CplComp
call ESMF_AttributeGet(cplcomp, name='MyAttribute1', value=attrValue, &
     rc=rc)


! Get custom "MyAttribute2" from CplComp
call ESMF_AttributeGet(cplcomp, name='MyAttribute2', value=attrValue, &
     rc=rc)


! Get custom "MyAttribute3" from CplComp
call ESMF_AttributeGet(cplcomp, name='MyAttribute3', value=attrValue, &
     rc=rc)


! Get custom "MyAttribute4" from CplComp
call ESMF_AttributeGet(cplcomp, name='MyAttribute4', value=attrValue, &
     rc=rc)


! Get custom "MyAttribute5" from CplComp
call ESMF_AttributeGet(cplcomp, name='MyAttribute5', value=attrValue, &
     rc=rc)


call ESMF_CplCompDestroy(cplcomp, rc=rc)


! finalize ESMF framework
call ESMF_Finalize(rc=rc)
```

35.7 Restrictions and Future Work

35.7.1 Attributes

- Case insensitive Attribute names, conventions, purposes, and values will be enabled in a future release.

35.7.2 Attribute packages

- A future capability may be to automatically create default object Attribute packages upon ESMF object creation.
- The implemention of Grids is still in flux within the CIM. This will affect the final appearance of the Gridspec package in both ESMF and ESG. It is anticipated that an additional CIM Grid Attribute Package will be created.
- A CIM Scientific Property Attribute Package will be added. For CMIP5, hundreds of Scientific Properties have been identified. All of these will be added to ESMF.
- The Attribute packages ISO Responsible Party, ISO Citation, and CIM Platform can only be created automatically within a CIM Main component Attribute package. In a future release, it will be possible to create these within other CIM Attribute packages as required, or as separate, standalone packages.

35.7.3 Attribute hierarchies

- The option of "deep" copies of an Attribute hierarchy will be added.

35.7.4 Attribute import and export

- The CIM XML output in this release validates against the official CIM v1.5 release. CIM development is continuing, with further releases expected. ESMF, in its future releases, will conform to these future CIM releases.
- The CIM XML output format, as described above, is currently ingestable into ESG. However, ESG, like CIM, is in flux with active development. When ESG officially releases, future releases of ESMF will be compatible with it. The goal is for all three of ESMF, ESG, and CIM to be compatible with each other.
- Only the CIM XML format is targeted for ingestion by ESG, not ESMF XML nor tab-delimited.
- CIM Attribute packages can only be output (to CIM XML); they may be inputtable (via XML) in a future release.
- The ESMF grid Attribute package XML output file only contains the nested GridSpec Attributes; Attributes RegDecompX and RegDecompY will be added in a future release.

35.8 Design and Implementation Notes

This section covers Attribute memory deallocation, the use of `ESMF_AttributeGet()`, Attribute package nesting capabilities, issues with Attributes in a distributed environment, and reading/writing of Attributes via XML files. Issues and procedures dealing with Attribute memory deallocation, using `ESMF_AttributeGet()` to retrieve Attribute lists, and nested Attribute package capabilities are discussed to help avoid misuse. The limitations with Attributes in a distributed environment are also discussed, with an outline of the future work to be done in this area.

35.8.1 Attribute memory deallocation

The Attribute class presents a somewhat different paradigm with respect to memory deallocation than other ESMF objects. The `ESMF_AttributeRemove()` call can be issued to remove any Attribute from an ESMF object or an Attribute package on an ESMF object. This call is also enabled to remove entire Attribute packages with one call, which would remove any nested Attribute packages as well. The user is **not** required to remove all Attributes that are used in a model run. The entire Attribute hierarchy will be removed automatically by ESMF, provided the ESMF objects which contain them are properly destroyed.

The decision to remove either an Attribute or an Attribute package is made by calling `ESMF_AttributeRemove()` with the correct optional arguments. If an Attribute which is not associated with any Attribute package should be removed, then the call must be issued without a convention or purpose argument. If an Attribute in an Attribute package is to be removed, then the call should be issued with all three of name, convention, and purpose. Finally, if an entire Attribute package is to be removed the call should be issued with a convention and purpose, but no Attribute name.

35.8.2 Using `ESMF_AttributeGet()` to retrieve Attribute lists

The behavior of the `ESMF_AttributeGet()` routine, when retrieving an Attribute containing a value list, follows a slightly different convention than other similar ESMF routines. This routine requires the input of a Fortran array as a place to store the retrieved values of the Attribute list. If the array that is given is longer that the list of Attribute values, the first part of the array will be filled, leaving the extra space untouched. If, however, the array passed in is shorter than the number of Attribute values, the routine will exit with a return code which is not equal to **ESMF_SUCCESS**. It is suggested that if it is required by the user to use a Fortran array that is longer than the number of Attribute values returned, only the indices of the array which the user desires to be filled with retrieved Attribute values should be passed into the routine.

Similar behavior is exhibited with the `defaultvalueList` argument in the `ESMF_AttributeGet()` routine. The difference here is that if the `valueList` is shorter than the `defaultvalueList` only the appropriate values will be filed in, and the routine will exit without error. Likewise, if the `valueList` is longer than the `defaultvalueList` then the entire `valueList` will be populated with the beginning section of the `defaultvalueList` that is given.

35.8.3 Using Attribute package nesting capabilites

There is a recommended practice to organizing metadata conventions when using nested Attribute packages. The most general Attribute packages should always be added first, followed by the more specific ones. For instance, when adding Attribute packages to a Field, it is recommended that the CF convention be added first, followed by the ESG convention, followed by any additional customized Attribute packages.

At this time there are several ESMF supplied Attribute packages, with a convention of ESMF and a purpose of General. These Attribute packages are generated by calling `ESMF_AttributeAdd()` with the appropriate convention and purpose. The ESMF standard Attribute packages can be customized by nesting a custom Attribute package around them; they can also be modified in other ways but this is not suggested practice at this time.

Another consideration when using nested Attribute packages is to remember that when a nested Attribute package is removed every nested Attribute package below the point of removal will also be removed (like pruning a tree branch). Thus, by removing the ESG Attribute package on a Field, the CF Attribute package contained within it will also be removed.

35.8.4 Attributes in a distributed environment

This section discusses the methods of building a consistent view of the metadata across the VM of a model run. To better explain the ESMF capabilities for ensuring the integrity of Attributes in a distributed environment, a small working vocabulary of ESMF Attributes will be presented. Three types of changes to an Attribute hierarchy need to be specified, these are: 1. **link changes** are structural links created when two separate Attribute hierarchies are linked, 2. **structural changes** are changes which occur when Attributes or Attribute packages are added or removed within a single level of an Attribute hierarchy, and 3. **value changes** occur when the value portion of any single Attribute is modified. These definitions will help to describe how `ESMF_StateReconcile()` and `ESMF_AttributeUpdate()` can be effectively used to ensure a consistent view of the metadata throughout a model run.

The `ESMF_StateReconcile()` call is used to create a consistent view of ESMF objects over the entire VM in the initialization phase of a model run. All Attributes that are attached to an ESMF object contained in the State, i.e. an object that is being reconciled, can also be reconciled. This is done by setting a flag in the `ESMF_StateReconcile()` call, see the State documentation for details. This means that, at the conclusion of `ESMF_StateReconcile()` there is a one-to-one correspondence between Attribute hierarchies and the ESMF objects they represent. This is the only place where link changes in an Attribute hierarchy can be resolved.

The `ESMF_AttributeUpdate()` call can be used any time during the run phase of a model to insure that either structural or value changes made to an Attribute hierarchy on a subset of the VM are consistently represented across the remainder of the VM. At this time, link changes cannot be resolved by `ESMF_AttributeUpdate()` as this would represent a departure from the one-to-one correspondence between the Attribute hierarchy and the ESMF objects it represents.

This call is similar to `ESMF_StateReconcile()` in that it must be called from a location that has a view of the entire VM across which to update the Attribute hierarchy, such as a coupler Component. The main difference is that `ESMF_AttributeUpdate()` operates only on the underlying Attribute hierarchy of the given ESMF object. The Attribute hierarchy may be updated as many times as necessary, this call is much more efficient than `ESMF_StateReconcile()` for this reason.

The specification of a list of PETs that are to be used as the basis for the update is a key feature of this interface. This allows a many-to-many communication, as well as the direct specification of which PETs are to be updated and which are to be used as the "real" values. One caveat with this routine is that upon completion the destination PETs will have all of the missing Attributes from the source PETs, but this is not true the other way around. This basically boils down to the fact that the end product of calling `ESMF_AttributeUpdate()` is *not* the union of the Attributes on both source and destination PETs. This can be achieved, however, by calling `ESMF_AttributeUpdate()` twice, once from source to destination, and then again from destination to source.

35.8.5 Writing Attribute packages to file

The `ESMF_AttributeWrite()` interface is in limited form at the present time, as it can only be used reliably on the ESMF standard Attribute packages. Chances are that it will perform as expected for most Attribute packages, but for now it is only guaranteed for the ESMF standard Attribute packages. This routine is also not yet enabled to handle multi-valued Attributes. One thing to remember when using this interface is that if you are writing an Attribute package that contains nested Attribute packages then all Attribute nested below the top level Attribute package will be written.

35.8.6 Copying Attribute hierarchies

The ability to copy an Attribute hierarchy is limited at this time. The ESMF_AttributeCopy() routine can be used to *locally* copy an Attribute hierarchy between two States or Components. It is important to note that this is a local copy, and no inter-PET communication is carried out. Another thing to note is that when this functionality is based on a reference copy any further changes made to some portions of the original Attribute hierarchy will also affect the new Attribute hierarchy.
There are two flags in the ESMF_AttributeCopy() routine which specify which type of copy is desired. At this point there are only two different varieties of Attribute hierarchy copies available. One of the requires the ESMF_Copy_Flag to be set to ESMF_COPY_VALUE and the ESMF_AttTreeFlag to be set to ESMF_ATTTREE_OFF. This does a copy of only the first level of an Attribute hierarchy, by value.
The second available copy can be applied by setting the ESMF_Copy_Flag to ESMF_COPY_ALIAS and the ESMF_AttTreeFlag to ESMF_ATTTREE_ON. This copy is more of a hybrid approach of reference and value copies. In this case the Attributes which *belong* to the object being copied are actually copied in full (by value), while the Attributes which are linked to the object being copied are referenced by a pointer (by reference). This means that after copying an Attribute hierarchy from ESMF object A to ESMF object B with this approach, the changes made to the lower portion of either A or B's Attribute hierarchy will be reflected on *both* object A and object B.

35.8.7 Reading and writing Attributes from XML files

The Xerces C++ library, v3.1.0 or better, is used to read and write XML files. More specifically, the SAX2 API is currently used, although future releases may also use the DOM API. The Xerces C++ website is http://xerces.apache.org/xerces-c/. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, Xerces". Also please see the section on Attribute I/O, 32.2.

35.9 Object Model

Each Attribute contains a name-value pair in which the value can be any of several numeric, character, and logical types. Each value type is implemented as a vector, and can hold one or several values. The available ESMF Attribute value types include:

- ESMF_TYPEKIND_I4
- ESMF_TYPEKIND_I8
- ESMF_TYPEKIND_R4
- ESMF_TYPEKIND_R8
- ESMF_TYPEKIND_Logical
- EMSF_TYPEKIND_Character

The other members of the Attribute class can be seen in Figure 27 below, which shows a UML representation of the ESMF Attribute class.
In addition to a name, all Attributes within an Attribute package are identified by a convention, purpose, and the ESMF object type with which they are associated. These are additional strings that are initialized as empty until specified.
Also, all Attributes contain three vectors of pointers to other Attributes, which are empty until specified otherwise. These vectors of Attribute pointers hold the Attributes, Attribute packages, and Attribute links. This feature is what allows the Attribute class to self assemble complex structures for representing and organizing the metadata of an ESMF object hierarchy.
For a more detailed view of how Attribute packages and hierarchies are formed, see Figures 28 and 29, respectively.

Attribute
+ attrName : string + tk : ESMFC_Typekind + attrRoot : ESMC_Logical + attrConvention : string + attrPurpose : string + attrObject : string + attrPack : ESMC_Logical + attrPackHead : ESMC_Logical + attrNested : ESMC_Logical + linkChange : ESMC_Logical + structChange: ESMF_Logical + valueChange : ESMC_Logical + attrBase : ESMC_Base* + parent : Attribute * + attrList : vector<Attribute *> + packList : vector<Attribute *> + linkList : vector<Attribute *> + vi : ESMC_I4 + vip : vector<ESMC_I4> + vi : ESMC_I8 + vip : vector<ESMC_I8> + vf : ESMC_R4 + vfp : vector<ESMC_R4> + vd : ESMC_R8 + vdp: vector<esmc_R8> + vb : ESMC_Logical + vbp : vector<ESMC_Logical> + vcp : string + vcpp : vector<string>

Figure 27: The structure of the Attribute class

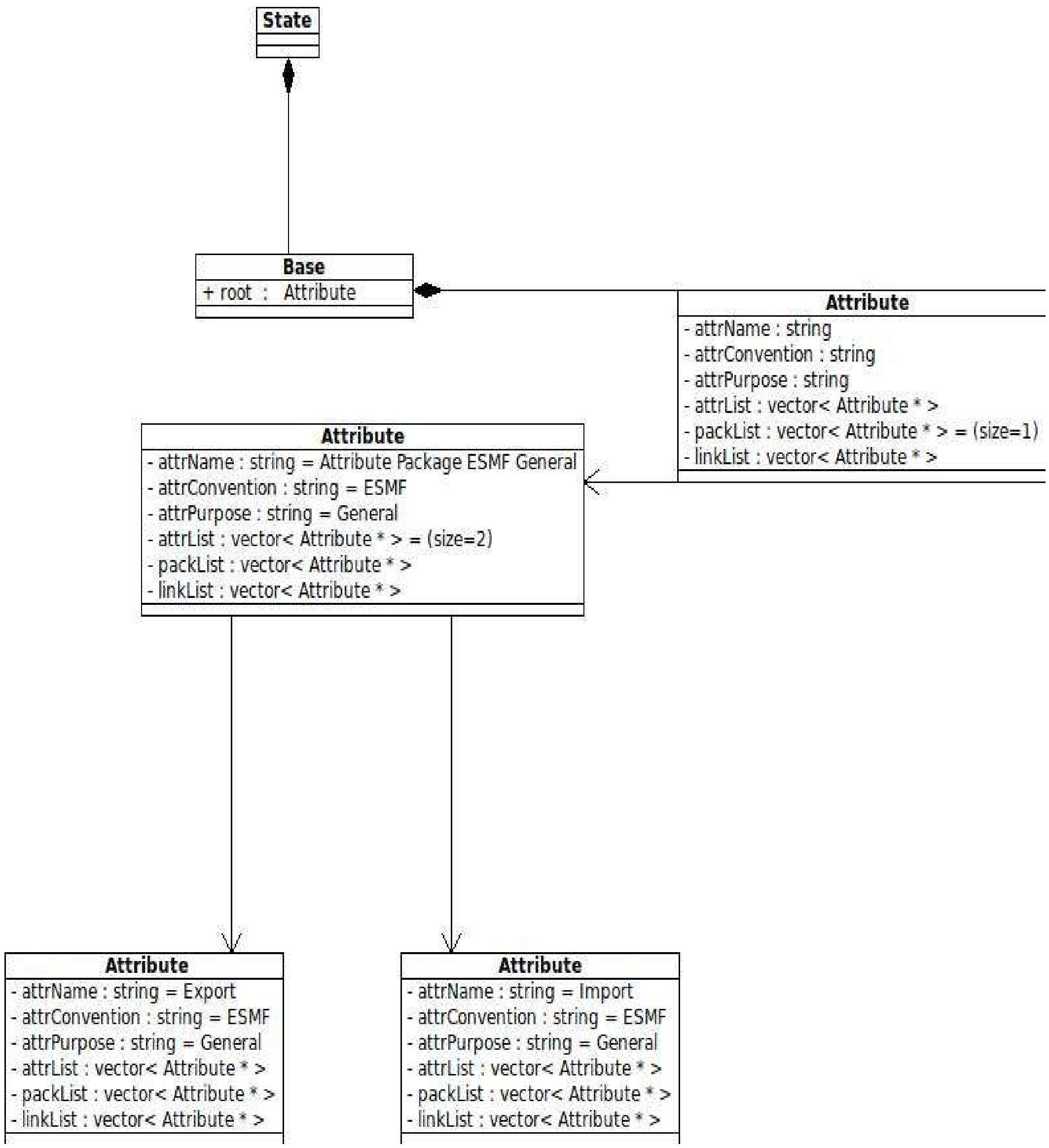


Figure 28: The internal object organization for the representation of Attribute packages

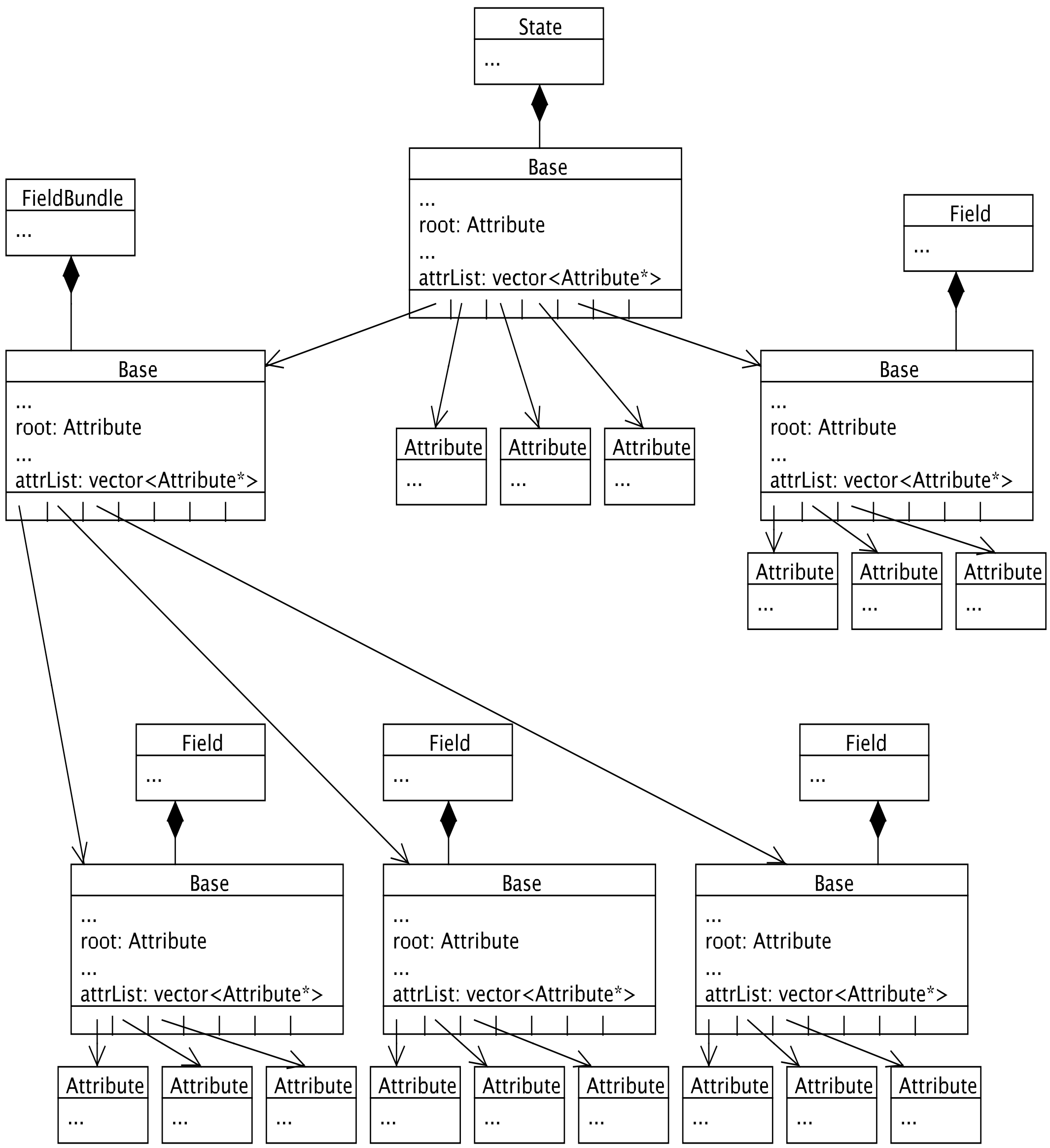


Figure 29: The internal object organization for the representation of Attribute hierarchies

35.10 Class API

35.10.1 ESMF_AttributeAdd - Add an ESMF standard Attribute package

INTERFACE:

```
! Private name; call using ESMF_AttributeAdd()
subroutine ESMF_AttAddPackStd(<object>, convention, purpose, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: convention
character (len = *), intent(in) :: purpose
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add an ESMF standard Attribute package. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_Field), intent(inout) :: field

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An ESMF object

convention The convention of the new Attribute package

purpose The purpose of the new Attribute package

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

35.10.2 ESMF_AttributeAdd - Add an ESMF standard Attribute package, containing nested standard Attribute packages

INTERFACE:

```
! Private name; call using ESMF_AttributeAdd()
subroutine ESMF_AttAddPackStdN(<object>, convention, purpose, &
nestConvention, nestPurpose, nestAttPackInstanceCountList, &
nestAttPackInstanceNameList, nestCount, &
nestAttPackInstanceNameCount,rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: convention
character (len = *), intent(in) :: purpose
character (len = *), intent(in) :: nestConvention(:)
character (len = *), intent(in) :: nestPurpose(:)
integer, intent(in) :: nestAttPackInstanceCountList(:)
character (len = *), intent(out) :: nestAttPackInstanceNameList(:)
integer, intent(in), optional :: nestCount
integer, intent(out), optional :: nestAttPackInstanceNameCount
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add an ESMF standard Attribute package which contains a user-specified number of nested standard Attribute packages. ESMF generates and returns default instance names for the nested Attribute packages. These names can be used later to distinguish among multiple nested Attribute packages of the same type in calls to `ESMF_AttributeGet()`, `ESMF_AttributeSet()`, and `ESMF_AttributeRemove()`. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

The arguments are:

<object> An ESMF object

convention The convention of the new Attribute package

purpose The purpose of the new Attribute package

nestConvention The convention(s) of the standard Attribute package(s) around which to nest the new Attribute package

nestPurpose The purpose(s) of the standard Attribute package(s) around which to nest the new Attribute package

nestAttPackInstanceCountList The desired number of nested Attribute package instances for each nested (nestConvention, nestPurpose) package type. Note: if only one of each nested package type is desired, then the `ESMF_AttributeAdd()` overloaded method `ESMF_AttAddPackStd()` should be used.

nestAttPackInstanceNameList The name(s) of the nested Attribute package instances, generated by ESMF, used to distinguish between multiple instances of the same convention and purpose.

[nestCount] The count of the number of nested Attribute package types to add to the new Attribute package.

[nestAttPackInstanceNameCount] The number of nested Attribute package instance names.

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

35.10.3 ESMF_AttributeAdd - Add a custom Attribute package or modify an existing Attribute package

INTERFACE:

```
! Private name; call using ESMF_AttributeAdd()
subroutine ESMF_AttAddPackCst(<object>, convention, purpose, &
attrList, count, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: convention
character (len = *), intent(in) :: purpose
character (len=*), intent(in) :: attrList(:)
integer, intent(in), optional :: count
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a custom Attribute package to <object>. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An ESMF object

convention The convention of the Attribute package

purpose The purpose of the Attribute package

attrList The list of Attribute names to specify the custom Attribute package

[count] The number of Attributes to add to the custom Attribute package

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

35.10.4 ESMF_AttributeAdd - Add a custom Attribute package with nested Attribute Packages or modify an existing Attribute package

INTERFACE:

```
! Private name; call using ESMF_AttributeAdd()
subroutine ESMF_AttAddPackCstN(<object>, convention, purpose, &
attrList, count, nestConvention, nestPurpose, nestCount, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: convention
character (len = *), intent(in) :: purpose
character (len=*), intent(in), optional :: attrList(:)
integer, intent(in), optional :: count
character (len = *), intent(in) :: nestConvention(:)
character (len = *), intent(in) :: nestPurpose(:)
integer, intent(in), optional :: nestCount
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a custom Attribute package, with one or more nested Attribute packages, to <object>. Allows for building full multiple-child Attribute hierarchies (multi-child trees). See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An ESMF object

convention The convention of the Attribute package

purpose The purpose of the Attribute package

[attrList] The list of Attribute names to specify the custom Attribute package

[count] The number of Attributes to add to the custom Attribute package

nestConvention The convention(s) of the Attribute package(s) around which to nest the new Attribute package

nestPurpose The purpose(s) of the Attribute package(s) around which to nest the new Attribute package

[nestCount] The number of nested Attribute packages to add to the custom Attribute package

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

35.10.5 ESMF_AttributeAdd - Add a custom Attribute package with a single nested Atrribute package, or modify an existing Attribute package

INTERFACE:

```
! Private name; call using ESMF_AttributeAdd()
subroutine ESMF_AttAddPackCstN1(<object>, convention, purpose, &
attrList, count, nestConvention, nestPurpose, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len=*), intent(in) :: convention
character (len=*), intent(in) :: purpose
character (len=*), intent(in), optional :: attrList(:)
integer, intent(in), optional :: count
character (len=*), intent(in) :: nestConvention
character (len=*), intent(in) :: nestPurpose
integer, intent(out), optional :: rc
```

DESCRIPTION:

Add a custom Attribute package, with a single nested Attribute package, to <object>. Allows for building single-child Attribute hierarchies (single-child trees). See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An `ESMF` object

convention The convention of the Attribute package

purpose The purpose of the Attribute package

[attrList] The list of Attribute names to specify the custom Attribute package

[count] The number of Attributes to add to the custom Attribute package

nestConvention The convention of the Attribute package around which to nest the new Attribute package

nestPurpose The purpose of the Attribute package around which to nest the new Attribute package

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

35.10.6 ESMF_AttributeCopy - Copy an Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeCopy()
subroutine ESMF_AttributeCopy(<object1>, <object2>, copyflag, &
attreeflag, rc)
```

ARGUMENTS:

```
<object1>, see below for supported values
<object2>, see below for supported values
type(ESMF_Copy_Flag), intent(in) :: copyflag
type(ESMF_AttTreeFlag), intent(in) :: atttreeflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Copy an Attribute hierarchy from <object1> to <object2>. Supported values for <object1> are:

type(ESMF_CplComp), intent(inout) :: comp1

type(ESMF_GridComp), intent(inout) :: comp1

type(ESMF_State), intent(inout) :: state

Supported values for <object2> are:

type(ESMF_CplComp), intent(inout) :: comp2

type(ESMF_GridComp), intent(inout) :: comp2

type(ESMF_State), intent(inout) :: state

NOTE: Copies between different ESMF objects are not possible at this time.
The arguments are:

<object1> An ESMF object

<object2> An ESMF object

copyflag A flag to determine if the copy is to be by reference, value, or both. This flag is documented in section 9.11.

atttreeflag A flag to determine if the copy is supposed to descend the Attribute hierarchy. This flag is documented in section 35.5.2.

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

35.10.7 ESMF_AttributeGet - Get an Attribute

INTERFACE:

```
subroutine ESMF_AttributeGet(<object>, name, <value argument>, &
<defaultvalue argument>, convention, purpose, &
attPackInstanceName, isPresent, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: name
<value argument>, see below for supported values
<defaultvalue argument>, see below for supported values
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
character (len = *), intent(in), optional :: attPackInstanceName
logical, intent(out), optional :: isPresent
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return an Attribute `value` from the <object>, or from an Attribute package on the <object>, specified by `convention` and `purpose`, and optionally `attPackInstanceName`. A `defaultvalue` argument may be given if a return code is not desired when the Attribute is not found. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

Supported values for <value argument> are:

integer(ESMF_KIND_I4), intent(out) :: value

integer(ESMF_KIND_I8), intent(out) :: value

real (ESMF_KIND_R4), intent(out) :: value

real (ESMF_KIND_R8), intent(out) :: value

logical, intent(out) :: value

character (len = *), intent(out), value

Supported values for <defaultvalue argument> are:

integer(ESMF_KIND_I4), intent(out), optional :: defaultvalue

integer(ESMF_KIND_I8), intent(out), optional :: defaultvalue

real (ESMF_KIND_R4), intent(out), optional :: defaultvalue

real (ESMF_KIND_R8), intent(out), optional :: defaultvalue

logical, intent(out), optional :: defaultvalue

character (len = *), intent(out), optional :: defaultvalue

The arguments are:

<object> An ESMF object

name The name of the Attribute to retrieve

<value argument> The value of the named Attribute

[<defaultvalue argument>] The default value of the named Attribute

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attPackInstanceName] The name of an Attribute package instance, specifying which one of multiple Attribute package instances of the same convention and purpose, within a nest. If not specified, defaults to the first instance.

[isPresent] A logical flag to tell if this Attribute is present or not

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

35.10.8 ESMF_AttributeGet - Get an Attribute

INTERFACE:

```
subroutine ESMF_AttributeGet(<object>, name, <valueList argument>, &
<defaultvalueList argument>, convention, purpose, &
attPackInstanceName, itemCount, isPresent, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: name
<valueList argument>, see below for supported values
<defaultvalueList argument>, see below for supported values
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
character (len = *), intent(in), optional :: attPackInstanceName
integer, intent(out), optional :: itemCount
logical, intent(out), optional :: isPresent
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return an Attribute valueList from the <object>, or from an Attribute package on the <object>, specified by convention and purpose, and optionally attPackInstanceName. A defaultvalueList list argument may be given if a return code is not desired when the Attribute is not found. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

Supported values for <value argument> are:

integer(ESMF_KIND_I4), intent(out) :: valueList(:)

integer(ESMF_KIND_I8), intent(out) :: valueList(:)

real (ESMF_KIND_R4), intent(out) :: valueList(:)

real (ESMF_KIND_R8), intent(out) :: valueList(:)

logical, intent(out) :: valueList(:)

character (len = *), intent(out) :: valueList(:)

Supported values for <defaultvalue argument> are:

integer(ESMF_KIND_I4), intent(out), optional :: defaultvalueList(:)

integer(ESMF_KIND_I8), intent(out), optional :: defaultvalueList(:)

real (ESMF_KIND_R4), intent(out), optional :: defaultvalueList(:)

real (ESMF_KIND_R8), intent(out), optional :: defaultvalueList(:)

logical, intent(out), optional :: defaultvalueList(:)

character (len = *), intent(out), optional :: defaultvalueList(:)

The arguments are:

<object> An `ESMF` object

name The name of the Attribute to retrieve

<valueList argument> The valueList of the named Attribute

[<defaultvalueList argument>] The default value list of the named Attribute

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attPackInstanceName] The name of an Attribute package instance, specifying which one of multiple Attribute package instances of the same convention and purpose, within a nest. If not specified, defaults to the first instance. (Not implemented yet)

[itemCount] The number of items in a multi-valued Attribute

[isPresent] A logical flag to tell if this Attribute is present or not

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

35.10.9 ESMF_AttributeGet - Get the Attribute count

INTERFACE:

```
! Private name; call using ESMF_AttributeGet()
subroutine ESMF_AttributeGetCount(<object>, count, attcountflag, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
integer, intent(out) :: count
type(ESMF_AttGetCountFlag), intent(in), optional :: attcountflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Return the Attribute count for <object>. Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An ESMF object

count The Attribute count for <object>

[attcountflag] The flag to specify which attribute count to return, the default is ESMF_ATTGETCOUNT_ATTRIBUTE. This flag is documented in section 35.5.1.

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

35.10.10 ESMF_AttributeGet - Get Attribute info by name

INTERFACE:

```
! Private name; call using ESMF_AttributeGet()
subroutine ESMF_AttributeGetInfoByNam(<object>, name, &
convention, purpose, &
attPackInstanceName, typekind, &
itemCount, isPresent, rc)
```

ARGUMENTS:

```
    <object>, see below for supported values
    character (len = *), intent(in) :: name
  -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    character (len=*), intent(in), optional :: convention
    character (len=*), intent(in), optional :: purpose
    character (len=*), intent(in), optional :: attPackInstanceName
    type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
    integer, intent(out), optional :: itemCount
    logical, intent(out), optional :: isPresent
    integer, intent(out), optional :: rc
```

DESCRIPTION:

Return information associated with the named Attribute, including `typekind` and `itemCount`. Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An `ESMF` object

name The name of the Attribute to query

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attPackInstanceName] The name of an Attribute package instance, specifying which one of multiple Attribute package instances of the same convention and purpose, within a nest. (Not implemented yet)

[typekind] The typekind of the Attribute

[itemCount] The number of items in this Attribute

[isPresent] A logical flag to tell if this Attribute is present or not

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

35.10.11 ESMF_AttributeGet - Get Attribute info by index number

INTERFACE:

```
! Private name; call using ESMF_AttributeGet()
subroutine ESMF_AttributeGetInfoByNum(<object>, attributeIndex, name, &
typekind, itemcount, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
integer, intent(in) :: attributeIndex
character (len = *), intent(out) :: name
type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
integer, intent(out), optional :: itemCount
integer, intent(out), optional :: rc
```

DESCRIPTION:

Returns information associated with the indexed Attribute, including `name`, `typekind` and `itemCount`. Keep in mind that these indexes start from 1, as expected in a Fortran API. Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An ESMF object

attributeIndex The index number of the Attribute to query

name The name of the Attribute

[typekind] The typekind of the Attribute

[itemCount] The number of items in this Attribute

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

35.10.12 ESMF_AttributeGet - Get Attribute package instance names

INTERFACE:

```
! Private name; call using ESMF_AttributeGet()
subroutine ESMF_AttributeGetAPinstNames(<object>, convention, purpose, &
attPackInstanceNameList, attPackInstanceNameCount, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
character (len = *), intent(out) :: attPackInstanceNameList(:)
integer, intent(out) :: attPackInstanceNameCount
integer, intent(out), optional :: rc
```

DESCRIPTION:

Get the Attribute package instance names of the specified convention and purpose. Also get the number of such names. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

The arguments are:

<object> An ESMF object

convention The convention of the Attribute package instances.

purpose The purpose of the Attribute package instances.

attPackInstanceNameList The name(s) of the Attribute package instances of the given convention and purpose.

attPackInstanceNameCount The number of Attribute package instance names.

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

35.10.13 ESMF_AttributeLink - Link a Component Attribute hierarchy to that of a Component or State

INTERFACE:

```
! Private name; call using ESMF_AttributeLink()
subroutine ESMF_CompAttLink(<object1>, <object2>, rc)
```

ARGUMENTS:

```
<object1>, see below for supported values
<object2>, see below for supported values
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach a CplComp or GridComp Attribute hierarchy to the hierarchy of a CplComp, GridComp, or State.
Supported values for the <object1> are:

type(ESMF_CplComp), intent(inout) :: comp1

type(ESMF_GridComp), intent(inout) :: comp1

Supported values for the <object2> are:

type(ESMF_CplComp), intent(inout) :: comp2

type(ESMF_GridComp), intent(inout) :: comp2

type(ESMF_State), intent(inout) :: state

The arguments are:

<object1> The "parent" object in the Attribute hierarchy link

<object2> The "child" object in the Attribute hierarchy link

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

35.10.14 ESMF_AttributeLink - Link a State Attribute hierarchy with the

hierarchy of a an Array, ArrayBundle, Field, FieldBundle, or State

INTERFACE:

```
! Private name; call using ESMF_AttributeLink()
subroutine ESMF_StateAttLink(state, <object>, rc)
```

ARGUMENTS:

```
type(ESMF\_State), intent(inout) :: state
<object>, see below for supported values
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach a `State` Attribute hierarchy to the hierarchy of a `Fieldbundle`, `Field`, or another `State`. Supported values for the <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_State), intent(inout) :: state

The arguments are:

state An `ESMF_State` object

<object> The object with which to link hierarchies

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

35.10.15 ESMF_AttributeLink - Link a FieldBundle and Field Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeLink()
subroutine ESMF_FieldBundleAttLink(fieldbundle, field, rc)
```

ARGUMENTS:

```
type(ESMF\_FieldBundle), intent(inout) :: fieldbundle
type(ESMF\_Field), intent(inout) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach a `FieldBundle` Attribute hierarchy to the hierarchy of a `Field`.
The arguments are:

fieldbundle An `ESMF_FieldBundle` object

field An `ESMF_Field` object

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

35.10.16 ESMF_AttributeLink - Link a Field and Grid Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeLink()
subroutine ESMF_FieldAttLink(field, grid, rc)
```

ARGUMENTS:

```
type(ESMF\_Field), intent(inout) :: field
type(ESMF\_Grid), intent(inout) :: grid
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach a `Field` Attribute hierarchy to the hierarchy of a `Grid`.
The arguments are:

field An `ESMF_Field` object

grid An `ESMF_Grid` object

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

35.10.17 ESMF_AttributeLink - Link an ArrayBundle and Array Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeLink()
subroutine ESMF_ArrayBundleAttLink(arraybundle, array, rc)
```

ARGUMENTS:

```
type(ESMF\_ArrayBundle), intent(inout) :: arraybundle
type(ESMF\_Array), intent(inout) :: array
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach an `ArrayBundle` Attribute hierarchy to the hierarchy of an `Array`.
The arguments are:

arraybundle An `ESMF_ArrayBundle` object

array An `ESMF_Array` object

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

35.10.18 ESMF_AttributeLinkRemove - Unlink a Component Attribute hierarchy from that of a Component or State

INTERFACE:

```
! Private name; call using ESMF_AttributeLinkRemove()
subroutine ESMF_CompAttLinkRemove(<object1>, <object2>, rc)
```

ARGUMENTS:

```
<object1>, see below for supported values
<object2>, see below for supported values
integer, intent(out), optional :: rc
```

DESCRIPTION:

Unattach a `CplComp` or `GridComp` Attribute hierarchy from the hierarchy of a `CplComp`, `GridComp`, or `State`.
Supported values for the <object1> are:

type(ESMF_CplComp), intent(inout) :: comp1

type(ESMF_GridComp), intent(inout) :: comp1

Supported values for the <object2> are:

type(ESMF_CplComp), intent(inout) :: comp2

type(ESMF_GridComp), intent(inout) :: comp2

type(ESMF_State), intent(inout) :: state

The arguments are:

<object1> The “parent” object in the Attribute hierarchy link

<object2> The “child” object in the Attribute hierarchy link

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

35.10.19 ESMF_AttributeLinkRemove - Unlink a State Attribute hierarchy with

the hierarchy of an Array, ArrayBundle, Field, FieldBundle, or State

INTERFACE:

```
! Private name; call using ESMF_AttributeLinkRemove()
subroutine ESMF_StateAttLinkRemove(state, <object>, rc)
```

ARGUMENTS:

```
type(ESMF\_State), intent(inout) :: state
<object>, see below for supported values
integer, intent(out), optional :: rc
```

DESCRIPTION:

Unattach a `State` Attribute hierarchy from the hierarchy of a `Fieldbundle`, `Field`, or another `State`. Supported values for the <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_State), intent(inout) :: state

The arguments are:

state An `ESMF_State` object

<object> The object with which to unlink hierarchies

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

35.10.20 ESMF_AttributeLinkRemove - Unlink a FieldBundle and Field Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeLinkRemove()
subroutine ESMF_FieldBundleAttLinkRemove(fieldbundle, field, rc)
```

ARGUMENTS:

```
type(ESMF\_FieldBundle), intent(inout) :: fieldbundle
type(ESMF\_Field), intent(inout) :: field
integer, intent(out), optional :: rc
```

DESCRIPTION:

Unattach a `FieldBundle` Attribute hierarchy from the hierarchy of a `Field`.
The arguments are:

fieldbundle An `ESMF_FieldBundle` object

field An `ESMF_Field` object

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

35.10.21 ESMF_AttributeLinkRemove - Unlink a Field and Grid Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeLinkRemove()
subroutine ESMF_FieldAttLinkRemove(field, grid, rc)
```

ARGUMENTS:

```
type(ESMF\_Field), intent(inout) :: field
type(ESMF\_Grid), intent(inout) :: grid
integer, intent(out), optional :: rc
```

DESCRIPTION:

Unattach a `Field` Attribute hierarchy from the hierarchy of a `Grid`.
The arguments are:

field An `ESMF_Field` object

grid An `ESMF_Grid` object

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

35.10.22 ESMF_AttributeLinkRemove - Unlink an ArrayBundle and Array Attribute hierarchy

INTERFACE:

```
! Private name; call using ESMF_AttributeLinkRemove()
subroutine ESMF_ArrayBundleAttLinkRemove(arraybundle, array, rc)
```

ARGUMENTS:

```
type(ESMF\_ArrayBundle), intent(inout) :: arraybundle
type(ESMF\_Array), intent(inout) :: array
integer, intent(out), optional :: rc
```

DESCRIPTION:

Unattach an `ArrayBundle` Attribute hierarchy from the hierarchy of an `Array`.
The arguments are:

arraybundle An `ESMF_ArrayBundle` object

array An `ESMF_Array` object

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

35.10.23 ESMF_AttributeRead - Read Attributes from an XML file

INTERFACE:

```
subroutine ESMF_AttributeRead(<object>, fileName, schemaFileName, &
convention, purpose, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len=*), intent(in), optional :: fileName
character (len=*), intent(in), optional :: schemaFileName
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
integer, intent(out), optional :: rc
```

DESCRIPTION:

Read Attributes for <object> from fileName, whose format is XML. schemaFileName format is XSD. If present, the schemaFileName is used to validate the contents of fileName. schemaFileName must be specified for a fileName containing custom, user-defined Attributes. schemaFileName need not be specified for convention and purposes specifying a standard, ESMF-supplied Attribute package. If present, the convention and purpose specify an Attribute package which is used to filter the reading to just those attributes belonging to the Attribute package. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Requires the third party Xerces C++ XML Parser library to be installed, v3.1.0 or better. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, Xerces" and the website "http://xerces.apache.org/xerces-c". Also please see the section on Attribute I/O, 32.2.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array ! not yet implemented

type(ESMF_ArrayBundle), intent(inout) :: arrayBundle ! not yet implemented

type(ESMF_CplComp), intent(inout) :: cplComp

type(ESMF_GridComp), intent(inout) :: gridComp

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle ! not yet implemented

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_DistGrid), intent(inout) :: distGrid ! not yet implemented

The arguments are:

<object> The ESMF object onto which the read Attributes will be placed

[fileName] The name of the XML file to read

[schemaFileName] The name of the XSD file to validate the contents of fileName

[convention] The convention of the Attribute package to read

[purpose] The purpose of the Attribute package to read

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

35.10.24 ESMF_AttributeRemove - Remove an Attribute or Attribute package

INTERFACE:

```
subroutine ESMF_AttributeRemove(<object>, name, convention, purpose, &
attPackInstanceName, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in), optional :: name
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
character (len = *), intent(in), optional :: attPackInstanceName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Remove an Attribute, or Attribute package on <object>. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An `ESMF` object

[name] The name of the Attribute to remove

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attPackInstanceName] The name of an Attribute package instance, specifying which one of multiple Attribute package instances of the same convention and purpose, within a nest. If not specified, defaults to the first instance. (Not implemented yet)

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

NOTE: An entire Attribute package can be removed by specifying `convention` and `purpose` only, without `name`. By specifying `convention`, `purpose`, and `name` an Attribute will be removed from the corresponding Attribute package, if it exists. An Attribute can be removed directly from <object> by specifying `name`, without `convention` and `purpose`.

35.10.25 ESMF_AttributeSet - Set an Attribute

INTERFACE:

```
subroutine ESMF_AttributeSet(<object>, name, <value argument>, &
convention, purpose, attPackInstanceName, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: name
<value argument>, see below for supported values
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
character (len = *), intent(in), optional :: attPackInstanceName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach an Attribute to <object>, or set an Attribute in an Attribute package. The Attribute has a `name` and `value`, and, if in an Attribute package, a `convention` and `purpose`, and optionally an `attPackInstanceName`. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

Supported values for the <value argument> are:

integer(ESMF_KIND_I4), intent(in) :: value

integer(ESMF_KIND_I8), intent(in) :: value

real (ESMF_KIND_R4), intent(in) :: value

real (ESMF_KIND_R8), intent(in) :: value

logical, intent(in) :: value

character (len = *), intent(in), :: value

The arguments are:

<object> An ESMF object

name The name of the Attribute to set

<value argument> The value of the Attribute to set

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attPackInstanceName] The name of an Attribute package instance, specifying which one of multiple Attribute package instances of the same convention and purpose, within a nest. If not specified, defaults to the first instance.

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

35.10.26 ESMF_AttributeSet - Set an Attribute

INTERFACE:

```
subroutine ESMF_AttributeSet(<object>, name, <valueList argument>, &
convention, purpose, attPackInstanceName, &
itemCount, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in) :: name
<valueList argument>, see below for supported values
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
character (len = *), intent(in), optional :: attPackInstanceName
integer, intent(in), optional :: itemCount
integer, intent(out), optional :: rc
```

DESCRIPTION:

Attach an Attribute to <object>, or set an Attribute in an Attribute package. The Attribute has a `name` and a `valueList`, with an `itemCount`, and, if in an Attribute package, a `convention` and `purpose`, and optionally an `attPackInstanceName`. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_DistGrid), intent(inout) :: distgrid

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

Supported values for the <value argument> are:

integer(ESMF_KIND_I4), intent(in) :: valueList(:)

integer(ESMF_KIND_I8), intent(in) :: valueList(:)

real (ESMF_KIND_R4), intent(in) :: valueList(:)

real (ESMF_KIND_R8), intent(in) :: valueList(:)

logical, intent(in) :: valueList(:)

character (len = *), intent(in) :: valueList(:)

The arguments are:

<object> An ESMF object

name The name of the Attribute to set

<valueList argument> The valueList of the Attribute to set

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attPackInstanceName] The name of an Attribute package instance, specifying which one of multiple Attribute package instances of the same convention and purpose, within a nest. If not specified, defaults to the first instance. (Not implemented yet)

[itemCount] The number of items in a multi-valued Attribute

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

35.10.27 ESMF_AttributeUpdate - Update an Attribute hierarchy

INTERFACE:

```
subroutine ESMF_AttributeUpdate(<object>, vm, rootList, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
type(ESMF_VM), intent(in) :: vm
integer, intent(in) :: rootList(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

Update an Attribute hierarchy during runtime. Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp

type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An ESMF object

vm The virtual machine over which this Attribute hierarchy should be updated

rootList The list of “root” PETs that are to be used to update

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

35.10.28 ESMF_AttributeWrite - Write an Attribute package

INTERFACE:

```
subroutine ESMF_AttributeWrite(<object>, convention, purpose, &
attwriteflag, rc)
```

ARGUMENTS:

```
<object>, see below for supported values
character (len = *), intent(in), optional :: convention
character (len = *), intent(in), optional :: purpose
type(ESMF_AttWriteFlag), intent(in), optional :: attwriteflag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Write the Attribute package for <object>. The Attribute package defines the convention, purpose, and object type of the associated Attributes. Either tab-delimited or xml format is acheived by using attwriteflag. Currently, only ESMF/ESG/CF Field Attribute packages can be written in tab-delimited format. See Section 35.2 for a description of Attribute packages and their conventions, purposes, and object types.
For xml output, requires the third party Xerces C++ XML Parser library to be installed, v3.1.0 or better. For more details, see the *ESMF Users Guide*, under the section *Building and Installing the ESMF, Third Party Libraries, Xerces* and the website "http://xerces.apache.org/xerces-c". Also please see the section on Attribute I/O, 32.2.
Note: For an object type of ESMF_GridComp, convention='WaterML', purpose='TimeSeries', and attwriteflag=ESMF_ATTWRITE_XML, an XML file conforming to a hydrologic standard called WaterML will be written. See the following for more information:

"http://his.cuahsi.org/wofws.html"

"http://www.earthsystemcurator.org/projects/waterml.shtml"

An ESMF Use Test Case is available which showcases an example of how to write a WaterML file; please see

"http://esmf.cvs.sourceforge.net/viewvc/esmf/use_test_cases/ESMF_WaterML"

"http://esmf.cvs.sourceforge.net/viewvc/esmf/use_test_cases/README"

Supported values for <object> are:

type(ESMF_Array), intent(inout) :: array

type(ESMF_ArrayBundle), intent(inout) :: arraybundle

type(ESMF_CplComp), intent(inout) :: comp type(ESMF_GridComp), intent(inout) :: comp

type(ESMF_Field), intent(inout) :: field

type(ESMF_FieldBundle), intent(inout) :: fieldbundle

type(ESMF_Grid), intent(inout) :: grid

type(ESMF_State), intent(inout) :: state

The arguments are:

<object> An `ESMF` object

[convention] The convention of the Attribute package

[purpose] The purpose of the Attribute package

[attwriteflag] The flag to specify which format is desired for the write, the default is ESMF_ATTWRITE_TAB. This flag is documented in section 35.5.3.

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

36 Time Manager Utility

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

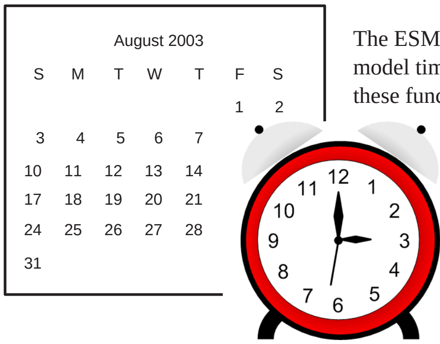
Key Features
Drift-free timekeeping through an integer-based internal time representation. Both integers and reals can be specified at the interface.
The ability to represent time as a rational fraction, to support exact timekeeping in applications that involve grid refinement.
Support for many calendar kinds, including user-customized calendars.
Support for both concurrent and sequential modes of component execution.
Support for varying and negative time steps.

36.1 Time Manager Classes

There are five ESMF classes that represent time concepts:

- **Calendar** A Calendar can be used to keep track of the date as an ESMF Gridded Component advances in time. Standard calendars (such as Gregorian and 360-day) and user-specified calendars are supported. Calendars can be queried for quantities such as seconds per day, days per month, and days per year.
- **Time** A Time represents a time instant in a particular calendar, such as November 28, 1964, at 7:31pm EST in the Gregorian calendar. The Time class can be used to represent the start and stop time of a time integration.
- **TimeInterval** TimeIntervals represent a period of time, such as 300 milliseconds. Time steps can be represented using TimeIntervals.
- **Clock** Clocks collect the parameters and methods used for model time advancement into a convenient package. A Clock can be queried for quantities such as start time, stop time, current time, and time step. Clock methods include incrementing the current time, and determining if it is time to stop.

- **Alarm** Alarms identify unique or periodic events by "ringing" - returning a true value - at specified times. For example, an Alarm might be set to ring on the day of the year when leaves start falling from the trees in a climate model.



The ESMF Time Manager utility includes software to manage model calendars, advance model time, and perform time and date calculations. The software classes that handle these functions are **Times**, **TimeIntervals**, **Clocks**, **Alarms**, and **Calendars**.

In the remainder of this section, we briefly summarize the functionality that the Time Manager classes provide. Detailed descriptions and usage examples precede the API listing for each class.

36.2 Calendar

An ESMF Calendar can be queried for seconds per day, days per month and days per year. The flexible definition of Calendars allows them to be defined for planetary bodies other than Earth. The set of supported calendars includes:

Gregorian The standard Gregorian calendar.

no-leap The Gregorian calendar with no leap years.

Julian The standard Julian date calendar.

Julian Day The standard Julian days calendar.

Modified Julian Day The Modified Julian days calendar.

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

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

See Section 37.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

36.3 Time Instants and TimeIntervals

TimeIntervals and Time instants (simply called Times) are the computational building blocks of the Time Manager utility. TimeIntervals support operations such as add, subtract, compare size, reset value, copy value, and subdivide by a scalar. Times, which are moments in time associated with specific Calendars, can be incremented or decremented by TimeIntervals, compared to determine which of two Times is later, differenced to obtain the TimeInterval between two Times, copied, reset, and manipulated in other useful ways. Times support a host of different queries, both for values of individual Time components such as year, month, day, and second, and for derived values such as day of year, middle of current month and Julian day. It is also possible to retrieve the value of the hardware realtime clock in the form of a Time. See Sections 38.1 and 39.1, respectively, for use and examples of Times and TimeIntervals.
Since climate modeling, numerical weather prediction and other Earth and space applications have widely varying time scales and require different sorts of calendars, Times and TimeIntervals must support a wide range of time specifiers, spanning nanoseconds to years. The interfaces to these time classes are defined so that the user can specify a time using a combination of units selected from the list shown in Table 36.4.

36.4 Clocks and Alarms

Although it is possible to repeatedly step a Time forward by a TimeInterval using arithmetic on these basic types, it is useful to identify a higher-level concept to represent this function. We refer to this capability as a Clock, and include in its required features the ability to store the start and stop times of a model run, to check when time advancement should cease, and to query the value of quantities such as the current time and the time at the previous time step. The Time Manager includes a class with methods that return a true value when a periodic or unique event has taken place; we refer to these as Alarms. Applications may contain temporary or multiple Clocks and Alarms. Sections 40.1 and 41.1 describe the use of Clocks and Alarms in detail.

Table 5: Specifiers for Times and TimeIntervals

<table>
<tr><th>Unit</th><th>Meaning</th></tr>
<tr><td><yy|yy_i8></td><td>Year.</td></tr>
<tr><td>mm</td><td>Month of the year.</td></tr>
<tr><td>dd</td><td>Day of the month.</td></tr>
<tr><td><d|d_i8|d_r8></td><td>Julian or Modified Julian day.</td></tr>
<tr><td><h|h_r8></td><td>Hour.</td></tr>
<tr><td><m|m_r8></td><td>Minute.</td></tr>
<tr><td><s|s_i8|s_r8></td><td>Second.</td></tr>
<tr><td><ms|ms_r8></td><td>Millisecond.</td></tr>
<tr><td><us|us_r8></td><td>Microsecond.</td></tr>
<tr><td><ns|ns_r8></td><td>Nanosecond.</td></tr>
<tr><td>O</td><td>Time zone offset in integer number of hours and minutes.</td></tr>
<tr><td><sN|sN_i8></td><td>Numerator for times of the form $s + \frac{sN}{sD}$, where s is seconds and s, sN, and sD are integers. This format provides a mechanism for supporting exact behavior.</td></tr>
<tr><td><sD|sD_i8</td><td>Denominator for times of the form $s + \frac{sN}{sD}$, where s is seconds and s, sN, and sD are integers.</td></tr>
</table>

36.5 Design and Implementation Notes

1. **Base TimeIntervals and Times on the same integer representation.** It is useful to allow both TimeIntervals and Times to inherit from a single class, BaseTime. In C++, this can be implemented by using inheritance. In Fortran, it can be implemented by having the derived types TimeIntervals and Times contain a derived type BaseTime. In both cases, the BaseTime class can be made private and invisible to the user.

   The result of this strategy is that Time Intervals and Times gain a consistent core representation of time as well a set of basic methods.

   The BaseTime class can be designed with a minimum number of elements to represent any required time. The design is based on the idea used in the real-time POSIX 1003.1b-1993 standard. That is, to represent time simply as a pair of integers: one for seconds (whole) and one for nanoseconds (fractional). These can then be converted at the interface level to any desired format.

   For ESMF, this idea can be modified and extended, in order to handle the requirements for a large time range (> 200,000 years) and to exactly represent any rational fraction, not just nanoseconds. To handle the large time range, a 64-bit or greater integer is used for whole seconds. Any rational fractional second is expressed using two additional integers: a numerator and a denominator. Both the whole seconds and fractional numerator are signed to handle negative time intervals and instants. For arithmetic consistency both must carry the same sign (both positve or both negative), except, of course, for zero values. The fractional seconds element (numerator) is bounded with respect to whole seconds. If the absolute value of the numerator becomes greater than or equal to the denominator, whole seconds are incremented or decremented accordingly and the numerator is reset to the remainder. Conversions are performed upon demand by interface methods within the TimeInterval and Time classes. This is done because different applications require different representations of time intervals and time instances. Floating point values as well as integers can be specified for the various time units in the interfaces, see Table 36.4. Floating point values are represented internally as integer-based rational fractions.

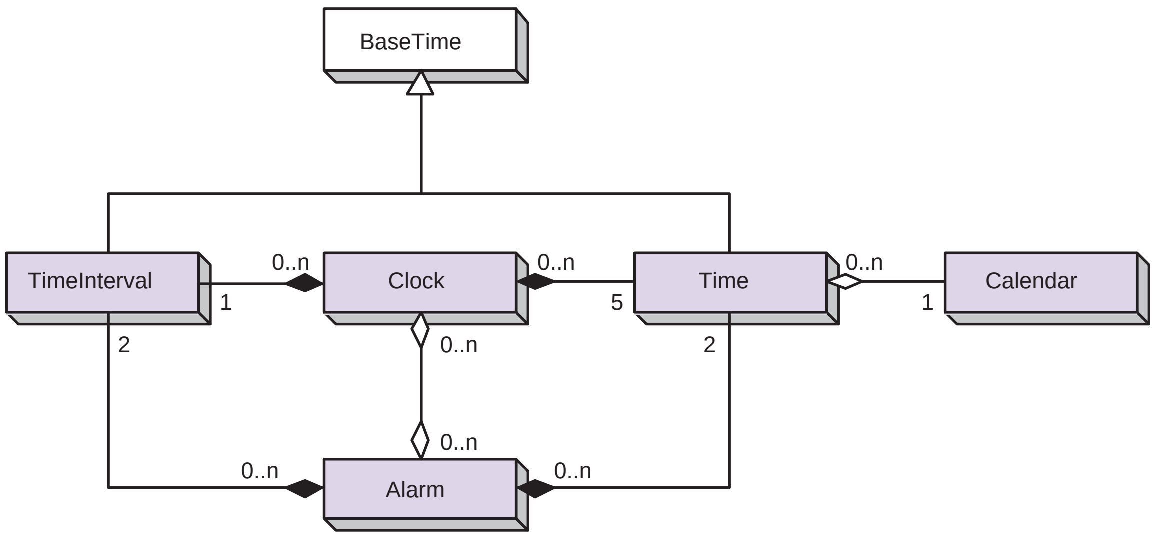
   The BaseTime class defines increment and decrement methods for basic TimeInterval calculations between Time instants. It is done here rather than in the Calendar class because it can be done with simple second-based arithmetic that is calendar independent.

   Comparison methods can also be defined in the BaseTime class. These perform equality/inequality, less than, and greater than comparisons between any two TimeIntervals or Times. These methods capture the common comparison logic between TimeIntervals and Times and hence are defined here for sharing.

2. **The Time class depends on a calendar.** The Time class contains an internal Calendar class. Upon demand by a user, the results of an increment or decrement operation are converted to user units, which may be calendar-dependent, via methods obtained from their internal Calendar.

36.6 Object Model

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



37 Calendar Class

37.1 Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, Julian Day, Modified Julian Day, no-leap, 360-day, and no-calendar. It also supports a user-customized calendar. Brief descriptions are provided for each calendar below. For more information on standard calendars, see [21] and [15].

37.2 Constants

37.2.1 ESMF_CALKIND

DESCRIPTION:
Supported calendar kinds.
The type of this flag is:
`type(ESMF_CalKind_Flag)`
The valid values are:

ESMF_CALKIND_360DAY *Valid range: machine limits*
In the 360-day calendar, there are 12 months, each of which has 30 days. Like the no-leap calendar, this is a simple approximation to the Gregorian calendar sometimes used by modelers.

ESMF_CALKIND_CUSTOM *Valid range: machine limits*
The user can set calendar parameters in the generic calendar.

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

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

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

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

MJD = JD - 2400000.5

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

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

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

37.3 Use and Examples

In most multi-component Earth system applications, the timekeeping in each component must refer to the same standard calendar in order for the components to properly synchronize. It therefore makes sense to create as few ESMF Calendars as possible, preferably one per application. A typical strategy would be to create a single Calendar at the start of an application, and use that Calendar in all subsequent calls that accept a Calendar, such as `ESMF_TimeSet`. The following example shows how to set up an ESMF Calendar.

```
! !PROGRAM: ESMF_CalendarEx - Calendar creation examples
!
! !DESCRIPTION:
!
! This program shows examples of how to create different calendar kinds
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF
      implicit none

      ! instantiate calendars
      type(ESMF_Calendar) :: gregorianCalendar
      type(ESMF_Calendar) :: julianDayCalendar

      ! local variables for Get methods
      integer(ESMF_KIND_I8) :: dl
      type(ESMF_Time) :: time

      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultlogfilename="CalendarEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

37.3.1 Calendar creation

This example shows how to create two `ESMF_Calendars`.

```
      ! create a Gregorian calendar
      gregorianCalendar = ESMF_CalendarCreate(ESMF_CALKIND_GREGORIAN, &
                                              name="Gregorian", rc=rc)


      ! create a Julian Day calendar
      julianDayCalendar = ESMF_CalendarCreate(ESMF_CALKIND_JULIANDAY, &
                                              name="JulianDay", rc=rc)
```

37.3.2 Calendar comparison

This example shows how to compare an `ESMF_Calendar` with a known calendar kind.

```
      ! compare calendar kind against a known type
      if (gregorianCalendar == ESMF_CALKIND_GREGORIAN) then
        print *, "gregorianCalendar is of type ESMF_CALKIND_GREGORIAN."
```

```
else
  print *, "gregorianCalendar is not of type ESMF_CALKIND_GREGORIAN."
end if
```

37.3.3 Time conversion between Calendars

This example shows how to convert a time from one `ESMF_Calendar` to another.

```
call ESMF_TimeSet(time, yy=2004, mm=4, dd=17, &
                  calendar=gregorianCalendar, rc=rc)


! switch time's calendar to perform conversion
call ESMF_TimeSet(time, calendar=julianDayCalendar, rc=rc)


call ESMF_TimeGet(time, d_i8=dl, rc=rc)
print *, "Gregorian date 2004/4/17 is ", dl, &
         " days in the Julian Day calendar."
```

37.3.4 Calendar destruction

This example shows how to destroy two `ESMF_Calendars`.

```
call ESMF_CalendarDestroy(julianDayCalendar, rc=rc)


call ESMF_CalendarDestroy(gregorianCalendar, rc=rc)


! finalize ESMF framework
call ESMF_Finalize(rc=rc)


end program ESMF_CalendarEx
```

37.4 Restrictions and Future Work

1. **Months per year set to 12.** Due to the requirement of only Earth modeling, the number of months per year is hard-coded at 12. However, for easy modification, this is implemented via a C preprocessor #define MONTHS_PER_YEAR in ESMCI_Calendar.h.

37.5 Class API

37.5.1 ESMF_CalendarAssignment(=) - Assign a Calendar to another Calendar

INTERFACE:

```
 interface assignment(=)
 calendar1 = calendar2
```

ARGUMENTS:

```
type(ESMF_Calendar) :: calendar1
type(ESMF_Calendar) :: calendar2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign `calendar1` as an alias to the same `ESMF_Calendar` object in memory as `calendar2`. If `calendar2` is invalid, then `calendar1` will be equally invalid after the assignment.
The arguments are:

calendar1 The `ESMF_Calendar` object on the left hand side of the assignment.

calendar2 The `ESMF_Calendar` object on the right hand side of the assignment.

37.5.2 ESMF_CalendarOperator(==) - Test if Calendar argument 1 is equal to Calendar argument 2

INTERFACE:

```
interface operator(==)
if (<calendar argument 1> == <calendar argument 2>) then ... endif
                            OR
result = (<calendar argument 1> == <calendar argument 2>)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
<calendar argument 1>, see below for supported values
<calendar argument 2>, see below for supported values
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Calendar` class. Compare an `ESMF_Calendar` object or `ESMF_CalKind_Flag` with another calendar object or calendar kind for equality. Return `.true.` if equal, `.false.` otherwise. Comparison is based on calendar kind, which is a property of a calendar object.
If both arguments are `ESMF_Calendar` objects, and both are of type `ESMF_CALKIND_CUSTOM`, then all the calendar's properties, except name, are compared.
If both arguments are `ESMF_Calendar` objects, and either of them is not in the `ESMF_INIT_CREATED` status, an error will be logged. However, this does not affect the return value, which is `.true.` when both arguments are in the *same* status, and `.false.` otherwise.
If one argument is an `ESMF_Calendar` object, and the other is an `ESMF_CalKind_Flag`, and the calendar object is not in the `ESMF_INIT_CREATED` status, an error will be logged and `.false.` will be returned.
Supported values for <calendar argument 1> are:

type(ESMF_Calendar), intent(in) :: calendar1

type(ESMF_CalKind_Flag), intent(in) :: calkindflag1

Supported values for <calendar argument 2> are:

type(ESMF_Calendar), intent(in) :: calendar2

type(ESMF_CalKind_Flag), intent(in) :: calkindflag2

The arguments are:

<calendar argument 1> The `ESMF_Calendar` object or `ESMF_CalKind_Flag` on the left hand side of the equality operation.

<calendar argument 2> The `ESMF_Calendar` object or `ESMF_CalKind_Flag` on the right hand side of the equality operation.

37.5.3 ESMF_CalendarOperator(/=) - Test if Calendar argument 1 is not equal to Calendar argument 2

INTERFACE:

```
interface operator(/=)
if (<calendar argument 1> /= <calendar argument 2>) then ... endif
                             OR
result = (<calendar argument 1> /= <calendar argument 2>)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
<calendar argument 1>, see below for supported values
<calendar argument 2>, see below for supported values
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Calendar` class. Compare a `ESMF_Calendar` object or `ESMF_CalKind_Flag` with another calendar object or calendar kind for inequality. Return `.true.` if not equal, `.false.` otherwise. Comparison is based on calendar kind, which is a property of a calendar object.
If both arguments are `ESMF_Calendar` objects, and both are of type `ESMF_CALKIND_CUSTOM`, then all the calendar's properties, except name, are compared.
If both arguments are `ESMF_Calendar` objects, and either of them is not in the `ESMF_INIT_CREATED` status, an error will be logged. However, this does not affect the return value, which is `.true.` when both arguments are *not* in the *same* status, and `.false.` otherwise.
If one argument is an `ESMF_Calendar` object, and the other is an `ESMF_CalKind_Flag`, and the calendar object is not in the `ESMF_INIT_CREATED` status, an error will be logged and `.true.` will be returned.
Supported values for <calendar argument 1> are:

type(ESMF_Calendar), intent(in) :: calendar1

type(ESMF_CalKind_Flag), intent(in) :: calkindflag1

Supported values for <calendar argument 2> are:

type(ESMF_Calendar), intent(in) :: calendar2

type(ESMF_CalKind_Flag), intent(in) :: calkindflag2

The arguments are:

<calendar argument 1> The `ESMF_Calendar` object or `ESMF_CalKind_Flag` on the left hand side of the non-equality operation.

<calendar argument 2> The `ESMF_Calendar` object or `ESMF_CalKind_Flag` on the right hand side of the non-equality operation.

37.5.4 ESMF_CalendarCreate - Create a new ESMF Calendar of built-in type

INTERFACE:

```
      ! Private name; call using ESMF_CalendarCreate()
      function ESMF_CalendarCreateBuiltIn(calkindflag, &
        name, rc)
```

RETURN VALUE:

```
      type(ESMF_Calendar) :: ESMF_CalendarCreateBuiltIn
```

ARGUMENTS:

```
      type(ESMF_CalKind_Flag), intent(in)            :: calkindflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character (len=*),       intent(in),  optional :: name
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates and sets a `calendar` to the given built-in `ESMF_CalKind_Flag`.
The arguments are:

calkindflag The built-in `ESMF_CalKind_Flag`. Valid values are:
`ESMF_CALKIND_360DAY`,
`ESMF_CALKIND_GREGORIAN`,
`ESMF_CALKIND_JULIAN`,
`ESMF_CALKIND_JULIANDAY`,
`ESMF_CALKIND_MODJULIANDAY`,
`ESMF_CALKIND_NOCALENDAR`,
and `ESMF_CALKIND_NOLEAP`.
See Section 37.2 for a description of each calendar kind.

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

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

37.5.5 ESMF_CalendarCreate - Create a copy of an ESMF Calendar

INTERFACE:

```
      ! Private name; call using ESMF_CalendarCreate()
      function ESMF_CalendarCreateCopy(calendar, rc)
```

RETURN VALUE:

```
      type(ESMF_Calendar) :: ESMF_CalendarCreateCopy
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates a complete (deep) copy of a given `ESMF_Calendar`.
The arguments are:

calendar The `ESMF_Calendar` to copy.

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

37.5.6 ESMF_CalendarCreate - Create a new custom ESMF Calendar

INTERFACE:

```
      ! Private name; call using ESMF_CalendarCreate()
      function ESMF_CalendarCreateCustom(&
        daysPerMonth, secondsPerDay, &
        daysPerYear, daysPerYearDn, daysPerYearDd, name, rc)
```

RETURN VALUE:

```
      type(ESMF_Calendar) :: ESMF_CalendarCreateCustom
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,               intent(in),  optional :: daysPerMonth(:)
      integer(ESMF_KIND_I4), intent(in),  optional :: secondsPerDay
      integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYear   ! not imp
      integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYearDn ! not imp
      integer(ESMF_KIND_I4), intent(in),  optional :: daysPerYearDd ! not imp
      character (len=*),     intent(in),  optional :: name
      integer,               intent(out), optional :: rc
```

DESCRIPTION:

Creates a custom `ESMF_Calendar` and sets its properties.
The arguments are:

[daysPerMonth] Integer array of days per month, for each month of the year. The number of months per year is variable and taken from the size of the array. If unspecified, months per year = 0, with the days array undefined.

[secondsPerDay] Integer number of seconds per day. Defaults to 86400 if not specified.

[daysPerYear] Integer number of days per year. Use with daysPerYearDn and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDn] Integer numerator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear (see above) and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDd] Integer denominator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear and daysPerYearDn (see above) to specify a days-per-year calendar for any planetary body. Default = 1. (Not implemented yet).

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

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

37.5.7 ESMF_CalendarDestroy - Release resources associated with a Calendar

INTERFACE:

```
      subroutine ESMF_CalendarDestroy(calendar, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Releases resources associated with this `ESMF_Calendar`.
The arguments are:

calendar Release resources associated with this `ESMF_Calendar` 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.

37.5.8 ESMF_CalendarGet - Get Calendar properties

INTERFACE:

```
      subroutine ESMF_CalendarGet(calendar, &
        name, calkindflag, daysPerMonth, monthsPerYear, &
        secondsPerDay, secondsPerYear, &
        daysPerYear, daysPerYearDn, daysPerYearDd, rc)
```

ARGUMENTS:

```
      type(ESMF_Calendar),    intent(in)            :: calendar
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_CalKind_Flag),intent(out), optional :: calkindflag
      integer,                intent(out), optional :: daysPerMonth(:)
      integer,                intent(out), optional :: monthsPerYear
```

```
integer(ESMF_KIND_I4),  intent(out), optional :: secondsPerDay
integer(ESMF_KIND_I4),  intent(out), optional :: secondsPerYear
integer(ESMF_KIND_I4),  intent(out), optional :: daysPerYear   !not imp
integer(ESMF_KIND_I4),  intent(out), optional :: daysPerYearDn !not imp
integer(ESMF_KIND_I4),  intent(out), optional :: daysPerYearDd !not imp
character (len=*),      intent(out), optional :: name
integer,                intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets one or more of an `ESMF_Calendar`'s properties.
The arguments are:

calendar The object instance to query.

[calkindflag] The `CalKind_Flag` ESMF_CALKIND_GREGORIAN, ESMF_CALKIND_JULIAN, etc.

[daysPerMonth] Integer array of days per month, for each month of the year.

[monthsPerYear] Integer number of months per year; the size of the daysPerMonth array.

[secondsPerDay] Integer number of seconds per day.

[secondsPerYear] Integer number of seconds per year.

[daysPerYear] Integer number of days per year. For calendars with intercalations, daysPerYear is the number of days for years without an intercalation. For other calendars, it is the number of days in every year. (Not implemented yet).

[daysPerYearDn] Integer fractional number of days per year (numerator). For calendars with intercalations, daysPerYearDn/daysPerYearDd is the average fractional number of days per year (e.g. 25/100 for Julian 4-year intercalation). For other calendars, it is zero. (Not implemented yet).

[daysPerYearDd] Integer fractional number of days per year (denominator). See daysPerYearDn above. (Not implemented yet).

[name] The name of this calendar.

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

37.5.9 ESMF_CalendarIsLeapYear - Determine if given year is a leap year

INTERFACE:

```
! Private name; call using ESMF_CalendarIsLeapYear()
function ESMF_CalendarIsLeapYear<kind>(calendar, yy, rc)
```

RETURN VALUE:

```
logical :: ESMF_CalendarIsLeapYear<kind>
```

ARGUMENTS:

```
       type(ESMF_Calendar),       intent(in)            :: calendar
       integer(ESMF_KIND_<kind>), intent(in)            :: yy
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns `.true.` if the given year is a leap year within the given calendar, and `.false.` otherwise. See also `ESMF_TimeIsLeapYear()`.
The arguments are:

calendar `ESMF_Calendar` to determine leap year within.

yy Year to check for leap year. The type is integer and the <kind> can be either I4 or I8: `ESMF_KIND_I4` or `ESMF_KIND_I8`.

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

37.5.10 ESMF_CalendarPrint - Print the contents of a Calendar

INTERFACE:

```
subroutine ESMF_CalendarPrint(calendar, options, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Prints out an `ESMF_Calendar`'s properties to `stdio`, in support of testing and debugging. The options control the type of information and level of detail.

The arguments are:

calendar `ESMF_Calendar` to be printed out.

[options] Print options. If none specified, prints all calendar property values.
"calkindflag" - print the calendar's type (e.g. ESMF_CALKIND_GREGORIAN).
"daysPerMonth" - print the array of number of days for each month.
"daysPerYear" - print the number of days per year (integer and fractional parts).
"monthsPerYear" - print the number of months per year.
"name" - print the calendar's name.
"secondsPerDay" - print the number of seconds in a day.
"secondsPerYear" - print the number of seconds in a year.

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

37.5.11 ESMF_CalendarSet - Set a Calendar to a built-in type

INTERFACE:

```
      ! Private name; call using ESMF_CalendarSet()
      subroutine ESMF_CalendarSetBuiltIn(calendar, calkindflag, &
        name, rc)
```

ARGUMENTS:

```
      type(ESMF_Calendar),     intent(inout)         :: calendar
      type(ESMF_CalKind_Flag), intent(in)            :: calkindflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character (len=*),       intent(in),  optional :: name
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets `calendar` to the given built-in `ESMF_CalKind_Flag`.
The arguments are:

calendar The object instance to initialize.

calkindflag The built-in `CalKind_Flag`. Valid values are:
`ESMF_CALKIND_360DAY`,
`ESMF_CALKIND_GREGORIAN`,
`ESMF_CALKIND_JULIAN`,
`ESMF_CALKIND_JULIANDAY`,
`ESMF_CALKIND_MODJULIANDAY`,
`ESMF_CALKIND_NOCALENDAR`,
and `ESMF_CALKIND_NOLEAP`.
See Section 37.2 for a description of each calendar kind.

[name] The new name for this calendar.

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

37.5.12 ESMF_CalendarSet - Set properties of a custom Calendar

INTERFACE:

```
      ! Private name; call using ESMF_CalendarSet()
      subroutine ESMF_CalendarSetCustom(calendar, &
        daysPerMonth, secondsPerDay, &
        daysPerYear, daysPerYearDn, daysPerYearDd, name, rc)
```

ARGUMENTS:

```
      type(ESMF_Calendar),  intent(inout)         :: calendar
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,              intent(in),  optional :: daysPerMonth(:)
```

```
integer(ESMF_KIND_I4),intent(in),  optional :: secondsPerDay
integer(ESMF_KIND_I4),intent(in),  optional :: daysPerYear   !not imp
integer(ESMF_KIND_I4),intent(in),  optional :: daysPerYearDn !not imp
integer(ESMF_KIND_I4),intent(in),  optional :: daysPerYearDd !not imp
character (len=*),    intent(in),  optional :: name
integer,              intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets properties in a custom `ESMF_Calendar`.
The arguments are:

calendar The object instance to initialize.

[daysPerMonth] Integer array of days per month, for each month of the year. The number of months per year is variable and taken from the size of the array. If unspecified, months per year = 0, with the days array undefined.

[secondsPerDay] Integer number of seconds per day. Defaults to 86400 if not specified.

[daysPerYear] Integer number of days per year. Use with daysPerYearDn and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDn] Integer numerator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear (see above) and daysPerYearDd (see below) to specify a days-per-year calendar for any planetary body. Default = 0. (Not implemented yet).

[daysPerYearDd] Integer denominator portion of fractional number of days per year (daysPerYearDn/daysPerYearDd). Use with daysPerYear and daysPerYearDn (see above) to specify a days-per-year calendar for any planetary body. Default = 1. (Not implemented yet).

[name] The new name for this calendar.

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

37.5.13 ESMF_CalendarSetDefault - Set the default Calendar kind

INTERFACE:

```
! Private name; call using ESMF_CalendarSetDefault()
subroutine ESMF_CalendarSetDefaultKind(calkindflag, rc)
```

ARGUMENTS:

```
type(ESMF_CalKind_Flag), intent(in)            :: calkindflag
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Sets the default `calendar` to the given type. Subsequent Time Manager operations requiring a calendar where one isn't specified will use the internal calendar of this type.
The arguments are:

calkindflag The calendar kind to be the default.

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

37.5.14 ESMF_CalendarSetDefault - Set the default Calendar

INTERFACE:

```
! Private name; call using ESMF_CalendarSetDefault()
subroutine ESMF_CalendarSetDefaultCal(calendar, rc)
```

ARGUMENTS:

```
type(ESMF_Calendar),     intent(in)            :: calendar
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Sets the default `calendar` to the one given. Subsequent Time Manager operations requiring a calendar where one isn't specified will use this calendar.
The arguments are:

calendar The object instance to be the default.

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

37.5.15 ESMF_CalendarValidate - Validate a Calendar's properties

INTERFACE:

```
subroutine ESMF_CalendarValidate(calendar, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Checks whether a `calendar` is valid. Must be one of the defined calendar kinds. daysPerMonth, daysPerYear, secondsPerDay must all be greater than or equal to zero.
The arguments are:

calendar `ESMF_Calendar` to be validated.

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

38 Time Class

38.1 Description

A Time represents a specific point in time. In order to accommodate the range of time scales in Earth system applications, Times in the ESMF an be specified in many different ways, from years to nanoseconds. The Time interface is designed so that you select one or more options from a list of time units in order to specify a Time. The options for specifying a Time are shown in Table 36.4.

There are Time methods defined for setting and getting a Time, incrementing and decrementing a Time by a TimeInterval, taking the difference between two Times, and comparing Times. Special quantities such as the middle of the month and the day of the year associated with a particular Time can be retrieved. There is a method for returning the Time value as a string in the ISO 8601 format YYYY-MM-DDThh:mm:ss [12].

A Time that is specified in hours, minutes, seconds, or subsecond intervals does not need to be associated with a standard calendar; a Time whose specification includes time units of a day and greater must be. The ESMF representation of a calendar, the Calendar class, is described in Section 37.1. The `ESMF_TimeSet` method is used to initialize a Time as well as associate it with a Calendar. If a Time method is invoked in which a Calendar is necessary and one has not been set, the ESMF method will return an error condition.

In the ESMF the TimeInterval class is used to represent time periods. This class is frequently used in combination with the Time class. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval.

38.2 Use and Examples

Times are most frequently used to represent start, stop, and current model times. The following examples show how to create, initialize, and manipulate `Time`.

```
! !PROGRAM: ESMF_TimeEx - Time initialization and manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of Time initialization and manipulation
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF
      implicit none

      ! instantiate two times
      type(ESMF_Time) :: time1, time2

      ! instantiate a time interval
      type(ESMF_TimeInterval) :: timeinterval1

      ! local variables for Get methods
      integer :: YY, MM, DD, H, M, S

      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultCalKind=ESMF_CALKIND_GREGORIAN, &
        defaultlogfilename="TimeEx.Log", &
        logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

38.2.1 Time initialization

This example shows how to initialize an `ESMF_Time`.

```
! initialize time1 to 2/28/2000 2:24:45
call ESMF_TimeSet(time1, yy=2000, mm=2, dd=28, h=2, m=24, s=45, rc=rc)


print *, "Time1 = "
call ESMF_TimePrint(time1, options="string", rc=rc)
```

38.2.2 Time increment

This example shows how to increment an `ESMF_Time` by an `ESMF_TimeInterval`.

```
! initialize a time interval to 2 days, 8 hours, 36 minutes, 15 seconds
call ESMF_TimeIntervalSet(timeinterval1, d=2, h=8, m=36, s=15, rc=rc)


print *, "Timeinterval1 = "
call ESMF_TimeIntervalPrint(timeinterval1, options="string", rc=rc)


! increment time1 with timeinterval1
time2 = time1 + timeinterval1

call ESMF_TimeGet(time2, yy=YY, mm=MM, dd=DD, h=H, m=M, s=S, rc=rc)
print *, "time2 = time1 + timeinterval1 = ", YY, "/", MM, "/", DD, &
         " ",  H, ":", M, ":", S
```

38.2.3 Time comparison

This example shows how to compare two `ESMF_Times`.

```
if (time2 > time1) then
  print *, "time2 is larger than time1"
else
  print *, "time1 is smaller than or equal to time2"
endif

! finalize ESMF framework
call ESMF_Finalize(rc=rc)


end program ESMF_TimeEx
```

38.3 Restrictions and Future Work

1. **Limits on size and resolution of Time.** The limits on the size and resolution of the time representation are based on the 64-bit integer types used. For seconds, a signed 64-bit integer will have a range of +/- 2^{63}-1, or +/- 9,223,372,036,854,775,807. This corresponds to a maximum size of +/- (2^{63}-1)/(86400 * 365.25) or +/- 292,271,023,045 years.

   For fractional seconds, a signed 64-bit integer will handle a resolution of +/- 2^{31}-1, or +/- 9,223,372,036,854,775,807 parts of a second.

38.4 Class API

38.4.1 ESMF_TimeAssignment(=) - Assign a Time to another Time

INTERFACE:

```
interface assignment(=)
time1 = time2
```

ARGUMENTS:

```
type(ESMF_Time) :: time1
type(ESMF_Time) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Set `time1` equal to `time2`. This is the default Fortran assignment, which creates a complete, independent copy of `time2` as `time1`. If `time2` is an invalid `ESMF_Time` object then `time1` will be equally invalid after the assignment.
The arguments are:

time1 The `ESMF_Time` to be set.

time2 The `ESMF_Time` to be copied.

38.4.2 ESMF_TimeOperator(+) - Increment a Time by a TimeInterval

INTERFACE:

```
interface operator(+)
time2 = time1 + timeinterval
```

RETURN VALUE:

```
type(ESMF_Time) :: time2
```

ARGUMENTS:

```
type(ESMF_Time),         intent(in) :: time1
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (+) operator for the `ESMF_Time` class to increment `time1` with `timeinterval` and return the result as an `ESMF_Time`.
The arguments are:

time1 The `ESMF_Time` to increment.

timeinterval The `ESMF_TimeInterval` to add to the given `ESMF_Time`.

38.4.3 ESMF_TimeOperator(-) - Decrement a Time by a TimeInterval

INTERFACE:

```
interface operator(-)
time2 = time1 - timeinterval
```

RETURN VALUE:

```
type(ESMF_Time) :: time2
```

ARGUMENTS:

```
type(ESMF_Time),         intent(in) :: time1
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (-) operator for the `ESMF_Time` class to decrement `time1` with `timeinterval`, and return the result as an `ESMF_Time`.
The arguments are:

time1 The `ESMF_Time` to decrement.

timeinterval The `ESMF_TimeInterval` to subtract from the given `ESMF_Time`.

38.4.4 ESMF_TimeOperator(-) - Return the difference between two Times

INTERFACE:

```
interface operator(-)
time3 = time1 - time2
```

RETURN VALUE:

```
type(ESMF_Time) :: time3
```

ARGUMENTS:

```
type(ESMF_Time),         intent(in) :: time1
type(ESMF_Time),         intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (-) operator for the `ESMF_Time` class to return the difference between `time1` and `time2` as an `ESMF_TimeInterval`. It is assumed that `time1` is later than `time2`; if not, the resulting `ESMF_TimeInterval` will have a negative value.
The arguments are:

time1 The first `ESMF_Time` in comparison.

time2 The second `ESMF_Time` in comparison.

38.4.5 ESMF_TimeOperator(==) - Test if Time 1 is equal to Time 2

INTERFACE:

```
interface operator(==)
if (time1 == time2) then ... endif
             OR
result = (time1 == time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Time` class to return `.true.` if `time1` and `time2` represent the same instant in time, and `.false.` otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

38.4.6 ESMF_TimeOperator(/=) - Test if Time 1 is not equal to Time 2

INTERFACE:

```
interface operator(/=)
if (time1 /= time2) then ... endif
             OR
result = (time1 /= time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Time` class to return `.true.` if `time1` and `time2` do not represent the same instant in time, and `.false.` otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

38.4.7 ESMF_TimeOperator(<) - Test if Time 1 is less than Time 2

INTERFACE:

```
interface operator(<)
if (time1 < time2) then ... endif
             OR
result = (time1 < time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (<) operator for the `ESMF_Time` class to return `.true.` if `time1` is earlier in time than `time2`, and `.false.` otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

38.4.8 ESMF_TimeOperator(<=) - Test if Time 1 is less than or equal to Time 2

INTERFACE:

```
interface operator(<=)
if (time1 <= time2) then ... endif
             OR
result = (time1 <= time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (<=) operator for the `ESMF_Time` class to return `.true.` if `time1` is earlier in time or the same time as `time2`, and `.false.` otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

38.4.9 ESMF_TimeOperator(>) - Test if Time 1 is greater than Time 2

INTERFACE:

```
interface operator(>)
if (time1 > time2) then ... endif
             OR
result = (time1 > time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (>) operator for the `ESMF_Time` class to return `.true.` if `time1` is later in time than `time2`, and `.false.` otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

38.4.10 ESMF_TimeOperator(>=) - Test if Time 1 is greater than or equal to Time 2

INTERFACE:

```
interface operator(>=)
if (time1 >= time2) then ... endif
             OR
result = (time1 >= time2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Time), intent(in) :: time1
type(ESMF_Time), intent(in) :: time2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (>=) operator for the `ESMF_Time` class to return `.true.` if `time1` is later in time or the same time as `time2`, and `.false.` otherwise.
The arguments are:

time1 First `ESMF_Time` in comparison.

time2 Second `ESMF_Time` in comparison.

38.4.11 ESMF_TimeGet - Get a Time value

INTERFACE:

```
      subroutine ESMF_TimeGet(time, &
        yy, yy_i8, &
        mm, dd, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, &
        calendar, calkindflag, timeZone, &
        timeString, timeStringISOFrac, &
        dayOfWeek, midMonth, &
        dayOfYear,  dayOfYear_r8, &
        dayOfYear_intvl, rc)
```

ARGUMENTS:

```
      type(ESMF_Time),         intent(in)            :: time
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(out), optional :: yy
      integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
      integer,                 intent(out), optional :: mm
      integer,                 intent(out), optional :: dd
      integer(ESMF_KIND_I4),   intent(out), optional :: d
      integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: h
      integer(ESMF_KIND_I4),   intent(out), optional :: m
      integer(ESMF_KIND_I4),   intent(out), optional :: s
      integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: ms
      integer(ESMF_KIND_I4),   intent(out), optional :: us
      integer(ESMF_KIND_I4),   intent(out), optional :: ns
      real(ESMF_KIND_R8),      intent(out), optional :: d_r8
      real(ESMF_KIND_R8),      intent(out), optional :: h_r8
      real(ESMF_KIND_R8),      intent(out), optional :: m_r8
      real(ESMF_KIND_R8),      intent(out), optional :: s_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ms_r8
      real(ESMF_KIND_R8),      intent(out), optional :: us_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ns_r8
      integer(ESMF_KIND_I4),   intent(out), optional :: sN
      integer(ESMF_KIND_I8),   intent(out), optional :: sN_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: sD
      integer(ESMF_KIND_I8),   intent(out), optional :: sD_i8
      type(ESMF_Calendar),     intent(out), optional :: calendar
      type(ESMF_CalKind_Flag), intent(out), optional :: calkindflag
      integer,                 intent(out), optional :: timeZone ! not imp
      character (len=*),       intent(out), optional :: timeString
      character (len=*),       intent(out), optional :: timeStringISOFrac
      integer,                 intent(out), optional :: dayOfWeek
      type(ESMF_Time),         intent(out), optional :: midMonth
```

```
      integer(ESMF_KIND_I4),   intent(out), optional :: dayOfYear
      real(ESMF_KIND_R8),      intent(out), optional :: dayOfYear_r8
      type(ESMF_TimeInterval), intent(out), optional :: dayOfYear_intvl
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the value of `time` in units specified by the user via Fortran optional arguments. See `ESMF_TimeSet()` above for a description of time units and calendars.

The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers. For example, if a time value is 5 and 3/8 seconds (s=5, sN=3, sD=8), and you want to get it as floating point seconds, you would get 5.375 (s_r8=5.375).

Units are bound (normalized) by the next larger unit specified. For example, if a time is defined to be 2:00 am on February 2, 2004, then `ESMF_TimeGet(dd=day, h=hours, s=seconds)` would return `day = 2`, `hours = 2`, `seconds = 0`, whereas `ESMF_TimeGet(dd = day, s=seconds)` would return `day = 2`, `seconds = 7200`. Note that `hours` and `seconds` are bound by a day. If bound by a month, `ESMF_TimeGet(mm=month, h=hours, s=seconds)` would return `month = 2`, `hours = 26`, `seconds = 0`, and `ESMF_TimeGet(mm = month, s=seconds)` would return `month = 2`, `seconds = 93600` (26 * 3600). Similarly, if bound to a year, `ESMF_TimeGet(yy=year, h=hours, s=seconds)` would return `year = 2004`, `hours = 770` (32*24 + 2), `seconds = 0`, and `ESMF_TimeGet(yy = year, s=seconds)` would return `year = 2004`, `seconds = 2772000` (770 * 3600).

For `timeString`, `timeStringISOFrac`, `dayOfWeek`, `midMonth`, `dayOfYear`, `dayOfYear_intvl`, and `dayOfYear_r8` described below, valid calendars are Gregorian, Julian, No Leap, 360 Day and Custom calendars. Not valid for Julian Day, Modified Julian Day, or No Calendar.

For `timeString` and `timeStringISOFrac`, YYYY format returns at least 4 digits; years <= 999 are padded on the left with zeroes and years >= 10000 return the number of digits required.

For timeString, convert `ESMF_Time`'s value into partial ISO 8601 format YYYY-MM-DDThh:mm:ss[:n/d]. See [12] and [2]. See also method `ESMF_TimePrint()`.

For timeStringISOFrac, convert `ESMF_Time`'s value into full ISO 8601 format YYYY-MM-DDThh:mm:ss[.f]. See [12] and [2]. See also method `ESMF_TimePrint()`.

For dayOfWeek, gets the day of the week the given `ESMF_Time` instant falls on. ISO 8601 standard: Monday = 1 through Sunday = 7. See [12] and [2].

For midMonth, gets the middle time instant of the month that the given `ESMF_Time` instant falls on.

For dayOfYear, gets the day of the year that the given `ESMF_Time` instant falls on. See range discussion in argument list below. Return as an integer value.

For dayOfYear_r8, gets the day of the year the given `ESMF_Time` instant falls on. See range discussion in argument list below. Return as floating point value; fractional part represents the time of day.

For dayOfYear_intvl, gets the day of the year the given `ESMF_Time` instant falls on. Return as an `ESMF_TimeInterval`.

The arguments are:

time The object instance to query.

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

[yy_i8] Integer year (large, >= 64-bit).

[mm] Integer month.

[dd] Integer day of the month.

[d] Integer Julian, or Modified Julian, days (>= 32-bit).

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

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

[ms] Integer milliseconds.

[us] Integer microseconds.

[ns] Integer nanoseconds.

[d_r8] Double precision days.

[h_r8] Double precision hours.

[m_r8] Double precision minutes.

[s_r8] Double precision seconds.

[ms_r8] Double precision milliseconds.

[us_r8] Double precision microseconds.

[ns_r8] Double precision nanoseconds.

[sN] Integer numerator of fractional seconds (sN/sD).

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[sD] Integer denominator of fractional seconds (sN/sD).

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[calendar] Associated `Calendar`.

[calkindflag] Associated `CalKind_Flag`.

[timeZone] Associated timezone (hours offset from UCT, e.g. EST = -5). (Not implemented yet).

[timeString] Convert time value to format string YYYY-MM-DDThh:mm:ss[:n/d], where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [12] and [2]. See also method `ESMF_TimePrint()`.

[timeStringISOFrac] Convert time value to strict ISO 8601 format string YYYY-MM-DDThh:mm:ss[.f], where f is decimal form of any fractional seconds. See [12] and [2]. See also method `ESMF_TimePrint()`.

[dayOfWeek] The time instant's day of the week [1-7].

[MidMonth] The given time instant's middle-of-the-month time instant.

[dayOfYear] The `ESMF_Time` instant's integer day of the year. [1-366] for Gregorian and Julian calendars, [1-365] for No-Leap calendar. [1-360] for 360-Day calendar. User-defined range for Custom calendar.

[dayOfYear_r8] The `ESMF_Time` instant's floating point day of the year. [1.x-366.x] for Gregorian and Julian calendars, [1.x-365.x] for No-Leap calendar. [1.x-360.x] for 360-Day calendar. User-defined range for Custom calendar.

[dayOfYear_intvl] The `ESMF_Time` instant's day of the year as an `ESMF_TimeInterval`.

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

38.4.12 ESMF_TimeIsLeapYear - Determine if a Time is in a leap year

INTERFACE:

```
      function ESMF_TimeIsLeapYear(time, rc)
```

RETURN VALUE:

```
      logical :: ESMF_TimeIsLeapYear
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns true if given time is in a leap year, and false otherwise. See also `ESMF_CalendarIsLeapYear()`.
The arguments are:

time The `ESMF_Time` to check for leap year.

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

38.4.13 ESMF_TimeIsSameCalendar - Compare Calendars of two Times

INTERFACE:

```
      function ESMF_TimeIsSameCalendar(time1, time2, rc)
```

RETURN VALUE:

```
      logical :: ESMF_TimeIsSameCalendar
```

ARGUMENTS:

```
      type(ESMF_Time), intent(in)            :: time1
      type(ESMF_Time), intent(in)            :: time2
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,         intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns true if the Calendars in these Times are the same, false otherwise.
The arguments are:

time1 The first `ESMF_Time` in comparison.

time2 The second `ESMF_Time` in comparison.

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

38.4.14 ESMF_TimePrint - Print the contents of a Time

INTERFACE:

```
subroutine ESMF_TimePrint(time, options, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Prints out the contents of an `ESMF_Time` to `stdout`, in support of testing and debugging. The options control the type of information and level of detail. For options "string" and "string isofrac", YYYY format returns at least 4 digits; years <= 999 are padded on the left with zeroes and years >= 10000 return the number of digits required.

The arguments are:

time The `ESMF_Time` to be printed out.

[options] Print options. If none specified, prints all Time property values.
"string" - prints `time`'s value in ISO 8601 format for all units through seconds. For any non-zero fractional seconds, prints in integer rational fraction form n/d. Format is YYYY-MM-DDThh:mm:ss[:n/d], where [:n/d] is the integer numerator and denominator of the fractional seconds value, if present. See [12] and [2]. See also method `ESMF_TimeGet(..., timeString= , ...)`
"string isofrac" - prints `time`'s value in strict ISO 8601 format for all units, including any fractional seconds part. Format is YYYY-MM-DDThh:mm:ss[.f] where [.f] represents fractional seconds in decimal form, if present. See [12] and [2]. See also method `ESMF_TimeGet(..., timeStringISOFrac= , ...)`

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

38.4.15 ESMF_TimeSet - Initialize or set a Time

INTERFACE:

```
subroutine ESMF_TimeSet(time, &
  yy, yy_i8, &
  mm, dd, &
  d, d_i8, &
  h, m, &
  s, s_i8, &
  ms, us, ns, &
  d_r8, h_r8, m_r8, s_r8, &
  ms_r8, us_r8, ns_r8, &
  sN, sN_i8, sD, sD_i8, &
  calendar, calkindflag, &
  timeZone, rc)
```

ARGUMENTS:

```
      type(ESMF_Time),         intent(inout)         :: time
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(in),  optional :: yy
      integer(ESMF_KIND_I8),   intent(in),  optional :: yy_i8
      integer,                 intent(in),  optional :: mm
      integer,                 intent(in),  optional :: dd
      integer(ESMF_KIND_I4),   intent(in),  optional :: d
      integer(ESMF_KIND_I8),   intent(in),  optional :: d_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: h
      integer(ESMF_KIND_I4),   intent(in),  optional :: m
      integer(ESMF_KIND_I4),   intent(in),  optional :: s
      integer(ESMF_KIND_I8),   intent(in),  optional :: s_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: ms
      integer(ESMF_KIND_I4),   intent(in),  optional :: us
      integer(ESMF_KIND_I4),   intent(in),  optional :: ns
      real(ESMF_KIND_R8),      intent(in),  optional :: d_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: h_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: m_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: s_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: ms_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: us_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: ns_r8
      integer(ESMF_KIND_I4),   intent(in),  optional :: sN
      integer(ESMF_KIND_I8),   intent(in),  optional :: sN_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: sD
      integer(ESMF_KIND_I8),   intent(in),  optional :: sD_i8
      type(ESMF_Calendar),     intent(in),  optional :: calendar
      type(ESMF_CalKind_Flag), intent(in),  optional :: calkindflag
      integer,                 intent(in),  optional :: timeZone ! not imp
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Initializes an `ESMF_Time` with a set of user-specified units via Fortran optional arguments.
The range of valid values for mm and dd depend on the calendar used. For Gregorian, Julian, and No-Leap calendars, mm is [1-12] and dd is [1-28,29,30, or 31], depending on the value of mm and whether yy or yy_i8 is a leap year. For the 360-day calendar, mm is [1-12] and dd is [1-30]. For the Julian-day, Modified Julian-day, and No-calendar, yy, yy_i8, mm, and dd are invalid inputs, since these calendars do not define them. When valid, the yy and yy_i8 arguments should be fully specified, e.g. 2003 instead of 03. yy and yy_i8 ranges are only limited by machine word size, except for the Gregorian and Julian calendars, where the lowest (proleptic) date limits are 3/1/-4800 and 3/1/-4712, respectively. This is a limitation of the Gregorian date-to-Julian day and Julian date-to-Julian day conversion algorithms used to convert Gregorian and Julian dates to the internal representation of seconds. See [8] for a description of the Gregorian date-to-Julian day algorithm and [11] for a description of the Julian date-to-Julian day algorithm. The Custom calendar will have user-defined values for yy, yy_i8, mm, and dd.
The Julian day specifier, d or d_i8, can only be used with the Julian-day and Modified Julian Day calendars, and has a valid range depending on the word size. For a signed 32-bit d, the range for Julian-day is [+/- 24855]. For a signed 64-bit d or d_i8, the valid range for Julian-day is [+/- 106,751,991,167,300]. The Julian day number system adheres to the conventional standard where the reference day of d=0 corresponds to 11/24/-4713 in the proleptic Gregorian calendar and 1/1/-4712 in the proleptic Julian calendar. See [16] and [1].
The Modified Julian Day, introduced by space scientists in the late 1950's, is defined as Julian-day - 2400000.5. See [23].
Note that d and d_i8 are not valid for the No-Calendar. To remain consistent with non-Earth calendars added to ESMF in the future, ESMF requires a calendar to be planet-specific. Hence the No-Calendar does not know what a day is; it

cannot assume an Earth day of 86400 seconds.
Hours, minutes, seconds, and sub-seconds can be used with any calendar, since they are standardized units that are the same for any planet.
Time manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers. Sub-second values are represented internally with an integer numerator and denominator fraction (sN/sD). The smallest required resolution is nanoseconds (denominator), per Time Manager requirement TMG3.1. For example, pi can be represented as s=3, sN=141592654, sD=1000000000. However, via sN_i8 and sD_i8, larger values can be used. If specifying a constant floating point value, be sure to provide at least 16 digits to take full advantage of double precision, for example s_r8=2.718281828459045d0 for 'e' seconds.
The arguments are:

time The object instance to initialize.

[yy] Integer year (>= 32-bit). Default = 0

[yy_i8] Integer year (large, >= 64-bit). Default = 0

[mm] Integer month. Default = 1

[dd] Integer day of the month. Default = 1

[d] Integer Julian, or Modified Julian, days (>= 32-bit). Default = 0

[d_i8] Integer Julian, or Modified Julian days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0

[us] Integer microseconds. Default = 0

[ns] Integer nanoseconds. Default = 0

[d_r8] Double precision days. Default = 0.0.

[h_r8] Double precision hours. Default = 0.0.

[m_r8] Double precision minutes. Default = 0.0.

[s_r8] Double precision seconds. Default = 0.0.

[ms_r8] Double precision milliseconds. Default = 0.0.

[us_r8] Double precision microseconds. Default = 0.0.

[ns_r8] Double precision nanoseconds. Default = 0.0.

[sN] Integer numerator of fractional seconds (sN/sD). Default = 0

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 0

[sD] Integer denominator of fractional seconds (sN/sD). Default = 1

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 1

calendar Associated `Calendar`. Defaults to calendar `ESMF_CALKIND_NOCALENDAR` or default specified in `ESMF_Initialize()` or `ESMF_CalendarSetDefault()`. Alternate to, and mutually exclusive with, calkindflag below. Primarily for specifying a custom calendar kind.

[calkindflag] Alternate to, and mutually exclusive with, calendar above. More convenient way of specifying a built-in calendar kind.

[timeZone] Associated timezone (hours offset from UTC, e.g. EST = -5). Default = 0 (UTC). (Not implemented yet).

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

38.4.16 ESMF_TimeSyncToRealTime - Get system real time (wall clock time)

INTERFACE:

```
      subroutine ESMF_TimeSyncToRealTime(time, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the system real time (wall clock time), and returns it as an `ESMF_Time`. Accurate to the nearest second.
The arguments are:

time The object instance to receive the real time.

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

38.4.17 ESMF_TimeValidate - Validate a Time

INTERFACE:

```
      subroutine ESMF_TimeValidate(time, options, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Checks whether an `ESMF_Time` is valid. Must be a valid date/time on a valid calendar. The options control the type of validation.
The arguments are:

time `ESMF_Time` instant to be validated.

[options] Validation options. If none specified, validates all `time` property values.
"calendar" - validate only the `time`'s calendar.
"timezone" - validate only the `time`'s timezone.

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

39 TimeInterval Class

39.1 Description

A TimeInterval represents a period between time instants. It can be either positive or negative. Like the Time interface, the TimeInterval interface is designed so that you can choose one or more options from a list of time units in order to specify a TimeInterval. See Section 36.3, Table 36.4 for the available options.

There are TimeInterval methods defined for setting and getting a TimeInterval, for incrementing and decrementing a TimeInterval by another TimeInterval, and for multiplying and dividing TimeIntervals by integers, reals, fractions and other TimeIntervals. Methods are also defined to take the absolute value and negative absolute value of a TimeInterval, and for comparing the length of two TimeIntervals.

The class used to represent time instants in ESMF is Time, and this class is frequently used in operations along with TimeIntervals. For example, the difference between two Times is a TimeInterval.

When a TimeInterval is used in calculations that involve an absolute reference time, such as incrementing a Time with a TimeInterval, calendar dependencies may be introduced. The length of the time period that the TimeInterval represents will depend on the reference Time and the standard calendar that is associated with it. The calendar dependency becomes apparent when, for example, adding a TimeInterval of 1 day to the Time of February 28, 1996, at 4:00pm EST. In a 360 day calendar, the resulting date would be February 29, 1996, at 4:00pm EST. In a no-leap calendar, the result would be March 1, 1996, at 4:00pm EST.

TimeIntervals are used by other parts of the ESMF timekeeping system, such as Clocks (Section 40.1) and Alarms (Section 41.1).

39.2 Use and Examples

A typical use for a TimeInterval in a geophysical model is representation of the time step by which the model is advanced. Some models change the size of their time step as the model run progresses; this could be done by incrementing or decrementing the original time step by another TimeInterval, or by dividing or multiplying the time step by an integer value. An example of advancing model time using a TimeInterval representation of a time step is shown in Section 40.1.

The following brief example shows how to create, initialize and manipulate `TimeInterval`.

```
! !PROGRAM: ESMF_TimeIntervalEx - Time Interval initialization and
!                                  manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of Time Interval initialization and manipulation
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF
      implicit none

      ! instantiate some time intervals
      type(ESMF_TimeInterval) :: timeinterval1, timeinterval2, timeinterval3

      ! local variables
      integer :: d, h, m, s

      ! return code
      integer:: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultCalKind=ESMF_CALKIND_GREGORIAN, &
        defaultlogfilename="TimeIntervalEx.Log", &
```

```
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

39.2.1 TimeInterval initialization

This example shows how to initialize two `ESMF_TimeIntervals`.

```
! initialize time interval1 to 1 day
call ESMF_TimeIntervalSet(timeinterval1, d=1, rc=rc)


call ESMF_TimeIntervalPrint(timeinterval1, options="string", rc=rc)


! initialize time interval2 to 4 days, 1 hour, 30 minutes, 10 seconds
call ESMF_TimeIntervalSet(timeinterval2, d=4, h=1, m=30, s=10, rc=rc)


call ESMF_TimeIntervalPrint(timeinterval2, options="string", rc=rc)
```

39.2.2 TimeInterval conversion

This example shows how to convert `ESMF_TimeIntervals` into different units.

```
call ESMF_TimeIntervalGet(timeinterval1, s=s, rc=rc)
print *, "Time Interval1 = ", s, " seconds."


call ESMF_TimeIntervalGet(timeinterval2, h=h, m=m, s=s, rc=rc)
print *, "Time Interval2 = ", h, " hours, ", m, " minutes, ", &
                              s, " seconds."
```

39.2.3 TimeInterval difference

This example shows how to calculate the difference between two `ESMF_TimeIntervals`.

```
 ! difference between two time intervals
 timeinterval3 = timeinterval2 - timeinterval1
call ESMF_TimeIntervalGet(timeinterval3, d=d, h=h, m=m, s=s, rc=rc)
print *, "Difference between TimeInterval2 and TimeInterval1 = ", &
      d, " days, ", h, " hours, ", m, " minutes, ", s, " seconds."
```

39.2.4 TimeInterval multiplication

This example shows how to multiply an `ESMF_TimeInterval`.

```
! multiply time interval by an integer
timeinterval3 = timeinterval2 * 3
call ESMF_TimeIntervalGet(timeinterval3, d=d, h=h, m=m, s=s, rc=rc)
print *, "TimeInterval2 multiplied by 3 = ", d, " days, ", h, &
         " hours, ", m, " minutes, ", s, " seconds."
```

39.2.5 TimeInterval comparison

This example shows how to compare two `ESMF_TimeIntervals`.

```
! comparison
if (timeinterval1 < timeinterval2) then
  print *, "TimeInterval1 is smaller than TimeInterval2"
else
  print *, "TimeInterval1 is larger than or equal to TimeInterval2"
end if

! finalize ESMF framework
call ESMF_Finalize(rc=rc)


end program ESMF_TimeIntervalEx
```

39.3 Restrictions and Future Work

1. **Limits on time span.** The limits on the time span that can be represented are based on the 64-bit integer types used. For seconds, a signed 64-bit integer will have a range of +/- 2^{63}-1, or +/- 9,223,372,036,854,775,807. This corresponds to a range of +/- (2^{63}-1)/(86400 * 365.25) or +/- 292,271,023,045 years.

   For fractional seconds, a signed 64-bit integer will handle a resolution of +/- 2^{31}-1, or +/- 9,223,372,036,854,775,807 parts of a second.

39.4 Class API

39.4.1 ESMF_TimeIntervalAssignment(=) - Assign a TimeInterval to another TimeInterval

INTERFACE:

```
interface assignment(=)
timeinterval1 = timeinterval2
```

ARGUMENTS:

```
type(ESMF_TimeInterval) :: timeinterval1
type(ESMF_TimeInterval) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Set `timeinterval1` equal to `timeinterval2`. This is the default Fortran assignment, which creates a complete, independent copy of `timeinterval2` as `timeinterval1`. If `timeinterval2` is an invalid `ESMF_TimeInterval` object then `timeinterval1` will be equally invalid after the assignment.
The arguments are:

timeinterval1 The `ESMF_TimeInterval` to be set.

timeinterval2 The `ESMF_TimeInterval` to be copied.

39.4.2 ESMF_TimeIntervalOperator(+) - Add two TimeIntervals

INTERFACE:

```
interface operator(+)
sum = timeinterval1 + timeinterval2
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: sum
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (+) operator for the `ESMF_TimeInterval` class to add `timeinterval1` to `timeinterval2` and return the sum as an `ESMF_TimeInterval`.
The arguments are:

timeinterval1 The augend.

timeinterval2 The addend.

39.4.3 ESMF_TimeIntervalOperator(-) - Subtract one TimeInterval from another

INTERFACE:

```
interface operator(-)
difference = timeinterval1 - timeinterval2
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: difference
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (-) operator for the `ESMF_TimeInterval` class to subtract `timeinterval2` from `timeinterval1` and return the difference as an `ESMF_TimeInterval`.
The arguments are:

timeinterval1 The minuend.

timeinterval2 The subtrahend.

39.4.4 ESMF_TimeIntervalOperator(-) - Perform unary negation on a TimeInterval

INTERFACE:

```
interface operator(-)
timeinterval = -timeinterval
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: -timeInterval
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (-) operator for the `ESMF_TimeInterval` class to perform unary negation on `timeinterval` and return the result.
The arguments are:

timeinterval The time interval to be negated.

39.4.5 ESMF_TimeIntervalOperator(/) - Divide two TimeIntervals, return double precision quotient

INTERFACE:

```
interface operator(/)
quotient = timeinterval1 / timeinterval2
```

RETURN VALUE:

```
real(ESMF_KIND_R8) :: quotient
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (/) operator for the `ESMF_TimeInterval` class to return `timeinterval1` divided by `timeinterval2` as a double precision quotient.
The arguments are:

timeinterval1 The dividend.

timeinterval2 The divisor.

39.4.6 ESMF_TimeIntervalOperator(/) - Divide a TimeInterval by an integer, return TimeInterval quotient

INTERFACE:

```
interface operator(/)
quotient = timeinterval / divisor
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: quotient
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
integer(ESMF_KIND_I4),   intent(in) :: divisor
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (/) operator for the `ESMF_TimeInterval` class to divide a `timeinterval` by an integer `divisor`, and return the quotient as an `ESMF_TimeInterval`.
The arguments are:

timeinterval The dividend.

divisor Integer divisor.

39.4.7 ESMF_TimeIntervalFunction(MOD) - Divide two TimeIntervals, return TimeInterval remainder

INTERFACE:

```
interface MOD
function MOD(timeinterval1, timeinterval2)
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: MOD
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the Fortran intrinsic MOD() function for the `ESMF_TimeInterval` class to return the remainder of `timeinterval1` divided by `timeinterval2` as an `ESMF_TimeInterval`.
The arguments are:

timeinterval1 The dividend.

timeinterval2 The divisor.

39.4.8 ESMF_TimeIntervalOperator(*) - Multiply a TimeInterval by an integer

INTERFACE:

```
interface operator(*)
product = timeinterval * multiplier
                 OR
product = multiplier * timeinterval
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: product
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
integer(ESMF_KIND_I4),   intent(in) :: multiplier
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (*) operator for the `ESMF_TimeInterval` class to multiply a `timeinterval` by an integer `multiplier`, and return the product as an `ESMF_TimeInterval`.
The arguments are:

timeinterval The multiplicand.

multiplier The integer multiplier.

39.4.9 ESMF_TimeIntervalOperator(==) - Test if TimeInterval 1 is equal to TimeInterval 2

INTERFACE:

```
interface operator(==)
if (timeinterval1 == timeinterval2) then ... endif
                OR
result = (timeinterval1 == timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (==) operator for the `ESMF_TimeInterval` class to return `.true.` if `timeinterval1` and `timeinterval2` represent an equal duration of time, and `.false.` otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

39.4.10 ESMF_TimeIntervalOperator(/=) - Test if TimeInterval 1 is not equal to TimeInterval 2

INTERFACE:

```
interface operator(/=)
if (timeinterval1 /= timeinterval2) then ... endif
             OR
result = (timeinterval1 /= timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_TimeInterval` class to return `.true.` if `timeinterval1` and `timeinterval2` do not represent an equal duration of time, and `.false.` otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

39.4.11 ESMF_TimeIntervalOperator(<) - Test if TimeInterval 1 is less than TimeInterval 2

INTERFACE:

```
interface operator(<)
if (timeinterval1 < timeinterval2) then ... endif
             OR
result = (timeinterval1 < timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (<) operator for the `ESMF_TimeInterval` class to return `.true.` if `timeinterval1` is a lesser duration of time than `timeinterval2`, and `.false.` otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

39.4.12 ESMF_TimeIntervalOperator(<=) - Test if TimeInterval 1 is less than or equal to TimeInterval 2

INTERFACE:

```
interface operator(<=)
if (timeinterval1 <= timeinterval2) then ... endif
             OR
result = (timeinterval1 <= timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (<=) operator for the `ESMF_TimeInterval` class to return `.true.` if `timeinterval1` is a lesser or equal duration of time than `timeinterval2`, and `.false.` otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

39.4.13 ESMF_TimeIntervalOperator(>) - Test if TimeInterval 1 is greater than TimeInterval 2

INTERFACE:

```
interface operator(>)
if (timeinterval1 > timeinterval2) then ... endif
             OR
result = (timeinterval1 > timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (>) operator for the `ESMF_TimeInterval` class to return `.true.` if `timeinterval1` is a greater duration of time than `timeinterval2`, and `.false.` otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

39.4.14 ESMF_TimeIntervalOperator(>=) - Test if TimeInterval 1 is greater than or equal to TimeInterval 2

INTERFACE:

```
interface operator(>=)
if (timeinterval1 >= timeinterval2) then ... endif
             OR
result = (timeinterval1 >= timeinterval2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval1
type(ESMF_TimeInterval), intent(in) :: timeinterval2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Overloads the (>=) operator for the `ESMF_TimeInterval` class to return `.true.` if `timeinterval1` is a greater or equal duration of time than `timeinterval2`, and `.false.` otherwise.
The arguments are:

timeinterval1 First `ESMF_TimeInterval` in comparison.

timeinterval2 Second `ESMF_TimeInterval` in comparison.

39.4.15 ESMF_TimeIntervalAbsValue - Get the absolute value of a TimeInterval

INTERFACE:

```
function ESMF_TimeIntervalAbsValue(timeinterval)
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: ESMF_TimeIntervalAbsValue
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns the absolute value of `timeinterval`.
The argument is:

timeinterval The object instance to take the absolute value of. Absolute value is returned as the value of the function.

39.4.16 ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
! Private name; call using ESMF_TimeIntervalGet()
subroutine ESMF_TimeIntervalGetDur(timeinterval, &
  yy, yy_i8, &
  mm, mm_i8, &
  d, d_i8, &
  h, m, &
  s, s_i8, &
  ms, us, ns, &
  d_r8, h_r8, m_r8, s_r8, &
  ms_r8, us_r8, ns_r8, &
  sN, sN_i8, sD, sD_i8, &
  startTime, calendar, calkindflag, &
  timeString, timeStringISOFrac, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(in)            :: timeinterval
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(out), optional :: yy
      integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: mm
      integer(ESMF_KIND_I8),   intent(out), optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: d
      integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: h
      integer(ESMF_KIND_I4),   intent(out), optional :: m
      integer(ESMF_KIND_I4),   intent(out), optional :: s
      integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: ms
      integer(ESMF_KIND_I4),   intent(out), optional :: us
      integer(ESMF_KIND_I4),   intent(out), optional :: ns
      real(ESMF_KIND_R8),      intent(out), optional :: d_r8
      real(ESMF_KIND_R8),      intent(out), optional :: h_r8
      real(ESMF_KIND_R8),      intent(out), optional :: m_r8
      real(ESMF_KIND_R8),      intent(out), optional :: s_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ms_r8
      real(ESMF_KIND_R8),      intent(out), optional :: us_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ns_r8
      integer(ESMF_KIND_I4),   intent(out), optional :: sN
      integer(ESMF_KIND_I8),   intent(out), optional :: sN_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: sD
      integer(ESMF_KIND_I8),   intent(out), optional :: sD_i8
      type(ESMF_Time),         intent(out), optional :: startTime
      type(ESMF_Calendar),     intent(out), optional :: calendar
      type(ESMF_CalKind_Flag), intent(out), optional :: calkindflag
      character (len=*),       intent(out), optional :: timeString
      character (len=*),       intent(out), optional :: timeStringISOFrac
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the value of `timeinterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers.
Units are bound (normalized) to the next larger unit specified. For example, if a time interval is defined to be one day, then `ESMF_TimeIntervalGet(d = days, s = seconds)` would return `days = 1`, `seconds = 0`, whereas `ESMF_TimeIntervalGet(s = seconds)` would return `seconds = 86400`.
For timeString, converts `ESMF_TimeInterval`'s value into partial ISO 8601 format PyYmMdDThHmMs[:n/d]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
For timeStringISOFrac, converts `ESMF_TimeInterval`'s value into full ISO 8601 format PyYmMdDThHmMs[.f]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
The arguments are:

timeinterval The object instance to query.

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

[yy_i8] Integer years (large, >= 64-bit).

[mm] Integer months (>= 32-bit).

[mm_i8] Integer months (large, >= 64-bit).

[d] Integer Julian, or Modified Julian, days (>= 32-bit).

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

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

[ms] Integer milliseconds.

[us] Integer microseconds.

[ns] Integer nanoseconds.

[d_r8] Double precision days.

[h_r8] Double precision hours.

[m_r8] Double precision minutes.

[s_r8] Double precision seconds.

[ms_r8] Double precision milliseconds.

[us_r8] Double precision microseconds.

[ns_r8] Double precision nanoseconds.

[sN] Integer numerator of fractional seconds (sN/sD).

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[sD] Integer denominator of fractional seconds (sN/sD).

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).

[calendar] Associated `Calendar`, if any.

[calkindflag] Associated `CalKind_Flag`, if any.

[timeString] Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[.f], where f is decimal form of any fractional seconds. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

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

39.4.17 ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
      ! Private name; call using ESMF_TimeIntervalGet()
      subroutine ESMF_TimeIntervalGetDurStart(timeinterval, startTimeIn, &
        &
        yy, yy_i8, &
        mm, mm_i8, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, &
        startTime, &
        calendar, calkindflag, &
        timeString, timeStringISOFrac, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(in)            :: timeinterval
      type(ESMF_Time),         intent(in)            :: startTimeIn ! Input
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(out), optional :: yy
      integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: mm
      integer(ESMF_KIND_I8),   intent(out), optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: d
      integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: h
      integer(ESMF_KIND_I4),   intent(out), optional :: m
      integer(ESMF_KIND_I4),   intent(out), optional :: s
      integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: ms
      integer(ESMF_KIND_I4),   intent(out), optional :: us
      integer(ESMF_KIND_I4),   intent(out), optional :: ns
      real(ESMF_KIND_R8),      intent(out), optional :: d_r8
```

```
real(ESMF_KIND_R8),      intent(out), optional :: h_r8
real(ESMF_KIND_R8),      intent(out), optional :: m_r8
real(ESMF_KIND_R8),      intent(out), optional :: s_r8
real(ESMF_KIND_R8),      intent(out), optional :: ms_r8
real(ESMF_KIND_R8),      intent(out), optional :: us_r8
real(ESMF_KIND_R8),      intent(out), optional :: ns_r8
integer(ESMF_KIND_I4),   intent(out), optional :: sN
integer(ESMF_KIND_I8),   intent(out), optional :: sN_i8
integer(ESMF_KIND_I4),   intent(out), optional :: sD
integer(ESMF_KIND_I8),   intent(out), optional :: sD_i8
type(ESMF_Time),         intent(out), optional :: startTime
type(ESMF_Calendar),     intent(out), optional :: calendar
type(ESMF_CalKind_Flag), intent(out), optional :: calkindflag
character (len=*),       intent(out), optional :: timeString
character (len=*),       intent(out), optional :: timeStringISOFrac
integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the value of `timeinterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers.
Units are bound (normalized) to the next larger unit specified. For example, if a time interval is defined to be one day, then `ESMF_TimeIntervalGet(d = days, s = seconds)` would return `days = 1`, `seconds = 0`, whereas `ESMF_TimeIntervalGet(s = seconds)` would return `seconds = 86400`.
For timeString, converts `ESMF_TimeInterval`'s value into partial ISO 8601 format PyYmMdDThHmMs[:n/d]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
For timeStringISOFrac, converts `ESMF_TimeInterval`'s value into full ISO 8601 format PyYmMdDThHmMs[.f]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
The arguments are:

timeinterval The object instance to query.

startTimeIn INPUT argument: pins a calendar interval to a specific point in time to allow conversion between relative units (yy, mm, d) and absolute units (d, h, m, s). Overrides any startTime and/or endTime previously set. Mutually exclusive with endTimeIn and calendarIn.

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

[yy_i8] Integer years (large, >= 64-bit).

[mm] Integer months (>= 32-bit).

[mm_i8] Integer months (large, >= 64-bit).

[d] Integer Julian, or Modified Julian, days (>= 32-bit).

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

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

[ms] Integer milliseconds.

[us] Integer microseconds.

[ns] Integer nanoseconds.

[d_r8] Double precision days.

[h_r8] Double precision hours.

[m_r8] Double precision minutes.

[s_r8] Double precision seconds.

[ms_r8] Double precision milliseconds.

[us_r8] Double precision microseconds.

[ns_r8] Double precision nanoseconds.

[sN] Integer numerator of fractional seconds (sN/sD).

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[sD] Integer denominator of fractional seconds (sN/sD).

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).

[calendar] Associated `Calendar`, if any.

[calkindflag] Associated `CalKind_Flag`, if any.

[timeString] Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[.f], where f is decimal form of any fractional seconds. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

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

39.4.18 ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
      ! Private name; call using ESMF_TimeIntervalGet()
      subroutine ESMF_TimeIntervalGetDurCal(timeinterval, calendarIn, &
        &
        yy, yy_i8, &
        mm, mm_i8, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, &
```

```
        startTime, &
        calendar, calkindflag, &
        timeString, timeStringISOFrac, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(in)            :: timeinterval
      type(ESMF_Calendar),     intent(in)            :: calendarIn ! Input
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(out), optional :: yy
      integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: mm
      integer(ESMF_KIND_I8),   intent(out), optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: d
      integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: h
      integer(ESMF_KIND_I4),   intent(out), optional :: m
      integer(ESMF_KIND_I4),   intent(out), optional :: s
      integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: ms
      integer(ESMF_KIND_I4),   intent(out), optional :: us
      integer(ESMF_KIND_I4),   intent(out), optional :: ns
      real(ESMF_KIND_R8),      intent(out), optional :: d_r8
      real(ESMF_KIND_R8),      intent(out), optional :: h_r8
      real(ESMF_KIND_R8),      intent(out), optional :: m_r8
      real(ESMF_KIND_R8),      intent(out), optional :: s_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ms_r8
      real(ESMF_KIND_R8),      intent(out), optional :: us_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ns_r8
      integer(ESMF_KIND_I4),   intent(out), optional :: sN
      integer(ESMF_KIND_I8),   intent(out), optional :: sN_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: sD
      integer(ESMF_KIND_I8),   intent(out), optional :: sD_i8
      type(ESMF_Time),         intent(out), optional :: startTime
      type(ESMF_Calendar),     intent(out), optional :: calendar
      type(ESMF_CalKind_Flag), intent(out), optional :: calkindflag
      character (len=*),       intent(out), optional :: timeString
      character (len=*),       intent(out), optional :: timeStringISOFrac
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the value of `timeinterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers.
Units are bound (normalized) to the next larger unit specified. For example, if a time interval is defined to be one day, then `ESMF_TimeIntervalGet(d = days, s = seconds)` would return `days = 1`, `seconds = 0`, whereas `ESMF_TimeIntervalGet(s = seconds)` would return `seconds = 86400`.
For timeString, converts `ESMF_TimeInterval`'s value into partial ISO 8601 format PyYmMdDThHmMs[:n/d]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
For timeStringISOFrac, converts `ESMF_TimeInterval`'s value into full ISO 8601 format PyYmMdDThHmMs[.f]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
The arguments are:

timeinterval The object instance to query.

calendarIn INPUT argument: pins a calendar interval to a specific calendar to allow conversion between relative units (yy, mm, d) and absolute units (d, h, m, s). Mutually exclusive with startTimeIn and endTimeIn since they contain a calendar. Alternate to, and mutually exclusive with, calkindflagIn below. Primarily for specifying a custom calendar kind.

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

[yy_i8] Integer years (large, >= 64-bit).

[mm] Integer months (>= 32-bit).

[mm_i8] Integer months (large, >= 64-bit).

[d] Integer Julian, or Modified Julian, days (>= 32-bit).

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

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

[ms] Integer milliseconds.

[us] Integer microseconds.

[ns] Integer nanoseconds.

[d_r8] Double precision days.

[h_r8] Double precision hours.

[m_r8] Double precision minutes.

[s_r8] Double precision seconds.

[ms_r8] Double precision milliseconds.

[us_r8] Double precision microseconds.

[ns_r8] Double precision nanoseconds.

[sN] Integer numerator of fractional seconds (sN/sD).

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[sD] Integer denominator of fractional seconds (sN/sD).

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).

[calendar] Associated `Calendar`, if any.

[calkindflag] Associated `CalKind_Flag`, if any.

[timeString] Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[.f], where f is decimal form of any fractional seconds. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

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

39.4.19 ESMF_TimeIntervalGet - Get a TimeInterval value

INTERFACE:

```
      ! Private name; call using ESMF_TimeIntervalGet()
      subroutine ESMF_TimeIntervalGetDurCalTyp(timeinterval, calkindflagIn, &
        &
        yy, yy_i8, &
        mm, mm_i8, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, &
        startTime, &
        calendar, calkindflag, &
        timeString, &
        timeStringISOFrac, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(in)            :: timeinterval
      type(ESMF_CalKind_Flag), intent(in)            :: calkindflagIn ! Input
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(out), optional :: yy
      integer(ESMF_KIND_I8),   intent(out), optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: mm
      integer(ESMF_KIND_I8),   intent(out), optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: d
      integer(ESMF_KIND_I8),   intent(out), optional :: d_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: h
      integer(ESMF_KIND_I4),   intent(out), optional :: m
      integer(ESMF_KIND_I4),   intent(out), optional :: s
      integer(ESMF_KIND_I8),   intent(out), optional :: s_i8
      integer(ESMF_KIND_I4),   intent(out), optional :: ms
      integer(ESMF_KIND_I4),   intent(out), optional :: us
      integer(ESMF_KIND_I4),   intent(out), optional :: ns
      real(ESMF_KIND_R8),      intent(out), optional :: d_r8
      real(ESMF_KIND_R8),      intent(out), optional :: h_r8
      real(ESMF_KIND_R8),      intent(out), optional :: m_r8
      real(ESMF_KIND_R8),      intent(out), optional :: s_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ms_r8
      real(ESMF_KIND_R8),      intent(out), optional :: us_r8
      real(ESMF_KIND_R8),      intent(out), optional :: ns_r8
      integer(ESMF_KIND_I4),   intent(out), optional :: sN
      integer(ESMF_KIND_I8),   intent(out), optional :: sN_i8
```

```
integer(ESMF_KIND_I4),   intent(out), optional :: sD
integer(ESMF_KIND_I8),   intent(out), optional :: sD_i8
type(ESMF_Time),         intent(out), optional :: startTime
type(ESMF_Calendar),     intent(out), optional :: calendar
type(ESMF_CalKind_Flag), intent(out), optional :: calkindflag
character (len=*),       intent(out), optional :: timeString
character (len=*),       intent(out), optional :: timeStringISOFrac
integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the value of `timeinterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally from integers.
Units are bound (normalized) to the next larger unit specified. For example, if a time interval is defined to be one day, then `ESMF_TimeIntervalGet(d = days, s = seconds)` would return `days = 1`, `seconds = 0`, whereas `ESMF_TimeIntervalGet(s = seconds)` would return `seconds = 86400`.
For timeString, converts `ESMF_TimeInterval`'s value into partial ISO 8601 format PyYmMdDThHmMs[:n/d]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
For timeStringISOFrac, converts `ESMF_TimeInterval`'s value into full ISO 8601 format PyYmMdDThHmMs[.f]S. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.
The arguments are:

timeinterval The object instance to query.

calkindflagIn INPUT argument: Alternate to, and mutually exclusive with, calendarIn above. More convenient way of specifying a built-in calendar kind.

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

[yy_i8] Integer years (large, >= 64-bit).

[mm] Integer months (>= 32-bit).

[mm_i8] Integer months (large, >= 64-bit).

[d] Integer Julian, or Modified Julian, days (>= 32-bit).

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit).

[h] Integer hours.

[m] Integer minutes.

[s] Integer seconds (>= 32-bit).

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

[ms] Integer milliseconds.

[us] Integer microseconds.

[ns] Integer nanoseconds.

[d_r8] Double precision days.

[h_r8] Double precision hours.

[m_r8] Double precision minutes.

[s_r8] Double precision seconds.

[ms_r8] Double precision milliseconds.

[us_r8] Double precision microseconds.

[ns_r8] Double precision nanoseconds.

[sN] Integer numerator of fractional seconds (sN/sD).

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[sD] Integer denominator of fractional seconds (sN/sD).

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit).

[startTime] Starting time, if set, of an absolute calendar interval (yy, mm, and/or d).

[calendar] Associated `Calendar`, if any.

[calkindflag] Associated `CalKind_Flag`, if any.

[timeString] Convert time interval value to format string PyYmMdDThHmMs[:n/d]S, where n/d is numerator/denominator of any fractional seconds and all other units are in ISO 8601 format. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

[timeStringISOFrac] Convert time interval value to strict ISO 8601 format string PyYmMdDThHmMs[.f], where f is decimal form of any fractional seconds. See [12] and [2]. See also method `ESMF_TimeIntervalPrint()`.

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

39.4.20 ESMF_TimeIntervalNegAbsValue - Return the negative absolute value of a TimeInterval

INTERFACE:

```
function ESMF_TimeIntervalNegAbsValue(timeinterval)
```

RETURN VALUE:

```
type(ESMF_TimeInterval) :: ESMF_TimeIntervalNegAbsValue
```

ARGUMENTS:

```
type(ESMF_TimeInterval), intent(in) :: timeinterval
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns the negative absolute value of `timeinterval`.
The argument is:

timeinterval The object instance to take the negative absolute value of. Negative absolute value is returned as the value of the function.

39.4.21 ESMF_TimeIntervalPrint - Print the contents of a TimeInterval

INTERFACE:

```
subroutine ESMF_TimeIntervalPrint(timeinterval, options, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Prints out the contents of an `ESMF_TimeInterval` to `stdout`, in support of testing and debugging. The options control the type of information and level of detail.

The arguments are:

timeinterval Time interval to be printed out.

[options] Print options. If none specified, prints all `timeinterval` property values.
"string" - prints `timeinterval`'s value in ISO 8601 format for all units through seconds. For any non-zero fractional seconds, prints in integer rational fraction form n/d. Format is PyYmMdDThHmMs[:n/d]S, where [:n/d] is the integer numerator and denominator of the fractional seconds value, if present. See [12] and [2]. See also method `ESMF_TimeIntervalGet(..., timeString= , ...)`
"string isofrac" - prints `timeinterval`'s value in strict ISO 8601 format for all units, including any fractional seconds part. Format is PyYmMdDThHmMs[.f]S, where [.f] represents fractional seconds in decimal form, if present. See [12] and [2]. See also method `ESMF_TimeIntervalGet(..., timeStringISOFrac= , ...)`

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

39.4.22 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
! Private name; call using ESMF_TimeIntervalSet()
subroutine ESMF_TimeIntervalSetDur(timeinterval, &
  yy, yy_i8, &
  mm, mm_i8, &
  d, d_i8, &
  h, m, &
  s, s_i8, &
  ms, us, ns, &
  d_r8, h_r8, m_r8, s_r8, &
  ms_r8, us_r8, ns_r8, &
  sN, sN_i8, sD, sD_i8, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(inout)         :: timeinterval
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(in),  optional :: yy
      integer(ESMF_KIND_I8),   intent(in),  optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: mm
      integer(ESMF_KIND_I8),   intent(in),  optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: d
      integer(ESMF_KIND_I8),   intent(in),  optional :: d_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: h
      integer(ESMF_KIND_I4),   intent(in),  optional :: m
      integer(ESMF_KIND_I4),   intent(in),  optional :: s
      integer(ESMF_KIND_I8),   intent(in),  optional :: s_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: ms
      integer(ESMF_KIND_I4),   intent(in),  optional :: us
      integer(ESMF_KIND_I4),   intent(in),  optional :: ns
      real(ESMF_KIND_R8),      intent(in),  optional :: d_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: h_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: m_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: s_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: ms_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: us_r8
      real(ESMF_KIND_R8),      intent(in),  optional :: ns_r8
      integer(ESMF_KIND_I4),   intent(in),  optional :: sN
      integer(ESMF_KIND_I8),   intent(in),  optional :: sN_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: sD
      integer(ESMF_KIND_I8),   intent(in),  optional :: sD_i8
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets the value of the `ESMF_TimeInterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers.
Ranges are limited only by machine word size. Numeric defaults are 0, except for sD, which is 1.
The arguments are:

timeinterval The object instance to initialize.

[yy] Integer years (>= 32-bit). Default = 0

[yy_i8] Integer years (large, >= 64-bit). Default = 0

[mm] Integer months (>= 32-bit). Default = 0

[mm_i8] Integer months (large, >= 64-bit). Default = 0

[d] Integer Julian, or Modified Julian, days (>= 32-bit). Default = 0

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0

[us] Integer microseconds. Default = 0

[ns] Integer nanoseconds. Default = 0

[d_r8] Double precision days. Default = 0.0.

[h_r8] Double precision hours. Default = 0.0.

[m_r8] Double precision minutes. Default = 0.0.

[s_r8] Double precision seconds. Default = 0.0.

[ms_r8] Double precision milliseconds. Default = 0.0.

[us_r8] Double precision microseconds. Default = 0.0.

[ns_r8] Double precision nanoseconds. Default = 0.0.

[sN] Integer numerator of fractional seconds (sN/sD). Default = 0

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 0

[sD] Integer denominator of fractional seconds (sN/sD). Default = 1

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 1

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

39.4.23 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
      ! Private name; call using ESMF_TimeIntervalSet()
      subroutine ESMF_TimeIntervalSetDurStart(timeinterval, startTime, &
        &
        yy, yy_i8, &
        mm, mm_i8, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, &
        rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(inout)         :: timeinterval
      type(ESMF_Time),         intent(in)            :: startTime
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(in),  optional :: yy
      integer(ESMF_KIND_I8),   intent(in),  optional :: yy_i8
```

```
integer(ESMF_KIND_I4),   intent(in),  optional :: mm
integer(ESMF_KIND_I8),   intent(in),  optional :: mm_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: d
integer(ESMF_KIND_I8),   intent(in),  optional :: d_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: h
integer(ESMF_KIND_I4),   intent(in),  optional :: m
integer(ESMF_KIND_I4),   intent(in),  optional :: s
integer(ESMF_KIND_I8),   intent(in),  optional :: s_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: ms
integer(ESMF_KIND_I4),   intent(in),  optional :: us
integer(ESMF_KIND_I4),   intent(in),  optional :: ns
real(ESMF_KIND_R8),      intent(in),  optional :: d_r8
real(ESMF_KIND_R8),      intent(in),  optional :: h_r8
real(ESMF_KIND_R8),      intent(in),  optional :: m_r8
real(ESMF_KIND_R8),      intent(in),  optional :: s_r8
real(ESMF_KIND_R8),      intent(in),  optional :: ms_r8
real(ESMF_KIND_R8),      intent(in),  optional :: us_r8
real(ESMF_KIND_R8),      intent(in),  optional :: ns_r8
integer(ESMF_KIND_I4),   intent(in),  optional :: sN
integer(ESMF_KIND_I8),   intent(in),  optional :: sN_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: sD
integer(ESMF_KIND_I8),   intent(in),  optional :: sD_i8
integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets the value of the `ESMF_TimeInterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers.
Ranges are limited only by machine word size. Numeric defaults are 0, except for sD, which is 1.
The arguments are:

timeinterval The object instance to initialize.

startTime Starting time of an absolute calendar interval (yy, mm, and/or d); pins a calendar interval to a specific point in time. If not set, and calendar also not set, calendar interval "floats" across all calendars and times.

[yy] Integer years (>= 32-bit). Default = 0

[yy_i8] Integer years (large, >= 64-bit). Default = 0

[mm] Integer months (>= 32-bit). Default = 0

[mm_i8] Integer months (large, >= 64-bit). Default = 0

[d] Integer Julian, or Modified Julian, days (>= 32-bit). Default = 0

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0

[us] Integer microseconds. Default = 0

[ns] Integer nanoseconds. Default = 0

[d_r8] Double precision days. Default = 0.0.

[h_r8] Double precision hours. Default = 0.0.

[m_r8] Double precision minutes. Default = 0.0.

[s_r8] Double precision seconds. Default = 0.0.

[ms_r8] Double precision milliseconds. Default = 0.0.

[us_r8] Double precision microseconds. Default = 0.0.

[ns_r8] Double precision nanoseconds. Default = 0.0.

[sN] Integer numerator of fractional seconds (sN/sD). Default = 0

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 0

[sD] Integer denominator of fractional seconds (sN/sD). Default = 1

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8). (large, >= 64-bit). Default = 1

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

39.4.24 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
      ! Private name; call using ESMF_TimeIntervalSet()
      subroutine ESMF_TimeIntervalSetDurCal(timeinterval, calendar, &
        &
        yy, yy_i8, &
        mm, mm_i8, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(inout)         :: timeinterval
      type(ESMF_Calendar),     intent(in)            :: calendar
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),   intent(in),  optional :: yy
      integer(ESMF_KIND_I8),   intent(in),  optional :: yy_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: mm
      integer(ESMF_KIND_I8),   intent(in),  optional :: mm_i8
      integer(ESMF_KIND_I4),   intent(in),  optional :: d
```

```
integer(ESMF_KIND_I8),   intent(in),  optional :: d_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: h
integer(ESMF_KIND_I4),   intent(in),  optional :: m
integer(ESMF_KIND_I4),   intent(in),  optional :: s
integer(ESMF_KIND_I8),   intent(in),  optional :: s_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: ms
integer(ESMF_KIND_I4),   intent(in),  optional :: us
integer(ESMF_KIND_I4),   intent(in),  optional :: ns
real(ESMF_KIND_R8),      intent(in),  optional :: d_r8
real(ESMF_KIND_R8),      intent(in),  optional :: h_r8
real(ESMF_KIND_R8),      intent(in),  optional :: m_r8
real(ESMF_KIND_R8),      intent(in),  optional :: s_r8
real(ESMF_KIND_R8),      intent(in),  optional :: ms_r8
real(ESMF_KIND_R8),      intent(in),  optional :: us_r8
real(ESMF_KIND_R8),      intent(in),  optional :: ns_r8
integer(ESMF_KIND_I4),   intent(in),  optional :: sN
integer(ESMF_KIND_I8),   intent(in),  optional :: sN_i8
integer(ESMF_KIND_I4),   intent(in),  optional :: sD
integer(ESMF_KIND_I8),   intent(in),  optional :: sD_i8
integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets the value of the `ESMF_TimeInterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers.
Ranges are limited only by machine word size. Numeric defaults are 0, except for sD, which is 1.
The arguments are:

timeinterval The object instance to initialize.

calendar `Calendar` used to give better definition to calendar interval (yy, mm, and/or d) for arithmetic, comparison, and conversion operations. Allows calendar interval to "float" across all times on a specific calendar. Default = NULL; if startTime also not specified, calendar interval "floats" across all calendars and times. Mutually exclusive with startTime since it contains a calendar. Alternate to, and mutually exclusive with, calkindflag below. Primarily for specifying a custom calendar kind.

[yy] Integer years (>= 32-bit). Default = 0

[yy_i8] Integer years (large, >= 64-bit). Default = 0

[mm] Integer months (>= 32-bit). Default = 0

[mm_i8] Integer months (large, >= 64-bit). Default = 0

[d] Integer Julian, or Modified Julian, days (>= 32-bit). Default = 0

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0

[us] Integer microseconds. Default = 0

[ns] Integer nanoseconds. Default = 0

[d_r8] Double precision days. Default = 0.0.

[h_r8] Double precision hours. Default = 0.0.

[m_r8] Double precision minutes. Default = 0.0.

[s_r8] Double precision seconds. Default = 0.0.

[ms_r8] Double precision milliseconds. Default = 0.0.

[us_r8] Double precision microseconds. Default = 0.0.

[ns_r8] Double precision nanoseconds. Default = 0.0.

[sN] Integer numerator of fractional seconds (sN/sD). Default = 0

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8). (large, >= 64-bit). Default = 0

[sD] Integer denominator of fractional seconds (sN/sD). Default = 1

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8). (large, >= 64-bit). Default = 1

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

39.4.25 ESMF_TimeIntervalSet - Initialize or set a TimeInterval

INTERFACE:

```
      ! Private name; call using ESMF_TimeIntervalSet()
      subroutine ESMF_TimeIntervalSetDurCalTyp(timeinterval, calkindflag, &
        &
        yy, yy_i8, &
        mm, mm_i8, &
        d, d_i8, &
        h, m, &
        s, s_i8, &
        ms, us, ns, &
        d_r8, h_r8, m_r8, s_r8, &
        ms_r8, us_r8, ns_r8, &
        sN, sN_i8, sD, sD_i8, &
        rc)
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(inout)          :: timeinterval
      type(ESMF_CalKind_Flag), intent(in)             :: calkindflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),    intent(in),  optional :: yy
      integer(ESMF_KIND_I8),    intent(in),  optional :: yy_i8
      integer(ESMF_KIND_I4),    intent(in),  optional :: mm
      integer(ESMF_KIND_I8),    intent(in),  optional :: mm_i8
```

```
        integer(ESMF_KIND_I4),   intent(in),  optional :: d
        integer(ESMF_KIND_I8),   intent(in),  optional :: d_i8
        integer(ESMF_KIND_I4),   intent(in),  optional :: h
        integer(ESMF_KIND_I4),   intent(in),  optional :: m
        integer(ESMF_KIND_I4),   intent(in),  optional :: s
        integer(ESMF_KIND_I8),   intent(in),  optional :: s_i8
        integer(ESMF_KIND_I4),   intent(in),  optional :: ms
        integer(ESMF_KIND_I4),   intent(in),  optional :: us
        integer(ESMF_KIND_I4),   intent(in),  optional :: ns
        real(ESMF_KIND_R8),      intent(in),  optional :: d_r8
        real(ESMF_KIND_R8),      intent(in),  optional :: h_r8
        real(ESMF_KIND_R8),      intent(in),  optional :: m_r8
        real(ESMF_KIND_R8),      intent(in),  optional :: s_r8
        real(ESMF_KIND_R8),      intent(in),  optional :: ms_r8
        real(ESMF_KIND_R8),      intent(in),  optional :: us_r8
        real(ESMF_KIND_R8),      intent(in),  optional :: ns_r8
        integer(ESMF_KIND_I4),   intent(in),  optional :: sN
        integer(ESMF_KIND_I8),   intent(in),  optional :: sN_i8
        integer(ESMF_KIND_I4),   intent(in),  optional :: sD
        integer(ESMF_KIND_I8),   intent(in),  optional :: sD_i8
        integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets the value of the `ESMF_TimeInterval` in units specified by the user via Fortran optional arguments.
The ESMF Time Manager represents and manipulates time internally with integers to maintain precision. Hence, user-specified floating point values are converted internally to integers.
Ranges are limited only by machine word size. Numeric defaults are 0, except for sD, which is 1.
The arguments are:

timeinterval The object instance to initialize.

calkindflag Alternate to, and mutually exclusive with, calendar above. More convenient way of specifying a built-in calendar kind.

[yy] Integer years (>= 32-bit). Default = 0

[yy_i8] Integer years (large, >= 64-bit). Default = 0

[mm] Integer months (>= 32-bit). Default = 0

[mm_i8] Integer months (large, >= 64-bit). Default = 0

[d] Integer Julian, or Modified Julian, days (>= 32-bit). Default = 0

[d_i8] Integer Julian, or Modified Julian, days (large, >= 64-bit). Default = 0

[h] Integer hours. Default = 0

[m] Integer minutes. Default = 0

[s] Integer seconds (>= 32-bit). Default = 0

[s_i8] Integer seconds (large, >= 64-bit). Default = 0

[ms] Integer milliseconds. Default = 0

[us] Integer microseconds. Default = 0

[ns] Integer nanoseconds. Default = 0

[d_r8] Double precision days. Default = 0.0.

[h_r8] Double precision hours. Default = 0.0.

[m_r8] Double precision minutes. Default = 0.0.

[s_r8] Double precision seconds. Default = 0.0.

[ms_r8] Double precision milliseconds. Default = 0.0.

[us_r8] Double precision microseconds. Default = 0.0.

[ns_r8] Double precision nanoseconds. Default = 0.0.

[sN] Integer numerator of fractional seconds (sN/sD). Default = 0

[sN_i8] Integer numerator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 0

[sD] Integer denominator of fractional seconds (sN/sD). Default = 1

[sD_i8] Integer denominator of fractional seconds (sN_i8/sD_i8) (large, >= 64-bit). Default = 1

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

39.4.26 ESMF_TimeIntervalValidate - Validate a TimeInterval

INTERFACE:

```
      subroutine ESMF_TimeIntervalValidate(timeinterval, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Checks whether a `timeinterval` is valid. If fractional value, denominator must be non-zero.
The arguments are:

timeinterval `ESMF_TimeInterval` to be validated.

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

40 Clock Class

40.1 Description

The Clock class advances model time and tracks its associated date on a specified Calendar. It stores start time, stop time, current time, previous time, and a time step. It can also store a reference time, typically the time instant at which a simulation originally began. For a restart run, the reference time can be different than the start time, when the application execution resumes.
A user can call the `ESMF_ClockSet` method and reset the time step as desired.
A Clock also stores a list of Alarms, which can be set to flag events that occur at a specified time instant or at a specified time interval. See Section 41.1 for details on how to use Alarms.
There are methods for setting and getting the Times and Alarms associated with a Clock. Methods are defined for advancing the Clock's current time, checking if the stop time has been reached, reversing direction, and synchronizing with a real clock.

40.2 Constants

40.2.1 ESMF_DIRECTION

DESCRIPTION:

Specifies the time-stepping direction of a clock. Use with "direction" argument to methods `ESMF_ClockSet()` and `ESMF_ClockGet()`. Cannot be used with method `ESMF_ClockCreate()`, since it only initializes a clock in the default forward mode; a clock must be advanced (timestepped) at least once before reversing direction via `ESMF_ClockSet()`. This also holds true for negative timestep clocks which are initialized (created) with stopTime < startTime, since "forward" means timestepping from startTime towards stopTime (see `ESMF_DIRECTION_FORWARD` below).
"Forward" and "reverse" directions are distinct from postive and negative timesteps. "Forward" means timestepping in the direction established at `ESMF_ClockCreate()`, from startTime towards stopTime, regardless of the timestep sign. "Reverse" means timestepping in the opposite direction, back towards the clock's startTime, regardless of the timestep sign.
Clocks and alarms run in reverse in such a way that the state of a clock and its alarms after each time step is precisely replicated as it was in forward time-stepping mode. All methods which query clock and alarm state will return the same result for a given timeStep, regardless of the direction of arrival.
The type of this flag is:
`type(ESMF_Direction_Flag)`
The valid values are:

ESMF_DIRECTION_FORWARD Upon calling `ESMF_ClockAdvance()`, the clock will timestep from its startTime toward its stopTime. This is the default direction. A user can use either `ESMF_ClockIsStopTime()` or `ESMF_ClockIsDone()` methods to determine when stopTime is reached. This forward behavior also holds for negative timestep clocks which are initialized (created) with stopTime < startTime.

ESMF_DIRECTION_REVERSE Upon calling `ESMF_ClockAdvance()`, the clock will timestep backwards toward its startTime. Use method `ESMF_ClockIsDone()` to determine when startTime is reached. This reverse behavior also holds for negative timestep clocks which are initialized (created) with stopTime < startTime.

40.3 Use and Examples

The following is a typical sequence for using a Clock in a geophysical model.
At initialize:

- Set a Calendar.
- Set start time, stop time and time step as Times and Time Intervals.
- Create and Initialize a Clock using the start time, stop time and time step.

- Define Times and Time Intervals associated with special events, and use these to set Alarms.

At run:

- Advance the Clock, checking for ringing alarms as needed.
- Check if it is time to stop.

At finalize:

- Since Clocks and Alarms are deep classes, they need to be explicitly destroyed at finalization. Times and TimeIntervals are lightweight classes, so they don't need explicit destruction.

The following code example illustrates Clock usage.

```
! !PROGRAM: ESMF_ClockEx - Clock initialization and time-stepping
!
! !DESCRIPTION:
!
! This program shows an example of how to create, initialize, advance, and
! examine a basic clock
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF
      implicit none

      ! instantiate a clock
      type(ESMF_Clock) :: clock

      ! instantiate time_step, start and stop times
      type(ESMF_TimeInterval) :: timeStep
      type(ESMF_Time) :: startTime
      type(ESMF_Time) :: stopTime

      ! local variables for Get methods
      type(ESMF_Time) :: currTime
      integer(ESMF_KIND_I8) :: advanceCount
      integer :: YY, MM, DD, H, M, S

      ! return code
      integer :: rc


      ! initialize ESMF framework
      call ESMF_Initialize(defaultCalKind=ESMF_CALKIND_GREGORIAN, &
        defaultlogfilename="ClockEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

40.3.1 Clock creation

This example shows how to create and initialize an `ESMF_Clock`.

```
      ! initialize time interval to 2 days, 4 hours (6 timesteps in 13 days)
      call ESMF_TimeIntervalSet(timeStep, d=2, h=4, rc=rc)
```

```
! initialize start time to 4/1/2003 2:24:00 ( 1/10 of a day )
call ESMF_TimeSet(startTime, yy=2003, mm=4, dd=1, h=2, m=24, rc=rc)


! initialize stop time to 4/14/2003 2:24:00 ( 1/10 of a day )
call ESMF_TimeSet(stopTime, yy=2003, mm=4, dd=14, h=2, m=24, rc=rc)


! initialize the clock with the above values
clock = ESMF_ClockCreate(timeStep, startTime, stopTime=stopTime, &
                         name="Clock 1", rc=rc)
```

40.3.2 Clock advance

This example shows how to time-step an `ESMF_Clock`.

```
! time step clock from start time to stop time
do while (.not.ESMF_ClockIsStopTime(clock, rc=rc))


  call ESMF_ClockPrint(clock, options="currTime string", rc=rc)


  call ESMF_ClockAdvance(clock, rc=rc)


end do
```

40.3.3 Clock examination

This example shows how to examine an `ESMF_Clock`.

```
! get the clock's final current time
call ESMF_ClockGet(clock, currTime=currTime, rc=rc)


call ESMF_TimeGet(currTime, yy=YY, mm=MM, dd=DD, h=H, m=M, s=S, rc=rc)
print *, "The clock's final current time is ", YY, "/", MM, "/", DD, &
         " ", H, ":", M, ":", S


! get the number of times the clock was advanced
call ESMF_ClockGet(clock, advanceCount=advanceCount, rc=rc)
print *, "The clock was advanced ", advanceCount, " times."
```

40.3.4 Clock reversal

This example shows how to time-step an `ESMF_Clock` in reverse mode.

```
call ESMF_ClockSet(clock, direction=ESMF_DIRECTION_REVERSE, rc=rc)


! time step clock in reverse from stop time back to start time;
!  note use of ESMF_ClockIsDone() rather than ESMF_ClockIsStopTime()
do while (.not.ESMF_ClockIsDone(clock, rc=rc))
```

```
    call ESMF_ClockPrint(clock, options="currTime string", rc=rc)

    call ESMF_ClockAdvance(clock, rc=rc)

  end do
```

40.3.5 Clock destruction

This example shows how to destroy an ESMF_Clock.

```
  ! destroy clock
  call ESMF_ClockDestroy(clock, rc=rc)


  ! finalize ESMF framework
  call ESMF_Finalize(rc=rc)


  end program ESMF_ClockEx
```

40.4 Restrictions and Future Work

1. **Alarm list allocation factor** The alarm list within a clock is dynamically allocated automatically, 200 alarm references at a time. This constant is defined in both Fortran and C++ with a #define for ease of modification.
2. **Clock variable timesteps in reverse**

   In order for a clock with variable timesteps to be run in ESMF_DIRECTION_REVERSE, the user must supply those timesteps to ESMF_ClockAdvance(). Essentially, the user must save the timesteps while in forward mode. In a future release, the Time Manager will assume this responsibility by saving the clock state (including the timeStep) at every timestep while in forward mode.

40.5 Class API

40.5.1 ESMF_ClockAssignment(=) - Assign a Clock to another Clock

INTERFACE:

```
   interface assignment(=)
   clock1 = clock2
```

ARGUMENTS:

```
   type(ESMF_Clock) :: clock1
   type(ESMF_Clock) :: clock2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign clock1 as an alias to the same ESMF_Clock object in memory as clock2. If clock2 is invalid, then clock1 will be equally invalid after the assignment.
The arguments are:

clock1 The ESMF_Clock object on the left hand side of the assignment.

clock2 The ESMF_Clock object on the right hand side of the assignment.

40.5.2 ESMF_ClockOperator(==) - Test if Clock 1 is equal to Clock 2

INTERFACE:

```
interface operator(==)
if (clock1 == clock2) then ... endif
             OR
result = (clock1 == clock2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Clock), intent(in) :: clock1
type(ESMF_Clock), intent(in) :: clock2
```

DESCRIPTION:

Overloads the (==) operator for the ESMF_Clock class. Compare two clocks for equality; return .true. if equal, .false. otherwise. Comparison is based on IDs, which are distinct for newly created clocks and identical for clocks created as copies.
If either side of the equality test is not in the ESMF_INIT_CREATED status an error will be logged. However, this does not affect the return value, which is .true. when both sides are in the *same* status, and .false. otherwise.
The arguments are:

clock1 The ESMF_Clock object on the left hand side of the equality operation.

clock2 The ESMF_Clock object on the right hand side of the equality operation.

40.5.3 ESMF_ClockOperator(/=) - Test if Clock 1 is not equal to Clock 2

INTERFACE:

```
interface operator(/=)
if (clock1 /= clock2) then ... endif
             OR
result = (clock1 /= clock2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Clock), intent(in) :: clock1
type(ESMF_Clock), intent(in) :: clock2
```

DESCRIPTION:

Overloads the (/=) operator for the ESMF_Clock class. Compare two clocks for inequality; return .true. if not equal, .false. otherwise. Comparison is based on IDs, which are distinct for newly created clocks and identical for clocks created as copies.
If either side of the equality test is not in the ESMF_INIT_CREATED status an error will be logged. However, this does not affect the return value, which is .true. when both sides are *not* in the *same* status, and .false. otherwise.
The arguments are:

clock1 The ESMF_Clock object on the left hand side of the non-equality operation.

clock2 The ESMF_Clock object on the right hand side of the non-equality operation.

40.5.4 ESMF_ClockAdvance - Advance a Clock's current time by one time step

INTERFACE:

```
      subroutine ESMF_ClockAdvance(clock, &
        timeStep, ringingAlarmList, ringingAlarmCount, rc)
```

ARGUMENTS:

```
      type(ESMF_Clock),        intent(inout)         :: clock
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_TimeInterval), intent(in),  optional :: timeStep
      type(ESMF_Alarm),        intent(out), optional :: ringingAlarmList(:)
      integer,                 intent(out), optional :: ringingAlarmCount
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Advances the clock's current time by one time step: either the clock's, or the passed-in timeStep (see below). When the clock is in ESMF_DIRECTION_FORWARD (default), this method adds the timeStep to the clock's current time. In ESMF_DIRECTION_REVERSE, timeStep is subtracted from the current time. In either case, timeStep can be positive or negative. See the "direction" argument in method ESMF_ClockSet(). ESMF_ClockAdvance() optionally returns a list and number of ringing ESMF_Alarms. See also method ESMF_ClockGetRingingAlarms().
The arguments are:

clock The object instance to advance.

[timeStep] Time step is performed with given timeStep, instead of the ESMF_Clock's. Does not replace the ESMF_Clock's timeStep; use ESMF_ClockSet(clock, timeStep, ...) for this purpose. Supports applications with variable time steps. timeStep can be positive or negative.

[ringingAlarmList] Returns the array of alarms that are ringing after the time step.

[ringingAlarmCount] The number of alarms ringing after the time step.

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

40.5.5 ESMF_ClockCreate - Create a new ESMF Clock

INTERFACE:

```
! Private name; call using ESMF_ClockCreate()
function ESMF_ClockCreateNew(timeStep, startTime, &
  stopTime, runDuration, runTimeStepCount, refTime, name, rc)
```

RETURN VALUE:

```
type(ESMF_Clock) :: ESMF_ClockCreateNew
```

ARGUMENTS:

```
      type(ESMF_TimeInterval), intent(in)            :: timeStep
      type(ESMF_Time),         intent(in)            :: startTime
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Time),         intent(in),  optional :: stopTime
      type(ESMF_TimeInterval), intent(in),  optional :: runDuration
      integer,                 intent(in),  optional :: runTimeStepCount
      type(ESMF_Time),         intent(in),  optional :: refTime
      character (len=*),       intent(in),  optional :: name
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

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

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

[stopTime] The `ESMF_Clock`'s stopping time. Can be greater than or less than the startTime, depending on a positive or negative timeStep, respectively. If neither stopTime, runDuration, nor runTimeStepCount is specified, clock runs "forever"; user must use other means to know when to stop (e.g. ESMF_Alarm or ESMF_ClockGet(clock, currTime)). Mutually exclusive with runDuration and runTimeStepCount.

[runDuration] Alternative way to specify `ESMF_Clock`'s stopping time; stopTime = startTime + runDuration. Can be positive or negative, consistent with the timeStep's sign. Mutually exclusive with stopTime and runTimeStepCount.

[runTimeStepCount] Alternative way to specify `ESMF_Clock`'s stopping time; stopTime = startTime + (runTimeStepCount * timeStep). stopTime can be before startTime if timeStep is negative. Mutually exclusive with stopTime and runDuration.

[refTime] The `ESMF_Clock`'s reference time. Provides reference point for simulation time (see currSimTime in ESMF_ClockGet() below).

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

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

40.5.6 ESMF_ClockCreate - Create a copy of an existing ESMF Clock

INTERFACE:

```
! Private name; call using ESMF_ClockCreate()
function ESMF_ClockCreateCopy(clock, rc)
```

RETURN VALUE:

```
type(ESMF_Clock) :: ESMF_ClockCreateCopy
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates a deep copy of a given ESMF_Clock, but does not copy its list of ESMF_Alarms (pointers), since an ESMF_Alarm can only be associated with one ESMF_Clock. Hence, the returned ESMF_Clock copy has no associated ESMF_Alarms, the same as with a newly created ESMF_Clock. If desired, new ESMF_Alarms must be created and associated with this copied ESMF_Clock via ESMF_AlarmCreate(), or existing ESMF_Alarms must be re-associated with this copied ESMF_Clock via ESMF_AlarmSet(...clock=...).
The arguments are:

clock The ESMF_Clock to copy.

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

40.5.7 ESMF_ClockDestroy - Release resources associated with a Clock

INTERFACE:

```
subroutine ESMF_ClockDestroy(clock, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Releases resources associated with this ESMF_Clock. This releases the list of associated ESMF_Alarms (pointers), but not the ESMF_Alarms themselves; the user must explicitly call ESMF_AlarmDestroy() on each ESMF_Alarm to release its resources. ESMF_ClockDestroy() and corresponding ESMF_AlarmDestroy()s can be called in either order.

If ESMF_ClockDestroy() is called before ESMF_AlarmDestroy(), any ESMF_Alarms that were in the ESMF_Clock's list will no longer be associated with any ESMF_Clock. If desired, these "orphaned" ESMF_Alarms can be associated with a different ESMF_Clock via a call to ESMF_AlarmSet(...clock=...).
The arguments are:

clock Release resources associated with this ESMF_Clock 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.

40.5.8 ESMF_ClockGet - Get a Clock's properties

INTERFACE:

```
      subroutine ESMF_ClockGet(clock, &
        timeStep, startTime, stopTime, &
        runDuration, runTimeStepCount, refTime, currTime, prevTime, &
        currSimTime, prevSimTime, calendar, calkindflag, timeZone, &
        advanceCount, alarmCount, direction, name, rc)
```

ARGUMENTS:

```
      type(ESMF_Clock),        intent(in)            :: clock
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_TimeInterval), intent(out), optional :: timeStep
      type(ESMF_Time),         intent(out), optional :: startTime
      type(ESMF_Time),         intent(out), optional :: stopTime
      type(ESMF_TimeInterval), intent(out), optional :: runDuration
      real(ESMF_KIND_R8),      intent(out), optional :: runTimeStepCount
      type(ESMF_Time),         intent(out), optional :: refTime
      type(ESMF_Time),         intent(out), optional :: currTime
      type(ESMF_Time),         intent(out), optional :: prevTime
      type(ESMF_TimeInterval), intent(out), optional :: currSimTime
      type(ESMF_TimeInterval), intent(out), optional :: prevSimTime
      type(ESMF_Calendar),     intent(out), optional :: calendar
      type(ESMF_CalKind_Flag), intent(out), optional :: calkindflag
      integer,                 intent(out), optional :: timeZone
      integer(ESMF_KIND_I8),   intent(out), optional :: advanceCount
      integer,                 intent(out), optional :: alarmCount
      type(ESMF_Direction_Flag),    intent(out), optional :: direction
      character (len=*),       intent(out), optional :: name
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

clock The object instance to query.

[timeStep] The ESMF_Clock's time step interval.

[startTime] The ESMF_Clock's starting time.

[stopTime] The ESMF_Clock's stopping time.

[runDuration] Alternative way to get ESMF_Clock's stopping time; runDuration = stopTime - startTime.

[runTimeStepCount] Alternative way to get ESMF_Clock's stopping time; runTimeStepCount = (stopTime - startTime) / timeStep.

[refTime] The ESMF_Clock's reference time.

[currTime] The ESMF_Clock's current time.

[prevTime] The ESMF_Clock's previous time. Equals currTime at the previous time step.

[currSimTime] The current simulation time (currTime - refTime).

[prevSimTime] The previous simulation time. Equals currSimTime at the previous time step.

[calendar] The Calendar on which all the Clock's times are defined.

[calkindflag] The CalKind_Flag on which all the Clock's times are defined.

[timeZone] The timezone within which all the Clock's times are defined.

[advanceCount] The number of times the ESMF_Clock has been advanced. Increments in ESMF_DIRECTION_FORWARD and decrements in ESMF_DIRECTION_REVERSE; see "direction" argument below and in ESMF_ClockSet().

[alarmCount] The number of ESMF_Alarms in the ESMF_Clock's ESMF_Alarm list.

[direction] The ESMF_Clock's time stepping direction. See also ESMF_ClockIsReverse(), an alternative for convenient use in "if" and "do while" constructs.

[name] The name of this clock.

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

40.5.9 ESMF_ClockGetAlarm - Get an Alarm in a Clock's Alarm list

INTERFACE:

```
      subroutine ESMF_ClockGetAlarm(clock, alarmname, alarm, &
        rc)
```

ARGUMENTS:

```
      type(ESMF_Clock),  intent(in)            :: clock
      character (len=*), intent(in)            :: alarmname
      type(ESMF_Alarm),  intent(out)           :: alarm
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the alarm whose name is the value of alarmname in the clock's ESMF_Alarm list.
The arguments are:

clock The object instance to get the `ESMF_Alarm` from.

alarmname The name of the desired `ESMF_Alarm`.

alarm The desired alarm.

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

40.5.10 ESMF_ClockGetAlarmList - Get a list of Alarms from a Clock

INTERFACE:

```
      subroutine ESMF_ClockGetAlarmList(clock, alarmlistflag, &
        timeStep, alarmList, alarmCount, rc)
```

ARGUMENTS:

```
      type(ESMF_Clock),           intent(in)            :: clock
      type(ESMF_AlarmList_Flag), intent(in)             :: alarmlistflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_TimeInterval),   intent(in),  optional :: timeStep
      type(ESMF_Alarm),          intent(out), optional :: alarmList(:)
      integer,                   intent(out), optional :: alarmCount
      integer,                   intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets the `clock`'s list of alarms and/or number of alarms.
The arguments are:

clock The object instance from which to get an `ESMF_Alarm` list and/or count of `ESMF_Alarm`s.

alarmlistflag The kind of list to get:

`ESMF_ALARMLIST_ALL` : Returns the `ESMF_Clock`'s entire list of alarms.

`ESMF_ALARMLIST_NEXTRINGING` : Return only those alarms that will ring upon the next `clock` time step. Can optionally specify argument `timeStep` (see below) to use instead of the `clock`'s. See also method `ESMF_AlarmWillRingNext()` for checking a single alarm.

`ESMF_ALARMLIST_PREVRINGING` :

Return only those alarms that were ringing on the previous `ESMF_Clock` time step. See also method `ESMF_AlarmWasPrevRinging()` for checking a single alarm.

`ESMF_ALARMLIST_RINGING` : Returns only those `clock` alarms that are currently ringing. See also method `ESMF_ClockAdvance()` for getting the list of ringing alarms subsequent to a time step. See also method `ESMF_AlarmIsRinging()` for checking a single alarm.

[timeStep] Optional time step to be used instead of the `clock`'s. Only used with `ESMF_ALARMLIST_NEXTRINGING alarmlistflag` (see above); ignored if specified with other `alarmlistflags`.

[alarmList] The array of returned alarms. If given, the array must be large enough to hold the number of alarms of the specified `alarmlistflag` in the specified `clock`.

[alarmCount] If specified, returns the number of ESMF_Alarms of the specified alarmlistflag in the specified clock.

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

40.5.11 ESMF_ClockGetNextTime - Calculate a Clock's next time

INTERFACE:

```
      subroutine ESMF_ClockGetNextTime(clock, nextTime, &
        timeStep, rc)
```

ARGUMENTS:

```
      type(ESMF_Clock),        intent(in)            :: clock
      type(ESMF_Time),         intent(out)           :: nextTime
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_TimeInterval), intent(in),  optional :: timeStep
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Calculates what the next time of the clock will be, based on the clock's current time step or an optionally passed-in timeStep.
The arguments are:

clock The object instance for which to get the next time.

nextTime The resulting ESMF_Clock's next time.

[timeStep] The time step interval to use instead of the clock's.

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

40.5.12 ESMF_ClockIsDone - Based on its direction, test if the Clock has reached or exceeded its stop time or start time

INTERFACE:

```
      function ESMF_ClockIsDone(clock, rc)
```

RETURN VALUE:

```
      logical :: ESMF_ClockIsDone
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns true if currentTime is greater than or equal to stopTime in `ESMF_DIRECTION_FORWARD`, or if currentTime is less than or equal to startTime in `ESMF_DIRECTION_REVERSE`. It returns false otherwise.
The arguments are:

clock The object instance to check.

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

40.5.13 ESMF_ClockIsReverse - Test if the Clock is in reverse mode

INTERFACE:

```
      function ESMF_ClockIsReverse(clock, rc)
```

RETURN VALUE:

```
      logical :: ESMF_ClockIsReverse
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns true if clock is in `ESMF_DIRECTION_REVERSE`, and false if in `ESMF_DIRECTION_FORWARD`. Allows convenient use in "if" and "do while" constructs. Alternative to `ESMF_ClockGet(...direction=...)`.
The arguments are:

clock The object instance to check.

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

40.5.14 ESMF_ClockIsStopTime - Test if the Clock has reached or exceeded its stop time

INTERFACE:

```
      function ESMF_ClockIsStopTime(clock, rc)
```

RETURN VALUE:

```
      logical :: ESMF_ClockIsStopTime
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns true if the `clock` has reached or exceeded its stop time, and false otherwise.
The arguments are:

clock The object instance to check.

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

40.5.15 ESMF_ClockIsStopTimeEnabled - Test if the Clock's stop time is enabled

INTERFACE:

```
      function ESMF_ClockIsStopTimeEnabled(clock, rc)
```

RETURN VALUE:

```
      logical :: ESMF_ClockIsStopTimeEnabled
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Returns true if the `clock`'s stop time is set and enabled, and false otherwise.
The arguments are:

clock The object instance to check.

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

40.5.16 ESMF_ClockPrint - Print the contents of a Clock

INTERFACE:

```
      subroutine ESMF_ClockPrint(clock, options, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Prints out an `ESMF_Clock`'s properties to `stdout`, in support of testing and debugging. The options control the type of information and level of detail.

The arguments are:

clock `ESMF_Clock` to be printed out.

[options] Print options. If none specified, prints all `clock` property values.
"advanceCount" - print the number of times the clock has been advanced.
"alarmCount" - print the number of alarms in the clock's list.
"alarmList" - print the clock's alarm list.
"currTime" - print the current clock time.
"direction" - print the clock's timestep direction.
"name" - print the clock's name.
"prevTime" - print the previous clock time.
"refTime" - print the clock's reference time.
"startTime" - print the clock's start time.
"stopTime" - print the clock's stop time.
"timeStep" - print the clock's time step.

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

40.5.17 ESMF_ClockSet - Set one or more properties of a Clock

INTERFACE:

```
subroutine ESMF_ClockSet(clock, &
  timeStep, startTime, stopTime, &
  runDuration, runTimeStepCount, refTime, currTime, advanceCount, &
  direction, name, rc)
```

ARGUMENTS:

```
       type(ESMF_Clock),        intent(inout)         :: clock
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_TimeInterval), intent(in),  optional :: timeStep
       type(ESMF_Time),         intent(in),  optional :: startTime
       type(ESMF_Time),         intent(in),  optional :: stopTime
       type(ESMF_TimeInterval), intent(in),  optional :: runDuration
       integer,                 intent(in),  optional :: runTimeStepCount
       type(ESMF_Time),         intent(in),  optional :: refTime
       type(ESMF_Time),         intent(in),  optional :: currTime
       integer(ESMF_KIND_I8),   intent(in),  optional :: advanceCount
       type(ESMF_Direction_Flag),    intent(in),  optional :: direction
       character (len=*),       intent(in),  optional :: name
       integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets/resets one or more of the properties of an `ESMF_Clock` that was previously initialized via `ESMF_ClockCreate()`.
The arguments are:

clock The object instance to set.

[timeStep] The `ESMF_Clock`'s time step interval, which can be positive or negative. This is used to change a clock's timestep property for those applications that need variable timesteps. See `ESMF_ClockAdvance()` below for specifying variable timesteps that are NOT saved as the clock's internal time step property. See "direction" argument below for behavior with
t ESMF_DIRECTION_REVERSE direction.

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

[stopTime] The `ESMF_Clock`'s stopping time. Can be greater than or less than the startTime, depending on a positive or negative timeStep, respectively. If neither stopTime, runDuration, nor runTimeStepCount is specified, clock runs "forever"; user must use other means to know when to stop (e.g. ESMF_Alarm or ESMF_ClockGet(clock, currTime)). Mutually exclusive with runDuration and runTimeStepCount.

[runDuration] Alternative way to specify `ESMF_Clock`'s stopping time; stopTime = startTime + runDuration. Can be positive or negative, consistent with the timeStep's sign. Mutually exclusive with stopTime and runTimeStepCount.

[runTimeStepCount] Alternative way to specify `ESMF_Clock`'s stopping time; stopTime = startTime + (runTimeStepCount * timeStep). stopTime can be before startTime if timeStep is negative. Mutually exclusive with stopTime and runDuration.

[refTime] The `ESMF_Clock`'s reference time. See description in `ESMF_ClockCreate()` above.

[currTime] The current time.

[advanceCount] The number of times the clock has been timestepped.

[direction] Sets the clock's time-stepping direction. If called with `ESMF_DIRECTION_REVERSE`, sets the clock in "reverse" mode, causing it to timestep back towards its startTime. If called with `ESMF_DIRECTION_FORWARD`, sets the clock in normal, "forward" mode, causing it to timestep in the direction of its startTime to stopTime. This holds true for negative timestep clocks as well, which are initialized (created) with stopTime < startTime. The default mode is `ESMF_DIRECTION_FORWARD`, established at `ESMF_ClockCreate()`. timeStep can also be specified as an argument at the same time, which allows for a change in magnitude and/or sign of the clock's timeStep. If not specified with `ESMF_DIRECTION_REVERSE`, the clock's current timeStep is effectively negated. If timeStep is specified, its sign is used as specified; it is not negated internally. E.g., if the specified timeStep is negative and the clock is placed in `ESMF_DIRECTION_REVERSE`, subsequent calls to `ESMF_ClockAdvance()` will cause the clock's current time to be decremented by the new timeStep's magnitude.

[name] The new name for this clock.

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

40.5.18 ESMF_ClockStopTimeDisable - Disable a Clock's stop time

INTERFACE:

```
      subroutine ESMF_ClockStopTimeDisable(clock, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Disables a `ESMF_Clock`'s stop time; `ESMF_ClockIsStopTime()` will always return false, allowing a clock to run past its stopTime.
The arguments are:

clock The object instance whose stop time to disable.

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

40.5.19 ESMF_ClockStopTimeEnable - Enable an Clock's stop time

INTERFACE:

```
      subroutine ESMF_ClockStopTimeEnable(clock, stopTime, rc)
```

ARGUMENTS:

```
      type(ESMF_Clock), intent(inout)         :: clock
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Time),  intent(in),  optional :: stopTime
      integer,          intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Enables a `ESMF_Clock`'s stop time, allowing `ESMF_ClockIsStopTime()` to respect the stopTime.
The arguments are:

clock The object instance whose stop time to enable.

[stopTime] The stop time to set or reset.

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

40.5.20 ESMF_ClockSyncToRealTime - Set Clock's current time to wall clock time

INTERFACE:

```
      subroutine ESMF_ClockSyncToRealTime(clock, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets a `clock`'s current time to the wall clock time. It is accurate to the nearest second.
The arguments are:

clock The object instance to be synchronized with wall clock time.

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

40.5.21 ESMF_ClockValidate - Validate a Clock's properties

INTERFACE:

```
      subroutine ESMF_ClockValidate(clock, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Checks whether a `clock` is valid. Must have a valid startTime and timeStep. If `clock` has a stopTime, its currTime must be within startTime to stopTime, inclusive; also startTime's and stopTime's calendars must be the same.
The arguments are:

clock `ESMF_Clock` to be validated.

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

41 Alarm Class

41.1 Description

The Alarm class identifies events that occur at specific Times or specific TimeIntervals by returning a true value at those times or subsequent times, and a false value otherwise.

41.2 Constants

41.2.1 ESMF_ALARMLIST

DESCRIPTION:
Specifies the characteristics of Alarms that populate a retrieved Alarm list.
The type of this flag is:
`type(ESMF_AlarmList_Flag)`
The valid values are:

ESMF_ALARMLIST_ALL All alarms.

ESMF_ALARMLIST_NEXTRINGING Alarms that will ring before or at the next timestep.

ESMF_ALARMLIST_PREVRINGING Alarms that rang at or since the last timestep.

ESMF_ALARMLIST_RINGING Only ringing alarms.

41.3 Use and Examples

Alarms are used in conjunction with Clocks (see Section 40.1). Multiple Alarms can be associated with a Clock. During the `ESMF_ClockAdvance()` method, a Clock iterates over its internal Alarms to determine if any are ringing. Alarms ring when a specified Alarm time is reached or exceeded, taking into account whether the time step is positive or negative. In `ESMF_DIRECTION_REVERSE` (see Section 40.1), alarms ring in reverse, i.e., they begin ringing when they originally ended, and end ringing when they originally began. On completion of the time advance call, the Clock optionally returns a list of ringing alarms.
Each ringing Alarm can then be processed using Alarm methods for identifying, turning off, disabling or resetting the Alarm.
Alarm methods are defined for obtaining the ringing state, turning the ringer on/off, enabling/disabling the Alarm, and getting/setting associated times.
The following example shows how to set and process Alarms.

```
! !PROGRAM: ESMF_AlarmEx - Alarm examples
!
! !DESCRIPTION:
!
! This program shows an example of how to create, initialize, and process
! alarms associated with a clock.
!-----------------------------------------------------------------------------

      ! ESMF Framework module
      use ESMF
      implicit none

      ! instantiate time_step, start, stop, and alarm times
      type(ESMF_TimeInterval) :: timeStep, alarmInterval
      type(ESMF_Time) :: alarmTime, startTime, stopTime

      ! instantiate a clock
      type(ESMF_Clock) :: clock
```

```
! instantiate Alarm lists
integer, parameter :: NUMALARMS = 2
type(ESMF_Alarm) :: alarm(NUMALARMS)

! local variables for Get methods
integer :: ringingAlarmCount  ! at any time step (0 to NUMALARMS)

! name, loop counter, result code
character (len=ESMF_MAXSTR) :: name
integer :: i, rc


! initialize ESMF framework
call ESMF_Initialize(defaultCalKind=ESMF_CALKIND_GREGORIAN, &
  defaultlogfilename="AlarmEx.Log", &
  logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

41.3.1 Clock initialization

This example shows how to create and initialize an ESMF_Clock.

```
! initialize time interval to 1 day
call ESMF_TimeIntervalSet(timeStep, d=1, rc=rc)


! initialize start time to 9/1/2003
call ESMF_TimeSet(startTime, yy=2003, mm=9, dd=1, rc=rc)


! initialize stop time to 9/30/2003
call ESMF_TimeSet(stopTime, yy=2003, mm=9, dd=30, rc=rc)


! create & initialize the clock with the above values
clock = ESMF_ClockCreate(timeStep, startTime, stopTime=stopTime, &
                         name="The Clock", rc=rc)
```

41.3.2 Alarm initialization

This example shows how to create and initialize two ESMF_Alarms and associate them with the clock.

```
! Initialize first alarm to be a one-shot on 9/15/2003 and associate
! it with the clock
call ESMF_TimeSet(alarmTime, yy=2003, mm=9, dd=15, rc=rc)


alarm(1) = ESMF_AlarmCreate(clock, &
   ringTime=alarmTime, name="Example alarm 1", rc=rc)


! Initialize second alarm to ring on a 1 week interval starting 9/1/2003
! and associate it with the clock
call ESMF_TimeSet(alarmTime, yy=2003, mm=9, dd=1, rc=rc)
```

```
call ESMF_TimeIntervalSet(alarmInterval, d=7, rc=rc)


! Alarm gets default name "Alarm002"
alarm(2) = ESMF_AlarmCreate(clock=clock, ringTime=alarmTime, &
                            ringInterval=alarmInterval, rc=rc)
```

41.3.3 Clock advance and Alarm processing

This example shows how to advance an ESMF_Clock and process any resulting ringing alarms.

```
! time step clock from start time to stop time
do while (.not.ESMF_ClockIsStopTime(clock, rc=rc))


  ! perform time step and get the number of any ringing alarms
  call ESMF_ClockAdvance(clock, ringingAlarmCount=ringingAlarmCount, &
                         rc=rc)


  call ESMF_ClockPrint(clock, options="currTime string", rc=rc)


  ! check if alarms are ringing
  if (ringingAlarmCount > 0) then
    print *, "number of ringing alarms = ", ringingAlarmCount

    do i = 1, NUMALARMS
      if (ESMF_AlarmIsRinging(alarm(i), rc=rc)) then


        call ESMF_AlarmGet(alarm(i), name=name, rc=rc)
        print *, trim(name), " is ringing!"


        ! after processing alarm, turn it off
        call ESMF_AlarmRingerOff(alarm(i), rc=rc)


      end if ! this alarm is ringing
    end do ! each ringing alarm
  endif ! ringing alarms
end do ! timestep clock
```

41.3.4 Alarm and Clock destruction

This example shows how to destroy ESMF_Alarms and ESMF_Clocks.

```
call ESMF_AlarmDestroy(alarm(1), rc=rc)


call ESMF_AlarmDestroy(alarm(2), rc=rc)


call ESMF_ClockDestroy(clock, rc=rc)
```

```
! finalize ESMF framework
call ESMF_Finalize(rc=rc)


end program ESMF_AlarmEx
```

41.4 Restrictions and Future Work

1. **Alarm list allocation factor** The alarm list within a clock is dynamically allocated automatically, 200 alarm references at a time. This constant is defined in both Fortran and C++ with a #define for ease of modification.

2. **Sticky alarm end times in reverse** For sticky alarms, there is an implicit limitation that in order to properly reverse timestep through a ring end time, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call `ESMF_AlarmRingerOff()`. An error message will be logged when this limitation is not satisfied.

3. **Sticky alarm ring interval in reverse**

   For repeating sticky alarms, it is currently assumed that the ringInterval is constant, so that only the time of the last call to `ESMF_AlarmRingerOff()` is saved. In `ESMF_DIRECTION_REVERSE`, this information is used to turn sticky alarms back on. In a future release, ringIntervals will be allowed to be variable, by saving alarm state at every timestep.

41.5 Design and Implementation Notes

The Alarm class is designed as a deep, dynamically allocatable class, based on a pointer type. This allows for both indirect and direct manipulation of alarms. Indirect alarm manipulation is where ESMF_Alarm API methods, such as ESMF_AlarmRingerOff(), are invoked on alarm references (pointers) returned from ESMF_Clock queries such as "return ringing alarms." Since the method is performed on an alarm reference, the actual alarm held by the clock is affected, not just a user's local copy. Direct alarm manipulation is the more common case where alarm API methods are invoked on the original alarm objects created by the user.
For consistency, the ESMF_Clock class is also designed as a deep, dynamically allocatable class.
An additional benefit from this approach is that Clocks and Alarms can be created and used from anywhere in a user's code without regard to the scope in which they were created. In constrast, statically created Alarms and Clocks would disappear if created within a user's routine that returns, whereas dynamically allocated Alarms and Clocks will persist until explicitly destroyed by the user.

41.6 Class API

41.6.1 ESMF_AlarmAssignment(=) - Assign an Alarm to another Alarm

INTERFACE:

```
interface assignment(=)
alarm1 = alarm2
```

ARGUMENTS:

```
type(ESMF_Alarm) :: alarm1
type(ESMF_Alarm) :: alarm2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign `alarm1` as an alias to the same `ESMF_Alarm` object in memory as `alarm2`. If `alarm2` is invalid, then `alarm1` will be equally invalid after the assignment.
The arguments are:

alarm1 The `ESMF_Alarm` object on the left hand side of the assignment.

alarm2 The `ESMF_Alarm` object on the right hand side of the assignment.

41.6.2 ESMF_AlarmOperator(==) - Test if Alarm 1 is equal to Alarm 2

INTERFACE:

```
interface operator(==)
if (alarm1 == alarm2) then ... endif
             OR
result = (alarm1 == alarm2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in) :: alarm1
type(ESMF_Alarm), intent(in) :: alarm2
```

DESCRIPTION:

Overloads the (==) operator for the `ESMF_Alarm` class. Compare two alarms for equality; return `.true.` if equal, `.false.` otherwise. Comparison is based on IDs, which are distinct for newly created alarms and identical for alarms created as copies.
If either side of the equality test is not in the `ESMF_INIT_CREATED` status an error will be logged. However, this does not affect the return value, which is `.true.` when both sides are in the *same* status, and `.false.` otherwise.
The arguments are:

alarm1 The `ESMF_Alarm` object on the left hand side of the equality operation.

alarm2 The `ESMF_Alarm` object on the right hand side of the equality operation.

41.6.3 ESMF_AlarmOperator(/=) - Test if Alarm 1 is not equal to Alarm 2

INTERFACE:

```
interface operator(/=)
if (alarm1 /= alarm2) then ... endif
             OR
result = (alarm1 /= alarm2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Alarm), intent(in) :: alarm1
type(ESMF_Alarm), intent(in) :: alarm2
```

DESCRIPTION:

Overloads the (/=) operator for the `ESMF_Alarm` class. Compare two alarms for inequality; return `.true.` if not equal, `.false.` otherwise. Comparison is based on IDs, which are distinct for newly created alarms and identical for alarms created as copies.
If either side of the equality test is not in the `ESMF_INIT_CREATED` status an error will be logged. However, this does not affect the return value, which is `.true.` when both sides are *not* in the *same* status, and `.false.` otherwise.
The arguments are:

alarm1 The `ESMF_Alarm` object on the left hand side of the non-equality operation.

alarm2 The `ESMF_Alarm` object on the right hand side of the non-equality operation.

41.6.4 ESMF_AlarmCreate - Create a new ESMF Alarm

INTERFACE:

```
! Private name; call using ESMF_AlarmCreate()
function ESMF_AlarmCreateNew(clock, &
  ringTime, ringInterval, stopTime, ringDuration, ringTimeStepCount, &
  refTime, enabled, sticky, name, rc)
```

RETURN VALUE:

```
type(ESMF_Alarm) :: ESMF_AlarmCreateNew
```

ARGUMENTS:

```
      type(ESMF_Clock),         intent(in)            :: clock
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Time),          intent(in),  optional :: ringTime
      type(ESMF_TimeInterval),  intent(in),  optional :: ringInterval
      type(ESMF_Time),          intent(in),  optional :: stopTime
      type(ESMF_TimeInterval),  intent(in),  optional :: ringDuration
      integer,                  intent(in),  optional :: ringTimeStepCount
      type(ESMF_Time),          intent(in),  optional :: refTime
      logical,                  intent(in),  optional :: enabled
      logical,                  intent(in),  optional :: sticky
      character (len=*),        intent(in),  optional :: name
      integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates and sets the initial values in a new `ESMF_Alarm`.
In `ESMF_DIRECTION_REVERSE` (see Section 40.1), alarms ring in reverse, i.e., they begin ringing when they originally ended, and end ringing when they originally began.
The arguments are:

clock The clock with which to associate this newly created alarm.

[ringTime] The ring time for a one-shot alarm or the first ring time for a repeating (interval) alarm. Must specify at least one of ringTime or ringInterval.

[ringInterval] The ring interval for repeating (interval) alarms. If `ringTime` is not also specified (first ring time), it will be calculated as the `clock`'s current time plus `ringInterval`. Must specify at least one of ringTime or ringInterval.

[stopTime] The stop time for repeating (interval) alarms. If not specified, an interval alarm will repeat forever.

[ringDuration] The absolute ring duration. If not sticky (see argument below), alarms rings for ringDuration, then turns itself off. Default is zero (unused). Mutually exclusive with ringTimeStepCount (below); used only if set to a non-zero duration and ringTimeStepCount is 1 (see below). See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[ringTimeStepCount] The relative ring duration. If not sticky (see argument below), alarms rings for ringTimeStepCount, then turns itself off. Default is 1: a non-sticky alarm will ring for one clock time step. Mutually exclusive with ringDuration (above); used if ringTimeStepCount > 1. If ringTimeStepCount is 1 (default) and ringDuration is non-zero, ringDuration is used (see above), otherwise ringTimeStepCount is used. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[refTime] The reference (i.e. base) time for an interval alarm.

[enabled] Sets the enabled state; default is on (true). If disabled, an alarm will not function at all. See also `ESMF_AlarmEnable()`, `ESMF_AlarmDisable()`.

[sticky] Sets the sticky state; default is on (true). If sticky, once an alarm is ringing, it will remain ringing until turned off manually via a user call to `ESMF_AlarmRingerOff()`. If not sticky, an alarm will turn itself off after a certain ring duration specified by either ringDuration or ringTimeStepCount (see above). There is an implicit limitation that in order to properly reverse timestep through a ring end time in `ESMF_DIRECTION_REVERSE`, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call `ESMF_AlarmRingerOff()`. An error message will be logged when this limitation is not satisfied. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

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

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

41.6.5 ESMF_AlarmCreate - Create a copy of an existing ESMF Alarm

INTERFACE:

```
! Private name; call using ESMF_AlarmCreate()
function ESMF_AlarmCreateCopy(alarm, rc)
```

RETURN VALUE:

```
type(ESMF_Alarm) :: ESMF_AlarmCreateCopy
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Creates a complete (deep) copy of a given ESMF_Alarm. The returned ESMF_Alarm copy is associated with the same ESMF_Clock as the original ESMF_Alarm. If desired, use ESMF_AlarmSet(...clock=...) to re-associate the ESMF_Alarm copy with a different ESMF_Clock.
The arguments are:

alarm The ESMF_Alarm to copy.

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

41.6.6 ESMF_AlarmDestroy - Release resources associated with an Alarm

INTERFACE:

```
      subroutine ESMF_AlarmDestroy(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Releases resources associated with this ESMF_Alarm. Also removes this ESMF_Alarm from its associated ESMF_Clock's list of ESMF_Alarms (removes the ESMF_Alarm pointer from the list).
The arguments are:

alarm Release resources associated with this ESMF_Alarm 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.

41.6.7 ESMF_AlarmDisable - Disable an Alarm

INTERFACE:

```
      subroutine ESMF_AlarmDisable(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Disables an `ESMF_Alarm`.
The arguments are:

alarm The object instance to disable.

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

41.6.8 ESMF_AlarmEnable - Enable an Alarm

INTERFACE:

```
      subroutine ESMF_AlarmEnable(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Enables an `ESMF_Alarm` to function.
The arguments are:

alarm The object instance to enable.

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

41.6.9 ESMF_AlarmGet - Get Alarm properties

INTERFACE:

```
      subroutine ESMF_AlarmGet(alarm, &
        clock, ringTime, prevRingTime, ringInterval, stopTime, ringDuration, &
        ringTimeStepCount, timeStepRingingCount, ringBegin, ringEnd, &
        refTime, ringing, ringingOnPrevTimeStep, enabled, sticky, name, rc)
```

ARGUMENTS:

```
      type(ESMF_Alarm),        intent(in)            :: alarm
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Clock),        intent(out), optional :: clock
      type(ESMF_Time),         intent(out), optional :: ringTime
      type(ESMF_Time),         intent(out), optional :: prevRingTime
      type(ESMF_TimeInterval), intent(out), optional :: ringInterval
```

```
    type(ESMF_Time),         intent(out), optional :: stopTime
    type(ESMF_TimeInterval), intent(out), optional :: ringDuration
    integer,                 intent(out), optional :: ringTimeStepCount
    integer,                 intent(out), optional :: timeStepRingingCount
    type(ESMF_Time),         intent(out), optional :: ringBegin
    type(ESMF_Time),         intent(out), optional :: ringEnd
    type(ESMF_Time),         intent(out), optional :: refTime
    logical,                 intent(out), optional :: ringing
    logical,                 intent(out), optional :: ringingOnPrevTimeStep
    logical,                 intent(out), optional :: enabled
    logical,                 intent(out), optional :: sticky
    character (len=*),       intent(out), optional :: name
    integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets one or more of an `ESMF_Alarm`'s properties.
The arguments are:

alarm The object instance to query.

[clock] The associated clock.

[ringTime] The ring time for a one-shot alarm or the next repeating alarm.

[prevRingTime] The previous ring time.

[ringInterval] The ring interval for repeating (interval) alarms.

[stopTime] The stop time for repeating (interval) alarms.

[ringDuration] The ring duration. Mutually exclusive with ringTimeStepCount (see below).

[ringTimeStepCount] The number of time steps comprising the ring duration. Mutually exclusive with ringDuration (see above).

[timeStepRingingCount] The number of time steps for which the alarm has been ringing thus far. Used internally for tracking ringTimeStepCount ring durations (see above). Mutually exclusive with ringBegin (see below). Increments in `ESMF_DIRECTION_FORWARD` and decrements in `ESMF_DIRECTION_REVERSE`; see Section 40.1.

[ringBegin] The time when the alarm began ringing. Used internally for tracking ringDuration (see above). Mutually exclusive with timeStepRingingCount (see above).

[ringEnd] The time when the alarm ended ringing. Used internally for re-ringing alarm in `ESMF_DIRECTION_REVERSE`.

[refTime] The reference (i.e. base) time for an interval alarm.

[ringing] The current ringing state. See also `ESMF_AlarmRingerOn()`, `ESMF_AlarmRingerOff()`.

[ringingOnPrevTimeStep] The ringing state upon the previous time step. Same as `ESMF_AlarmWasPrevRinging()`.

[enabled] The enabled state. See also `ESMF_AlarmEnable()`, `ESMF_AlarmDisable()`.

[sticky] The sticky state. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[name] The name of this alarm.

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

41.6.10 ESMF_AlarmIsEnabled - Check if Alarm is enabled

INTERFACE:

```
      function ESMF_AlarmIsEnabled(alarm, rc)
```

RETURN VALUE:

```
      logical :: ESMF_AlarmIsEnabled
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Check if `ESMF_Alarm` is enabled.
The arguments are:

alarm The object instance to check for enabled state.

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

41.6.11 ESMF_AlarmIsRinging - Check if Alarm is ringing

INTERFACE:

```
      function ESMF_AlarmIsRinging(alarm, rc)
```

RETURN VALUE:

```
      logical :: ESMF_AlarmIsRinging
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Check if `ESMF_Alarm` is ringing.
See also method `ESMF_ClockGetAlarmList(clock, ESMF_ALARMLIST_RINGING, ...)` to get a list of all ringing alarms belonging to an `ESMF_Clock`.
The arguments are:

alarm The alarm to check for ringing state.

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

41.6.12 ESMF_AlarmIsSticky - Check if Alarm is sticky

INTERFACE:

```
      function ESMF_AlarmIsSticky(alarm, rc)
```

RETURN VALUE:

```
      logical :: ESMF_AlarmIsSticky
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Check if `alarm` is sticky.
The arguments are:

alarm The object instance to check for sticky state.

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

41.6.13 ESMF_AlarmNotSticky - Unset an Alarm's sticky flag

INTERFACE:

```
      subroutine ESMF_AlarmNotSticky(alarm, &
        ringDuration, ringTimeStepCount, rc)
```

ARGUMENTS:

```
      type(ESMF_Alarm),        intent(inout)         :: alarm
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_TimeInterval), intent(in),  optional :: ringDuration
      integer,                 intent(in),  optional :: ringTimeStepCount
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Unset an `ESMF_Alarm`'s sticky flag; once alarm is ringing, it turns itself off after ringDuration.
The arguments are:

alarm The object instance to unset sticky.

[ringDuration] If not sticky, alarms rings for ringDuration, then turns itself off. Mutually exclusive with ringTimeStepCount (see below and full description in method `ESMF_AlarmCreate()` or `ESMF_AlarmSet()`).

[ringTimeStepCount] If not sticky, alarms rings for ringTimeStepCount, then turns itself off. Mutually exclusive with ringDuration (see above and full description in method `ESMF_AlarmCreate()` or `ESMF_AlarmSet()`).

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

41.6.14 ESMF_AlarmPrint - Print out an Alarm's properties

INTERFACE:

```
      subroutine ESMF_AlarmPrint(alarm, options, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Prints out an `ESMF_Alarm`'s properties to `stdout`, in support of testing and debugging. The options control the type of information and level of detail.

The arguments are:

alarm `ESMF_Alarm` to be printed out.

[options] Print options. If none specified, prints all `alarm` property values.
"clock" - print the associated clock's name.
"enabled" - print the alarm's ability to ring.
"name" - print the alarm's name.
"prevRingTime" - print the alarm's previous ring time.
"ringBegin" - print time when the alarm actually begins to ring.
"ringDuration" - print how long this alarm is to remain ringing.
"ringEnd" - print time when the alarm actually ends ringing.
"ringing" - print the alarm's current ringing state.
"ringingOnPrevTimeStep" - print whether the alarm was ringing immediately after the previous clock time step.
"ringInterval" - print the alarm's periodic ring interval.
"ringTime" - print the alarm's next time to ring.
"ringTimeStepCount" - print how long this alarm is to remain ringing, in terms of a number of clock time steps.
"refTime" - print the alarm's interval reference (base) time.
"sticky" - print whether the alarm must be turned off manually.
"stopTime" - print when alarm intervals end.
"timeStepRingingCount" - print the number of time steps the alarm has been ringing thus far.

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

41.6.15 ESMF_AlarmRingerOff - Turn off an Alarm

INTERFACE:

```
      subroutine ESMF_AlarmRingerOff(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Turn off an ESMF_Alarm; unsets ringing state. For a sticky alarm, this method must be called to turn off its ringing state. This is true for either ESMF_DIRECTION_FORWARD (default) or ESMF_DIRECTION_REVERSE. See Section 40.1.
The arguments are:

alarm The object instance to turn off.

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

41.6.16 ESMF_AlarmRingerOn - Turn on an Alarm

INTERFACE:

```
      subroutine ESMF_AlarmRingerOn(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Turn on an ESMF_Alarm; sets ringing state.
The arguments are:

alarm The object instance to turn on.

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

41.6.17 ESMF_AlarmSet - Set Alarm properties

INTERFACE:

```
      subroutine ESMF_AlarmSet(alarm, &
        clock, ringTime, ringInterval, stopTime, ringDuration, &
        ringTimeStepCount, refTime, ringing, enabled, sticky, name, rc)
```

ARGUMENTS:

```
      type(ESMF_Alarm),        intent(inout)         :: alarm
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Clock),        intent(in),  optional :: clock
      type(ESMF_Time),         intent(in),  optional :: ringTime
      type(ESMF_TimeInterval), intent(in),  optional :: ringInterval
      type(ESMF_Time),         intent(in),  optional :: stopTime
      type(ESMF_TimeInterval), intent(in),  optional :: ringDuration
      integer,                 intent(in),  optional :: ringTimeStepCount
      type(ESMF_Time),         intent(in),  optional :: refTime
      logical,                 intent(in),  optional :: ringing
      logical,                 intent(in),  optional :: enabled
      logical,                 intent(in),  optional :: sticky
      character (len=*),       intent(in),  optional :: name
      integer,                 intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets/resets one or more of the properties of an `ESMF_Alarm` that was previously initialized via `ESMF_AlarmCreate()`.
The arguments are:

alarm The object instance to set.

[clock] Re-associates this alarm with a different clock.

[ringTime] The next ring time for a one-shot alarm or a repeating (interval) alarm.

[ringInterval] The ring interval for repeating (interval) alarms.

[stopTime] The stop time for repeating (interval) alarms.

[ringDuration] The absolute ring duration. If not sticky (see argument below), alarms rings for ringDuration, then turns itself off. Default is zero (unused). Mutually exclusive with ringTimeStepCount (below); used only if set to a non-zero duration and ringTimeStepCount is 1 (see below). See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[ringTimeStepCount] The relative ring duration. If not sticky (see argument below), alarms rings for ringTimeStepCount, then turns itself off. Default is 1: a non-sticky alarm will ring for one clock time step. Mutually exclusive with ringDuration (above); used if ringTimeStepCount > 1. If ringTimeStepCount is 1 (default) and ringDuration is non-zero, ringDuration is used (see above), otherwise ringTimeStepCount is used. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[refTime] The reference (i.e. base) time for an interval alarm.

[ringing] Sets the ringing state. See also `ESMF_AlarmRingerOn()`, `ESMF_AlarmRingerOff()`.

[enabled] Sets the enabled state. If disabled, an alarm will not function at all. See also `ESMF_AlarmEnable()`, `ESMF_AlarmDisable()`.

[sticky] Sets the sticky state. If sticky, once an alarm is ringing, it will remain ringing until turned off manually via a user call to `ESMF_AlarmRingerOff()`. If not sticky, an alarm will turn itself off after a certain ring duration specified by either ringDuration or ringTimeStepCount (see above). There is an implicit limitation that in order to properly reverse timestep through a ring end time in `ESMF_DIRECTION_REVERSE`, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call `ESMF_AlarmRingerOff()`. An error message will be logged when this limitation is not satisfied. See also `ESMF_AlarmSticky()`, `ESMF_AlarmNotSticky()`.

[name] The new name for this alarm.

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

41.6.18 ESMF_AlarmSticky - Set an Alarm's sticky flag

INTERFACE:

```
      subroutine ESMF_AlarmSticky(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Set an `ESMF_Alarm`'s sticky flag; once alarm is ringing, it remains ringing until `ESMF_AlarmRingerOff()` is called. There is an implicit limitation that in order to properly reverse timestep through a ring end time in `ESMF_DIRECTION_REVERSE`, that time must have already been traversed in the forward direction. This is due to the fact that an `ESMF_Alarm` cannot predict when user code will call `ESMF_AlarmRingerOff()`. An error message will be logged when this limitation is not satisfied.
The arguments are:

alarm The object instance to be set sticky.

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

41.6.19 ESMF_AlarmValidate - Validate an Alarm's properties

INTERFACE:

```
      subroutine ESMF_AlarmValidate(alarm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Performs a validation check on an `ESMF_Alarm`'s properties. Must have a valid ringTime, set either directly or indirectly via ringInterval. See `ESMF_AlarmCreate()`.
The arguments are:

alarm `ESMF_Alarm` to be validated.

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

41.6.20 ESMF_AlarmWasPrevRinging - Check if Alarm was ringing on the previous Clock timestep

INTERFACE:

```
      function ESMF_AlarmWasPrevRinging(alarm, rc)
```

RETURN VALUE:

```
      logical :: ESMF_AlarmWasPrevRinging
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Check if `ESMF_Alarm` was ringing on the previous clock timestep.
See also method `ESMF_ClockGetAlarmList(clock, ESMF_ALARMLIST_PREVRINGING, ...)` get a list of all alarms belonging to a `ESMF_Clock` that were ringing on the previous time step.
The arguments are:

alarm The object instance to check for previous ringing state.

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

41.6.21 ESMF_AlarmWillRingNext - Check if Alarm will ring upon the next Clock timestep

INTERFACE:

```
      function ESMF_AlarmWillRingNext(alarm, timeStep, rc)
```

RETURN VALUE:

```
      logical :: ESMF_AlarmWillRingNext
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Check if `ESMF_Alarm` will ring on the next clock timestep, either the current clock timestep or a passed-in timestep. See also method `ESMF_ClockGetAlarmList(clock, ESMF_ALARMLIST_NEXTRINGING, ...)` to get a list of all alarms belonging to a `ESMF_Clock` that will ring on the next time step.
The arguments are:

alarm The alarm to check for next ringing state.

[timeStep] Optional timestep to use instead of the clock's.

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

42 Config Class

42.1 Description

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

42.1.1 Package history

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

42.1.2 Resource files

A *Resource File (RF)* is a text file consisting of list of *label-value* pairs. There is a limit of 250 characters per line and the Resource File can contain a maximum of 200 records. Each *label* should be followed by some data, the *value*. An example Resource File follows. It is the file used in the example below.

```
 # This is an example Resource File.
 # It contains a list of <label,value> pairs.
 # The colon after the label is required.

 # The values after the label can be an list.
 # Multiple types are authorized.
```

```
 my_file_names:         jan87.dat jan88.dat jan89.dat  # all strings
 constants:             3.1415   25                    # float and integer
 my_favorite_colors:    green blue 022


# Or, the data can be a list of single value pairs.
# It is simplier to retrieve data in this format:

 radius_of_the_earth:   6.37E6
 parameter_1:           89
 parameter_2:           78.2
 input_file_name:       dummy_input.netcdf


# Or, the data can be located in a table using the following
# syntax:

 my_table_name::
  1000     3000     263.0
   925     3000     263.0
   850     3000     263.0
   700     3000     269.0
   500     3000     287.0
   400     3000     295.8
   300     3000     295.8
 ::
```

Note that the colon after the label is required and that the double colon is required to declare tabular data.
Resource files are intended for random access (except between ::'s in a table definition). This means that order in which a particular *label-value* pair is retreived is not dependent upon the original order of the pairs. The only exception to this, however, is when the same *label* appears multiple times within the Resource File.

42.2 Use and Examples

This example/test code performs simple Config/Resource File routines. It does not include attaching a Config to a component. The important thing to remember there is that you can have one Config per component.
There are two methodologies for accessing data in a Resource File. This example will demonstrate both.
Note the API section contains a complete description of arguments in the methods/functions demonstrated in this example.

42.2.1 Variable declarations

The following are the variable declarations used as arguments in the following code fragments. They represent the locals names for the variables listed in the Resource File (RF). Note they do not need to be the same.

```
      character(ESMF_MAXSTR) :: fname ! config file name
      character*20  :: fn1, fn2, fn3, input_file ! strings to be read in
      integer       :: rc            ! error return code (0 is OK)
      integer       :: i_n           ! the first constant in the RF
      real          :: param_1       ! the second constant in the RF
      real          :: radius        ! radius of the earth
      real          :: table(7,3)    ! an array to hold the table in the RF

      type(ESMF_Config)   :: cf      ! the Config itself
```

42.2.2 Creation of a Config

While there are two methodologies for accessing the data within a Resource File, there is only one way to create the initial Config and load its ASCII text into memory. This is the first step in the process.
Note that subsequent calls to `ESMF_ConfigLoadFile` will OVERWRITE the current Config NOT append to it. There is no means of appending to a Config.

```
      cf = ESMF_ConfigCreate(rc=rc)             ! Create the empty Config


      fname = "myResourceFile.rc"                ! Name the Resource File
      call ESMF_ConfigLoadFile(cf, fname, rc=rc) ! Load the Resource File
                                                 ! into the empty Config
```

42.2.3 How to retrieve a label with a single value

The first method for retrieving information from the Resource File takes advantage of the <label,value> relationship within the file and access the data in a dictionary-like manner. This is the simplest methodology, but it does imply the use of only one value per label in the Resource File.
Remember, that the order in which a particular label/value pair is retrieved is not dependent upon the order which they exist within the Resource File.

```
    call ESMF_ConfigGetAttribute(cf, radius, label='radius_of_the_earth:', &
                                 default=1.0, rc=rc)
```

Note that the colon must be included in the label string when using this methodology. It is also important to provide a default value in case the label does not exist in the file
This methodology works for all types. The following is an example of retrieving a string:

```
    call ESMF_ConfigGetAttribute(cf, input_file, label='input_file_name:', &
                                 default="./default.nc", rc=rc)
```

The same code fragment can be used to demonstrate what happens when the label is not present. Note that "file_name" does not exist in the Resource File. The result of its abscense is the default value provided in the call.

```
    call ESMF_ConfigGetAttribute(cf, input_file, label='file_name:', &
                                 default="./default.nc", rc=rc)
```

42.2.4 How to retrieve a label with multiple values

When there are multiple, mixed-typed values associated with a label, the values can be retrieved in two steps: 1) Use ESMF_ConfigFindLabel() to find the label in the Config class; 2) use ESMF_ConfigGetAttribute() without the optional 'label' argument to retrieve the values one at a time, reading from left to right in the record.
A second reminder that the order in which a particular label/value pair is retrieved is not dependent upon the order which they exist within the Resource File. The label used in this method allows the user to skip to any point in the file.

```
      call ESMF_ConfigFindLabel(cf, 'constants:', rc=rc) ! Step a) Find the
                                                         ! label
```

Two constants, radius and i_n, can now be retrieved without having to specify their label or use an array. They are also different types.

```
call ESMF_ConfigGetAttribute(cf, param_1, rc=rc) ! Step b) read in the
                                                 ! first constant in
                                                 ! the sequence
call ESMF_ConfigGetAttribute(cf, i_n, rc=rc)     ! Step c) read in the
                                                 ! second constant in
                                                 ! the sequence
```

This methodology also works with strings.

```
call ESMF_ConfigFindLabel(cf, 'my_file_names:', &
        rc=rc)                       ! Step a) find the label


call ESMF_ConfigGetAttribute(cf, fn1, &
          rc=rc)                    ! Step b) retrieve the 1st filename
call ESMF_ConfigGetAttribute(cf, fn2, &
          rc=rc)                    ! Step c) retrieve the 2nd filename
call ESMF_ConfigGetAttribute(cf, fn3, &
          rc=rc)                    ! Step d) retrieve the 3rd filename
```

42.2.5 How to retrieve a table

To access tabular data, the user must use the multi-value method.

```
call ESMF_ConfigFindLabel(cf, 'my_table_name::', &
         rc=rc)        ! Step a) Set the label location to the
                       ! beginning of the table
```

Subsequently, `call ESMF_ConfigNextLine()` is used to move the location to the next row of the table. The example table in the Resource File contains 7 rows and 3 columns (7,3).

```
do i = 1, 7
  call ESMF_ConfigNextLine(cf, rc=rc) ! Step b) Increment the rows
  do j = 1, 3                         ! Step c) Fill in the table
    call ESMF_ConfigGetAttribute(cf, table(i,j), rc=rc)
  enddo
enddo
```

42.2.6 Destruction of a Config

The work with the configuration file `cf` is finalized by call to `ESMF_ConfigDestroy()`:

```
call ESMF_ConfigDestroy(cf, rc=rc) ! Destroy the Config
```

42.3 Class API

42.3.1 ESMF_ConfigAssignment(=) - Config assignment

INTERFACE:

```
interface assignment(=)
config1 = config2
```

ARGUMENTS:

```
type(ESMF_Config) :: config1
type(ESMF_Config) :: config2
```

DESCRIPTION:

Assign config1 as an alias to the same ESMF Config object in memory as config2. If config2 is invalid, then config1 will be equally invalid after the assignment.
The arguments are:

config1 The `ESMF_Config` object on the left hand side of the assignment.

config2 The `ESMF_Config` object on the right hand side of the assignment.

42.3.2 ESMF_ConfigCreate - Instantiate a Config object

INTERFACE:

```
  type(ESMF_Config) function ESMF_ConfigCreate(rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Instantiates an `ESMF_Config` object for use in subsequent calls.
The arguments are:

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

42.3.3 ESMF_ConfigDestroy - Destroy a Config object

INTERFACE:

```
subroutine ESMF_ConfigDestroy(config, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroys the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

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

42.3.4 ESMF_ConfigFindLabel - Find a label

INTERFACE:

```
     subroutine ESMF_ConfigFindLabel(config, label, rc)
```

ARGUMENTS:

```
       type(ESMF_Config), intent(inout)           :: config
       character(len=*),  intent(in)              :: label
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer,           intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Finds the `label` (key) string in the `config` object.
Since the search is done by looking for a string, possibly multi-worded, in the whole `Config` object, it is important to use special conventions to distinguish `labels` from other words. This is done in the Resource File by using the DAO convention to finish line labels with a (:) and table labels with a double colon (::).
The arguments are:

config Already created `ESMF_Config` object.

label Identifying label.

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

42.3.5 ESMF_ConfigGetAttribute - Get a value

INTERFACE:

```
        subroutine ESMF_ConfigGetAttribute(config, <value>, &
          label, default, rc)
```

ARGUMENTS:

```
        type(ESMF_Config), intent(inout)         :: config
        <value argument>, see below for supported values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        character(len=*),  intent(in),  optional :: label
        character(len=*),  intent(in),  optional :: default
        integer,           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets a value from the `config` object. When the value is a sequence of characters it will be terminated by the first white space.
Supported values for <value argument> are:

character(len=*), intent(out) :: value

real(ESMF_KIND_R4), intent(out) :: value

real(ESMF_KIND_R8), intent(out) :: value

integer(ESMF_KIND_I4), intent(out) :: value

integer(ESMF_KIND_I8), intent(out) :: value

logical, intent(out) :: value

The arguments are:

config Already created `ESMF_Config` object.

<value argument> Returned value.

[label] Identifing label.

[default] Default value if `label` is not found in `config` object.

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

42.3.6 ESMF_ConfigGetAttribute - Get a list of values

INTERFACE:

```
        subroutine ESMF_ConfigGetAttribute(config, <value list argument>, &
          count, label, default, rc)
```

ARGUMENTS:

```
        type(ESMF_Config), intent(inout)         :: config
        <value list argument>, see below for values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,           intent(in)   optional :: count
        character(len=*),  intent(in),  optional :: label
        character(len=*),  intent(in),  optional :: default
        integer,           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets a list of values from the `config` object.
Supported values for <value list argument> are:

real(ESMF_KIND_R4), intent(inout) :: valueList(:)

real(ESMF_KIND_R8), intent(inout) :: valueList(:)

integer(ESMF_KIND_I4), intent(inout) :: valueList(:)

integer(ESMF_KIND_I8), intent(inout) :: valueList(:)

logical, intent(inout) :: valueList(:)

The arguments are:

config Already created `ESMF_Config` object.

<value list argument> Returned value.

count Number of returned values expected.

[label] Identifing label.

[default] Default value if `label` is not found in `config` object.

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

42.3.7 ESMF_ConfigGetChar - Get a character

INTERFACE:

```
      subroutine ESMF_ConfigGetChar(config, value, &
        label, default, rc)
```

ARGUMENTS:

```
      type(ESMF_Config), intent(inout)         :: config
      character,         intent(out)           :: value
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*),  intent(in),  optional :: label
      character,         intent(in),  optional :: default
      integer,           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Gets a character `value` from the `config` object.
The arguments are:

config Already created `ESMF_Config` object.

value Returned value.

[label] Identifying label.

[default] Default value if label is not found in configuration object.

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

42.3.8 ESMF_ConfigGetDim - Get table sizes

INTERFACE:

```
    subroutine ESMF_ConfigGetDim(config, lineCount, columnCount, &
      label, rc)
```

ARGUMENTS:

```
      type(ESMF_Config), intent(inout)         :: config
      integer,           intent(out)           :: lineCount
      integer,           intent(out)           :: columnCount
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*),  intent(in),  optional :: label
      integer,           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

config Already created `ESMF_Config` object.

lineCount Returned number of lines in the table.

columnCount Returned maximum number of words in a table line.

[label] Identifying label (if present), otherwise current line.

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

42.3.9 ESMF_ConfigGetLen - Get the length of the line in words

INTERFACE:

```
    integer function ESMF_ConfigGetLen(config, label, rc)
```

ARGUMENTS:

```
      type(ESMF_Config), intent(inout)          :: config
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*),  intent(in),   optional :: label
      integer,           intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

config Already created `ESMF_Config` object.

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

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

42.3.10 ESMF_ConfigLoadFile - Load resource file into memory

INTERFACE:

```
     subroutine ESMF_ConfigLoadFile(config, filename, &
       delayout, unique, rc)
```

ARGUMENTS:

```
       type(ESMF_Config),   intent(inout)         :: config
       character(len=*),    intent(in)            :: filename
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_DELayout), intent(in),  optional :: delayout
       logical,             intent(in),  optional :: unique
       integer,             intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

config Already created `ESMF_Config` object.

filename Configuration file name.

[delayout] `ESMF_DELayout` associated with this `config` object.

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

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

42.3.11 ESMF_ConfigNextLine - Find next line

INTERFACE:

```
     subroutine ESMF_ConfigNextLine(config, tableEnd, rc)
```

ARGUMENTS:

```
       type(ESMF_Config), intent(inout)          :: config
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       logical,           intent(out),  optional :: tableEnd
       integer,           intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

config Already created `ESMF_Config` object.

[tableEnd] If specifed as `TRUE`, end of table mark (::) is checked.

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

42.3.12 ESMF_ConfigSetAttribute - Set a value

INTERFACE:

```
      subroutine ESMF_ConfigSetAttribute(config, <value argument>, &
        label, rc)
```

ARGUMENTS:

```
      type(ESMF_Config), intent(inout)           :: config
      <value argument>, see below for supported values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*),  intent(in),   optional  :: label
      integer,           intent(out),  optional  :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Sets a value in the `config` object.
Supported values for <value argument> are:

integer(ESMF_KIND_I4), intent(in) :: value

The arguments are:

config Already created `ESMF_Config` object.

<value argument> Value to set.

[label] Identifying attribute label.

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

42.3.13 ESMF_ConfigValidate - Validate a Config object

INTERFACE:

```
    subroutine ESMF_ConfigValidate(config, &
      options, rc)
```

ARGUMENTS:

```
      type(ESMF_Config), intent(inout)          :: config
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character (len=*), intent(in),   optional :: options
      integer,           intent(out),  optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

config `ESMF_Config` object to be validated.

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

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

43 Log Class

43.1 Description

The Log class consists of a variety of methods for writing error, warning, and informational messages to files. A default Log is created at ESMF initialization. Other Logs can be created later in the code by the user. Most Log methods take a Log as an optional argument and apply to the default Log when another Log is not specified. A set of standard return codes and associated messages are provided for error handling.

Log provides capabilities to store message entries in a buffer, which is flushed to a file, either when the buffer is full, or when the user calls an `ESMF_LogFlush()` method. Currently, the default is for the Log to flush after every ten entries. This can easily be changed by using the `ESMF_LogSet()` method and setting the `maxElements` property to another value. The `ESMF_LogFlush()` method is automatically called when the program exits by any means (program completion, halt on error, or when the Log is closed).

The user has the capability to abort the program on conditions such as an error or on a warning by using the `ESMF_LogSet()` method with the `logmsgAbort` argument. For example if the `logmsgAbort` array is set to `(ESMF_LOGMSG_ERROR,ESMF_LOGMSG_WARNING)`, the program will stop on any and all warning or errors. When the `logmsgAbort` argument is set to `ESMF_LOGMSG_ERROR`, the program will only abort on errors. Lastly, the user can choose to never abort by using `ESMF_LOGMSG_NONE`; this is the default.

Log will automatically put the PET number into the Log. Also, the user can either specify `ESMF_LOGKIND_SINGLE` which writes all the entries to a single Log or `ESMF_LOGKIND_MULTI` which writes entries to multiple Logs according to the PET number. To distinguish Logs from each other when using `ESMF_LOGKIND_MULTI`, the PET number (in the format `PETx.`) will be prepended to the file name where x is the PET number.

Opening multiple log files and writing log messages from all the processors may affect the application performance while running on a large number of processors. For that reason, `ESMF_LOGKIND_NONE` is provided to switch off the Log capability. All the Log methods have no effect in the `ESMF_LOGKIND_NONE` mode.

A tracing capability may be enabled by setting the `trace` flag by using the `ESMF_LogSet()` method. When tracing is enabled, calls to methods such as `ESMF_LogFoundError`, `ESMF_LogFoundAllocError`, and `ESMF_LogFoundDeallocError` are logged in the default log file. This can result in voluminous output. It is typically used only around areas of code which are being debugged.

Other options that are planned for Log are to adjust the verbosity of output, and to optionally write to `stdout` instead of file(s).

43.2 Constants

43.2.1 ESMF_LOGERR

The valid values are:

ESMF_LOGERR_PASSTHRU A named character constant, with a predefined generic error message, that can be used for the `msg` argument in any `ESMF_Log` routine. The message indicated by this named constant is "*Passing error in return code*."

43.2.2 ESMF_LOGKIND

DESCRIPTION:
Specifies a single log file, multiple log files (one per PET), or no log files.

The type of this flag is:
`type(ESMF_LogKind_Flag)`
The valid values are:

ESMF_LOGKIND_SINGLE Use a single log file, combining messages from all of the PETs. Not supported on some platforms.

ESMF_LOGKIND_MULTI Use multiple log files — one per PET. (Default.)

ESMF_LOGKIND_NONE Do not issue messages to a log file.

43.2.3 ESMF_LOGMSG

DESCRIPTION:

Specifies a message level
The type of this flag is:
`type(ESMF_LogMsg_Flag)`
The valid values are:

ESMF_LOGMSG_INFO Informational messages

ESMF_LOGMSG_WARNING Warning messages

ESMF_LOGMSG_ERROR Error messages

ESMF_LOGMSG_TRACE Trace messages

Valid predefined named array constant values are:

ESMF_LOGMSG_ALL All messages

ESMF_LOGMSG_NONE No messages

ESMF_LOGMSG_NOTRACE All messages EXCEPT trace messages

43.3 Use and Examples

By default `ESMF_Initialize()` opens a default Log in `ESMF_LOGKIND_MULTI` mode. ESMF handles the initialization and finalization of the default Log so the user can immediately start using it. If additional Log objects are desired, they must be explicitly created or opened using `ESMF_LogOpen()`.
`ESMF_LogOpen()` requires a Log object and filename argument. Additionally, the user can specify single or multi Logs by setting the `logkindflag` property to `ESMF_LOGKIND_SINGLE` or `ESMF_LOGKIND_MULTI`. This is useful as the PET numbers are automatically added to the Log entries. A single Log will put all entries, regardless of PET number, into a single log while a multi Log will create multiple Logs with the PET number prepended to the filename and all entries will be written to their corresponding Log by their PET number.
By default, the Log file is not truncated at the start of a new run; it just gets appended each time. Future functionality may include an option to either truncate or append to the Log file.
In all cases where a Log is opened, a Fortran unit number is assigned to a specific Log. A Log is assigned an unused unit number using the algorithm described in the `ESMF_IOUnitGet()` method.
The user can then set or get options on how the Log should be used with the `ESMF_LogSet()` and `ESMF_LogGet()` methods. These are partially implemented at this time.
Depending on how the options are set, `ESMF_LogWrite()` either writes user messages directly to a Log file or writes to a buffer that can be flushed when full or by using the `ESMF_LogFlush()` method. The default is to flush after every ten entries because `maxElements` is initialized to ten (which means the buffer reaches its full state after every ten writes and then flushes).
A message filtering option may be set with `ESMF_LogSet()` so that only selected message types are actually written to the log. One key use of this feature is to allow placing informational log write requests into the code for debugging

or tracing. Then, when the informational entries are not needed, the messages at that level may be turned off — leaving only warning and error messages in the logs.
For every ESMF_LogWrite(), a time and date stamp is prepended to the Log entry. The time is given in microsecond precision. The user can call other methods to write to the Log. In every case, all methods eventually make a call implicitly to ESMF_LogWrite() even though the user may never explicitly call it.
When calling ESMF_LogWrite(), the user can supply an optional line, file and method. These arguments can be passed in explicitly or with the help of cpp macros. In the latter case, a define for an ESMF_FILENAME must be placed at the beginning of a file and a define for ESMF_METHOD must be placed at the beginning of each method. The user can then use the ESMF_CONTEXT cpp macro in place of line, file and method to insert the parameters into the method. The user does not have to specify line number as it is a value supplied by cpp.
An example of Log output is given below running with logkindflag property set to ESMF_LOGKIND_MULTI (default) using the default Log:
(Log file PET0.ESMF_LogFile)

```
20041105 163418.472210 INFO      PET0     Running with ESMF Version 2.2.1
```

(Log file PET1.ESMF_LogFile)

```
20041105 163419.186153 ERROR     PET1     ESMF_Field.F90             812
ESMF_FieldGet No Grid or Bad Grid attached to Field
```

The first entry shows date and time stamp. The time is given in microsecond precision. The next item shown is the type of message (INFO in this case). Next, the PET number is added. Lastly, the content is written.
The second entry shows something slightly different. In this case, we have an ERROR. The method name (ESMF_Field.F90) is automatically provided from the cpp macros as well as the line number (812). Then the content of the message is written.
When done writing messages, the default Log is closed by calling ESMF_LogFinalize() or ESMF_LogClose() for user created Logs. Both methods will release the assigned unit number.

```
! !PROGRAM: ESMF_LogErrEx - Log Error examples
!
! !DESCRIPTION:
!
! This program shows examples of Log Error writing
!-----------------------------------------------------------------------------

! Macros for cpp usage
! File define
#define ESMF_FILENAME "ESMF_LogErrEx.F90"
! Method define
#define ESMF_METHOD "program ESMF_LogErrEx"
#include "ESMF_LogMacros.inc"

    ! ESMF Framework module
    use ESMF
    implicit none

    ! return variables
    integer :: rc1, rc2, rc3, rcToTest, allocRcToTest
    type(ESMF_LOG) :: alog  ! a log object that is not the default log
    type(ESMF_LogKind_Flag) :: logkindflag
    type(ESMF_Time) :: time
    integer, pointer :: intptr(:)
```

43.3.1 Default Log

This example shows how to use the default Log. This example does not use cpp macros but does use multi Logs. A separate Log will be created for each PET.

```
! Initialize ESMF to initialize the default Log
call ESMF_Initialize(rc=rc1, logkindflag=ESMF_LOGKIND_MULTI)


! LogWrite
call ESMF_LogWrite("Log Write 2", ESMF_LOGMSG_INFO, rc=rc2)


! LogMsgSetError
call ESMF_LogSetError(ESMF_RC_OBJ_BAD, msg="Convergence failure", &
                         rcToReturn=rc2)
! LogMsgFoundError
call ESMF_TimeSet(time, calkindflag=ESMF_CALKIND_NOCALENDAR)
call ESMF_TimeSyncToRealTime(time, rc=rcToTest)
if (ESMF_LogFoundError(rcToTest, msg="getting wall clock time", &
                          rcToReturn=rc2)) then
    ! Error getting time. The previous call will have printed the error
    ! already into the log file.  Add any additional error handling here.
    ! (This call is expected to provoke an error from the Time Manager.)
endif

! LogMsgFoundAllocError
allocate(intptr(10), stat=allocRcToTest)
if (ESMF_LogFoundAllocError(allocRcToTest, msg="integer array", &
                               rcToReturn=rc2)) then
    ! Error during allocation.  The previous call will have logged already
    ! an error message into the log.
endif
deallocate(intptr)
```

43.3.2 User created Log

This example shows how to use a user created Log. This example uses cpp macros.

```
! Open a Log named "Testlog.txt" associated with alog.
call ESMF_LogOpen(alog, "TestLog.txt", rc=rc1)


! LogWrite
call ESMF_LogWrite("Log Write 2", ESMF_LOGMSG_INFO, &
                   line=__LINE__, file=ESMF_FILENAME, &
                   method=ESMF_METHOD, log=alog, rc=rc2)


! LogMsgSetError
call ESMF_LogSetError(ESMF_RC_OBJ_BAD,  msg="Interpolation Failure", &
                      line=__LINE__, file=ESMF_FILENAME, &
                       method=ESMF_METHOD, rcToReturn=rc2, log=alog)
```

43.3.3 Get and Set

This example shows how to use Get and Set routines, on both the default Log and the user created Log from the previous examples.

```
! This is an example showing a query of the default Log.  Please note that
! no Log is passed in the argument list, so the default Log will be used.
call ESMF_LogGet(logkindflag=logkindflag, rc=rc3)


! This is an example setting a property of a Log that is not the default.
! It was opened in a previous example, and the handle for it must be
! passed in the argument list.
call ESMF_LogSet(log=alog, logmsgAbort=(/ESMF_LOGMSG_ERROR/), rc=rc2)


! Close the user log.
call ESMF_LogClose(alog, rc=rc3)


! Finalize ESMF to close the default log
call ESMF_Finalize(rc=rc1)
```

43.4 Restrictions and Future Work

1. **Line, file and method are only available when using the C preprocessor** Message writing methods are expanded using the ESMF macro ESMF_CONTEXT that adds the predefined symbolic constants __LINE__ and __FILE__ (or the ESMF constant ESMF_FILENAME if defined) and the ESMF constant ESMF_METHOD to the argument list. Using these constants, we can associate a file name, line number and method name with the message. If the CPP preprocessor is not used, this expansion will not be done and hence the ESMF macro ESMF_CONTEXT can not be used, leaving the file name, line number and method out of the Log text.

2. **Get and set methods are partially implemented.** Currently, the `ESMF_LogGet()` and `ESMF_LogSet()` methods are partially implemented.

3. **Log only appends entries.** All writing to the Log is appended rather than overwriting the Log. Future enhancements include the option to either append to an existing Log or overwrite the existing Log.

4. **Avoiding conflicts with the default Log.**

   The private methods `ESMF_LogInitialize()` and `ESMF_LogFinalize()` are called during `ESMF_Initialize()` and `ESMF_Finalize()` respectively, so they do not need to be called if the default Log is used. If a new Log is required, `ESMF_LogOpen()` is used with a new Log object passed in so that there are no conflicts with the default Log.

5. **ESMF_LOGKIND_SINGLE does not work properly.** When the `ESMF_LogKind_Flag` is set to `ESMF_LOGKIND_SINGLE`, different system may behave differently. The log messages from some processors may be lost or overwritten by other processors. Users are advised not to use this mode. The MPI-based I/O will be implemented to fix the problem in the future release.

43.5 Design and Implementation Notes

1. The Log class was implemented in Fortran and uses the Fortran I/O libraries when the class methods are called from Fortran. The C/C++ Log methods use the Fortran I/O library by calling utility functions that are written in Fortran. These utility functions call the standard Fortran write, open and close functions. At initialization an `ESMF_LOG` is created. The `ESMF_LOG` stores information for a specific Log file. When working with more than one Log file, multiple `ESMF_LOG`'s are required (one `ESMF_LOG` for each Log file). For each Log, a handle is returned through the `ESMF_LogInitialize` method for the default log or `ESMF_LogOpen` for a user created log. The user can specify single or multi logs by setting the `logkindflag` property in the `ESMF_LogInitialize` or `ESMF_Open` method to `ESMF_LOGKIND_SINGLE` or `ESMF_LOGKIND_MULTI`. Similarly, the user can set the `logkindflag`

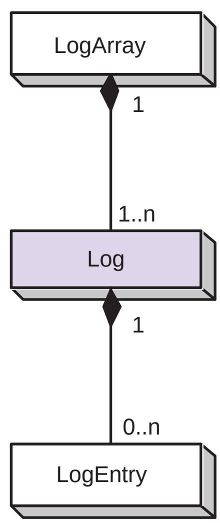
property for the default Log with the `ESMF_Initialize` method call. The `logkindflag` is useful as the PET numbers are automatically added to the log entries. A single log will put all entries, regardless of PET number, into a single log while a multi log will create multiple logs with the PET number prepended to the filename and all entries will be written to their corresponding log by their PET number.

The properties for a Log are set with the `ESMF_LogSet()` method and retrieved with the `ESMF_LogGet()` method.

Additionally, buffering is enabled. Buffering allows `ESMF` to manage output data streams in a desired way. Writing to the buffer is transparent to the user because all the Log entries are handled automatically by the `ESMF_LogWrite()` method. All the user has to do is specify the buffer size (the default is ten) by setting the `maxElements` property. Every time the `ESMF_LogWrite()` method is called, a LogEntry element is populated with the `ESMF_LogWrite()` information. When the buffer is full (i.e., when all the LogEntry elements are populated), the buffer will be flushed and all the contents will be written to file. If buffering is not needed, that is `maxElements=1` or `flushImmediately=ESMF_TRUE`, the `ESMF_LogWrite()` method will immediately write to the Log file(s).

43.6 Object Model

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



43.7 Class API

43.7.1 ESMF_LogAssignment(=) - Log assignment

INTERFACE:

```
interface assignment(=)
log1 = log2
```

ARGUMENTS:

```
type(ESMF_Log) :: log1
type(ESMF_Log) :: log2
```

DESCRIPTION:

Assign log1 as an alias to the same ESMF Log object in memory as log2. If log2 is invalid, then log1 will be equally invalid after the assignment.
The arguments are:

log1 The ESMF_Log object on the left hand side of the assignment.

log2 The ESMF_Log object on the right hand side of the assignment.

43.7.2 ESMF_LogClose - Close Log file(s)

INTERFACE:

```
      subroutine ESMF_LogClose(log, rc)
```

ARGUMENTS:

```
      type(ESMF_Log), intent(inout)         :: log
 type(ESMF_KeywordEnforcer), optional :: keywordEnforcer ! must use keywords below
      integer,        intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This routine closes the user log file(s) associated with log. If the log is not explicitly closed, it will be closed by ESMF_Finalize.
The arguments are:

log An ESMF_Log object.

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

43.7.3 ESMF_LogFlush - Flush the Log file(s)

INTERFACE:

```
      subroutine ESMF_LogFlush(log, rc)
```

ARGUMENTS:

```
 type(ESMF_KeywordEnforcer), optional :: keywordEnforcer ! must use keywords below
      type(ESMF_Log), intent(inout), optional :: log
      integer,        intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This subroutine flushes the file buffer associated with log.
The arguments are:

[log] An optional ESMF_Log object that can be used instead of the default Log.

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

43.7.4 ESMF_LogFoundAllocError - Check Fortran allocation status error and write message

INTERFACE:

```
      function ESMF_LogFoundAllocError(statusToCheck,  &
                                       msg,line,file, &
                                       method,rcToReturn,log)
```

RETURN VALUE:

```
      logical                                     :: ESMF_LogFoundAllocError
```

ARGUMENTS:

```
      integer,          intent(in)              :: statusToCheck
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*), intent(in),    optional :: msg
      integer,          intent(in),    optional :: line
      character(len=*), intent(in),    optional :: file
      character(len=*), intent(in),    optional :: method
      integer,          intent(out),   optional :: rcToReturn
      type(ESMF_Log),   intent(inout), optional :: log
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This function returns `.true.` when a Fortran status code returned from a memory allocation indicates an allocation error. An ESMF predefined memory allocation error message will be added to the `ESMF_Log` along with a user added `msg`, `line`, `file` and `method`. Additionally, statusToCheck will be converted to `rcToReturn`.
The arguments are:

statusToCheck Fortran allocation status to check.

[msg] User-provided message string.

[line] Integer source line number. Expected to be set by using the preprocessor `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, set the `rcToReturn` value to `ESMF_RC_MEM` which is the error code for a memory allocation eror.

[log] An optional `ESMF_Log` object that can be used instead of the default Log.

43.7.5 ESMF_LogFoundDeallocError - Check Fortran deallocation status error and write message

INTERFACE:

```
      function ESMF_LogFoundDeallocError(statusToCheck,  &
                                         msg,line,file, &
                                         method,rcToReturn,log)
```

RETURN VALUE:

```
logical ::ESMF_LogFoundDeallocError
```

ARGUMENTS:

```
      integer,           intent(in)              :: statusToCheck
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*), intent(in),    optional :: msg
      integer,           intent(in),    optional :: line
      character(len=*), intent(in),    optional :: file
      character(len=*), intent(in),    optional :: method
      integer,           intent(out),   optional :: rcToReturn
      type(ESMF_Log),    intent(inout), optional :: log
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This function returns `.true.` when a Fortran status code returned from a memory deallocation indicates an deallocation error. An ESMF predefined memory deallocation error message will be added to the `ESMF_Log` along with a user added `msg`, `line`, `file` and `method`. Additionally, statusToCheck will be converted to `rcToReturn`.
The arguments are:

statusToCheck Fortran deallocation status to check.

[msg] User-provided message string.

[line] Integer source line number. Expected to be set by using the preprocessor `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, set the `rcToReturn` value to `ESMF_RC_MEM` which is the error code for a memory allocation eror.

[log] An optional `ESMF_Log` object that can be used instead of the default Log.

43.7.6 ESMF_LogFoundError - Check ESMF return code for error and write message

INTERFACE:

```
      function ESMF_LogFoundError(rcToCheck,   &
                                  msg, line, file, method, &
                                  rcToReturn, log)
```

RETURN VALUE:

```
      logical :: ESMF_LogFoundError
```

ARGUMENTS:

```
      integer,           intent(in),    optional :: rcToCheck
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*), intent(in),    optional :: msg
      integer,           intent(in),    optional :: line
      character(len=*), intent(in),    optional :: file
      character(len=*), intent(in),    optional :: method
      integer,           intent(out),   optional :: rcToReturn
      type(ESMF_Log),   intent(inout), optional :: log
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This function returns `.true.` for ESMF return codes that indicate an error. A predefined error message will added to the `ESMF_Log` along with a user added `msg`, `line`, `file` and `method`. Additionally, `rcToReturn` is set to `rcToCheck`.
The arguments are:

[rcToCheck] Return code to check. Default is `ESMF_SUCCESS`.

[msg] User-provided message string.

[line] Integer source line number. Expected to be set by using the preprocessor `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, copy the `rcToCheck` value to `rc`. This is not the return code for this function; it allows the calling code to do an assignment of the error code at the same time it is testing the value.

[log] An optional `ESMF_Log` object that can be used instead of the default Log.

43.7.7 ESMF_LogOpen - Open Log file(s)

INTERFACE:

```
subroutine ESMF_LogOpen(log, filename, logkindflag, rc)
```

ARGUMENTS:

```
type(ESMF_Log),          intent(inout)          :: log
character(len=*),        intent(in)             :: filename
type(ESMF_LogKind_Flag), intent(in),  optional :: logkindflag
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

This routine opens a file named `filename` and associates it with the `ESMF_Log`. If the incoming log is already a valid Log object, no new Log is opened and the Log argument remains unchanged.
The arguments are:

log An `ESMF_Log` object.

filename Name of log file to be opened.

[logkindflag] Set the logkindflag. See section 43.2.2 for a list of valid options. If not specified, defaults to `ESMF_LOGKIND_MULTI`.

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

43.7.8 ESMF_LogSet - Set Log parameters

INTERFACE:

```
  subroutine ESMF_LogSet(log, verbose, flush, rootOnly,  &
                          logmsgAbort, stream, maxElements, logmsgList,  &
                          errorMask, trace, rc)
```

ARGUMENTS:

```
      type(ESMF_Log),      intent(inout), optional :: log
      logical,             intent(in),    optional :: verbose
      logical,             intent(in),    optional :: flush
      logical,             intent(in),    optional :: rootOnly
      type(ESMF_LogMsg_Flag), intent(in), optional :: logmsgAbort(:)
      integer,             intent(in),    optional :: stream
      integer,             intent(in),    optional :: maxElements
      type(ESMF_LogMsg_Flag), intent(in), optional :: logmsgList(:)
      integer,             intent(in),    optional :: errorMask(:)
      logical,             intent(in),    optional :: trace
      integer,             intent(out),   optional :: rc
```

DESCRIPTION:

This subroutine sets the properties for the Log object.
The arguments are:

[log] An optional `ESMF_Log` object that can be used instead of the default Log.

[verbose] Verbose flag.

[rootOnly] Root only flag.

[logmsgAbort] Sets the condition on which ESMF aborts. The array can contain any combination of ESMF_LOGMSG named constants. These named constants are described in section 43.2.3.

[stream] The type of stream, with the following valid values and meanings:

0 free;

1 preordered.

[maxElements] Maximum number of elements in the Log.

[logmsgList] An array of message types that will be logged. Log write requests not matching the list will be ignored. By default all messages will be logged. If an empty array is provided, no messages will be logged. See section 43.2.3 for a list of valid message types.

[errorMask] List of error codes that will *not* be logged as errors.

[trace] If set to true, calls such as `ESMF_LogFoundError`, `ESMF_LogFoundAllocError`, and `ESMF_LogFoundDeallocError` will be logged as a tool for program flow tracing. This may generate voluminous output in the log.

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

43.7.9 ESMF_LogSetError - Set ESMF return code for error and write msg

INTERFACE:

```
      subroutine ESMF_LogSetError(rcToCheck,  &
                                  msg, line, file, method, &
                                  rcToReturn, log)
```

ARGUMENTS:

```
      integer,          intent(in)              :: rcToCheck
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      character(len=*), intent(in),    optional :: msg
      integer,          intent(in),    optional :: line
      character(len=*), intent(in),    optional :: file
      character(len=*), intent(in),    optional :: method
      integer,          intent(out),   optional :: rcToReturn
      type(ESMF_Log),   intent(inout), optional :: log
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This subroutine sets the `rcToReturn` value to `rcToCheck` if `rcToReturn` is present and writes this error code to the `ESMF_Log` if an error is generated. A predefined error message will added to the `ESMF_Log` along with a user added `msg`, `line`, `file` and `method`.
The arguments are:

rcToCheck rc value for set

[msg] User-provided message string.

[line] Integer source line number. Expected to be set by using the preprocessor macro `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[rcToReturn] If specified, copy the `rcToCheck` value to `rcToreturn`. This is not the return code for this function; it allows the calling code to do an assignment of the error code at the same time it is testing the value.

[log] An optional `ESMF_Log` object that can be used instead of the default Log.

43.7.10 ESMF_LogWrite - Write to Log file(s)

INTERFACE:

```
      recursive subroutine ESMF_LogWrite(msg, logmsgList,  &
                                         line, file, method, log, rc)
```

ARGUMENTS:

```
      character(len=*),   intent(in)             :: msg
      type(ESMF_LogMsg_Flag), intent(in)         :: logmsgList
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,            intent(in),    optional :: line
      character(len=*),   intent(in),    optional :: file
      character(len=*),   intent(in),    optional :: method
      type(ESMF_Log),     intent(inout), optional :: log
      integer,            intent(out),   optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This subroutine writes to the file associated with an `ESMF_Log`. A message is passed in along with the `logmsgList`, `line`, `file` and `method`. If the write to the `ESMF_Log` is successful, the function will return a logical `true`. This function is the base function used by all the other `ESMF_Log` writing methods.
The arguments are:

msg User-provided message string.

logmsgList The type of message. See Section 43.2.3 for possible values.

[line] Integer source line number. Expected to be set by using the preprocessor macro `__LINE__` macro.

[file] User-provided source file name.

[method] User-provided method string.

[log] An optional `ESMF_Log` object that can be used instead of the default Log.

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

44 DELayout Class

44.1 Description

The DELayout class provides an additional layer of abstraction on top of the Virtual Machine (VM) layer. DELayout does this by introducing DEs (Decomposition Elements) as logical resource units. The DELayout object keeps track of the relationship between its DEs and the resources of the associated VM object.
The relationship between DEs and VM resources (PETs (Persistent Execution Threads) and VASs (Virtual Address Spaces)) contained in a DELayout object is defined during its creation and cannot be changed thereafter. There are, however, a number of hint and specification arguments that can be used to shape the DELayout during its creation.
Contrary to the number of PETs and VASs contained in a VM object, which are fixed by the available resources, the number of DEs contained in a DELayout can be chosen freely to best match the computational problem or other design criteria. Creating a DELayout with less DEs than there are PETs in the associated VM object can be used to share resources between decomposed objects within an ESMF component. Creating a DELayout with more DEs than there are PETs in the associated VM object can be used to evenly partition the computation over the available resources.

The simplest case, however, is where the DELayout contains the same number of DEs as there are PETs in the associated VM context. In this case the DELayout may be used to re-label the hardware and operating system resources held by the VM. For instance, it is possible to order the resources so that specific DEs have best available communication paths. The DELayout will map the DEs to the PETs of the VM according to the resource details provided by the VM instance.
Furthermore, general DE to PET mapping can be used to offer computational resources with finer granularity than the VM does. The DELayout can be queried for computational and communication capacities of DEs and DE pairs, respectively. This information can be used to best utilize the DE resources when partitioning the computational problem. In combination with other ESMF classes, general DE to PET mapping can be used to realize cache blocking, communication hiding and dynamic load balancing.
Finally, the DELayout layer offers primitives that allow a work queue style dynamic load balancing between DEs.

44.2 Constants

44.2.1 ESMF_PIN

DESCRIPTION:
Specifies which VM resource DEs are pinned to - i.e. PETs or VASs.
The type of this flag is:
`type(ESMF_Pin_Flag)`
The valid values are:

ESMF_PIN_DE_TO_PET Pin DEs against PETs. This means that even if a group of PETs in the VM are sharing a common virtual address space (VAS), DEs cannot be shared between PETs, but must be serviced by the specific PET they are pinned to.

ESMF_PIN_DE_TO_VAS Pin DEs against VASs. DEs may be serviced by any PET that is executing within the virtual address space (VAS) the DE is pinned to.

44.2.2 ESMF_SERVICEREPLY

DESCRIPTION:
Reply when a PET offers to service a DE.
The type of this flag is:
`type(ESMF_ServiceReply_Flag)`
The valid values are:

ESMF_SERVICEREPLY_ACCEPT The service offer has been accepted. The PET is expected to service the DE.

ESMF_SERVICEREPLY_DENY The service offer has been denied. The PET is expected to not service the DE.

44.3 Use and Examples

The following examples demonstrate how to create, use and destroy DELayout objects.

44.3.1 Default DELayout

Without specifying any of the optional parameters the created `ESMF_DELayout` defaults into having as many DEs as there are PETs in the associated VM object. Consequently the resulting DELayout describes a simple 1-to-1 DE to PET mapping.

```
  delayout = ESMF_DELayoutCreate(rc=rc)
```

The default DE to PET mapping is simply:

```
DE 0  -> PET 0
DE 1  -> PET 1
...
```

DELayout objects that are not used any longer should be destroyed.

```
call ESMF_DELayoutDestroy(delayout, rc=rc)
```

The optional `vm` argument can be provided to DELayoutCreate() to lower the method's overhead by the amount it takes to determine the current VM.

```
delayout = ESMF_DELayoutCreate(vm=vm, rc=rc)
```

By default all PETs of the associated VM will be considered. However, if the optional argument `petList` is present DEs will only be mapped against the PETs contained in the list. When the following example is executed on four PETs it creates a DELayout with four DEs by default that are mapped to the provided PETs in their given order. It is erroneous to specify PETs that are not part of the VM context on which the DELayout is defined.

```
delayout = ESMF_DELayoutCreate(petList=(/(i,i=petCount-1,1,-1)/), rc=rc)
```

Once the end of the petList has been reached the DE to PET mapping continues from the beginning of the list. For a 4 PET VM the above created DELayout will end up with the following DE to PET mapping:

```
DE 0  -> PET 3
DE 1  -> PET 2
DE 2  -> PET 1
DE 2  -> PET 3
```

44.3.2 DELayout with specified number of DEs

The `deCount` argument can be used to specify the number of DEs. In this example a DELayout is created that contains four times as many DEs as there are PETs in the VM.

```
delayout = ESMF_DELayoutCreate(deCount=4*petCount, rc=rc)
```

Cyclic DE to PET mapping is the default. For 4 PETs this means:

```
DE 0, 4,  8, 12  -> PET 0
DE 1, 5,  9, 13  -> PET 1
DE 2, 6, 10, 14  -> PET 2
DE 3, 7, 11, 15  -> PET 3
```

The default DE to PET mapping can be overridden by providing the `deGrouping` argument. This argument provides a positive integer group number for each DE in the DELayout. All of the DEs of a group will be mapped against the same PET. The actual group index is arbitrary (but must be positive) and its value is of no consequence.

```
delayout = ESMF_DELayoutCreate(deCount=4*petCount, &
  deGrouping=(/(i/4,i=0,4*petCount-1)/), rc=rc)
```

This will achieve blocked DE to PET mapping. For 4 PETs this means:

```
DE  0,  1,  2,  3  -> PET 0
DE  4,  5,  6,  7  -> PET 1
DE  8,  9, 10, 11  -> PET 2
DE 12, 13, 14, 15  -> PET 3
```

44.3.3 DELayout with computational and communication weights

The quality of the partitioning expressed by the DE to PET mapping depends on the amount and quality of information provided during DELayout creation. In the following example the `compWeights` argument is used to specify relative computational weights for all DEs and communication weights for DE pairs are provided by the `commWeights` argument. The example assumes four DEs.

```
allocate(compWeights(4))
allocate(commWeights(4, 4))
! setup compWeights and commWeights according to computational problem
delayout = ESMF_DELayoutCreate(deCount=4, compWeights=compWeights, &
  commWeights=commWeights, rc=rc)
deallocate(compWeights, commWeights)
```

The resulting DE to PET mapping depends on the specifics of the VM object and the provided compWeights and commWeights arrays.

44.3.4 DELayout from petMap

Full control over the DE to PET mapping is provided via the `petMap` argument. This example maps the DEs to PETs in reverse order. In the 4-PET case this will result in the following mapping:

```
DE 0 -> PET 3
DE 1 -> PET 2
DE 2 -> PET 1
DE 3 -> PET 0
```

```
delayout = ESMF_DELayoutCreate(petMap=(/(i,i=petCount-1,0,-1)/), rc=rc)
```

44.3.5 DELayout from petMap with multiple DEs per PET

The `petMap` argument gives full control over DE to PET mapping. The following example run on 4 or more PETs maps DEs to PETs according to the following table:

```
DE 0 -> PET 3
DE 1 -> PET 3
DE 2 -> PET 1
DE 3 -> PET 0
DE 4 -> PET 2
DE 5 -> PET 1
DE 6 -> PET 3
DE 7 -> PET 1
```

```
delayout = ESMF_DELayoutCreate(petMap=(/3, 3, 1, 0, 2, 1, 3, 1/), rc=rc)
```

44.3.6 Working with a DELayout - simple 1-to-1 DE to PET mapping

The simplest case is a DELayout with as many DEs as PETs where each DE is against a separate PET. This of course implies that the number of DEs equals the number of PETs. This special 1-to-1 DE to PET mapping is very common and many codes assume this mapping. The following example code shows how a DELayout can be queried about its mapping.

```
  delayout = ESMF_DELayoutCreate(rc=rc)


  call ESMF_DELayoutGet(delayout, oneToOneFlag=oneToOneFlag, rc=rc)
  if (rc /= ESMF_SUCCESS) finalrc=rc
  if (.not. oneToOneFlag) then
    ! handle the unexpected case of general DE to PET mapping
  endif
  allocate(localDeList(1))
  call ESMF_DELayoutGet(delayout, localDeList=localDeList, rc=rc)
  if (rc /= ESMF_SUCCESS) finalrc=rc
  myDe = localDeList(1)
  deallocate(localDeList)
```

44.3.7 Working with a DELayout - general DE to PET mapping

In general a DELayout may describe a DE to PET mapping that is not 1-to-1. The following example shows how code can be written in a general form that will work on all PETs for DELayouts with general or 1-to-1 DE to PET mapping.

```
  delayout = ESMF_DELayoutCreate(deCount=petCount+2, rc=rc)


  call ESMF_DELayoutGet(delayout, localDeCount=localDeCount, rc=rc)
  if (rc /= ESMF_SUCCESS) finalrc=rc
  allocate(localDeList(localDeCount))
  call ESMF_DELayoutGet(delayout, localDeList=localDeList, rc=rc)
  if (rc /= ESMF_SUCCESS) finalrc=rc
  do i=1, localDeCount
    workDe = localDeList(i)
!    print *, "I am PET", localPET, " and I am working on DE ", workDe
  enddo
  deallocate(localDeList)
```

44.3.8 Work queue dynamic load balancing

The DELayout API includes two calls that can be used to easily implement work queue dynamic load balancing. The work load is broken up into DEs (more than there are PETs) and processed by the PETs. Load balancing is only possible for ESMF multi-threaded VMs and requires that DEs are pinned to VASs instead of the PETs (default). The following example will run for any VM and DELayout, however, load balancing will only occur under the mentioned conditions.

```
  delayout = ESMF_DELayoutCreate(deCount=petCount+2, &
    pinflag=ESMF_PIN_DE_TO_VAS, rc=rc)


  call ESMF_DELayoutGet(delayout, vasLocalDeCount=localDeCount, rc=rc)
  if (rc /= ESMF_SUCCESS) finalrc=rc
  allocate(localDeList(localDeCount))
  call ESMF_DELayoutGet(delayout, vasLocalDeList=localDeList, rc=rc)
  if (rc /= ESMF_SUCCESS) finalrc=rc
  do i=1, localDeCount
    workDe = localDeList(i)
    print *, "I am PET", localPET, &
             " and I am offering service for DE ", workDe
```

```
    reply = ESMF_DELayoutServiceOffer(delayout, de=workDe, rc=rc)
    if (rc /= ESMF_SUCCESS) finalrc=rc
    if (reply == ESMF_SERVICEREPLY_ACCEPT) then
      ! process work associated with workDe
      print *, "I am PET", localPET, ", service offer for DE ", workDe, &
        " was accepted."
      call ESMF_DELayoutServiceComplete(delayout, de=workDe, rc=rc)
      if (rc /= ESMF_SUCCESS) finalrc=rc
    endif
  enddo
  deallocate(localDeList)
```

44.4 Restrictions and Future Work

44.5 Design and Implementation Notes

The DELayout class is a light weight object. It stores the DE to PET and VAS mapping for all DEs within all PET instances and a list of local DEs for each PET instance. The DELayout does not store the computational and communication weights optionally provided as arguments to the create method. These hints are only used during create while they are available in user owned arrays.

44.6 Class API

44.6.1 ESMF_DELayoutAssignment(=) - DELayout assignment

INTERFACE:

```
     interface assignment(=)
     delayout1 = delayout2
```

ARGUMENTS:

```
     type(ESMF_DELayout) :: delayout1
     type(ESMF_DELayout) :: delayout2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign delayout1 as an alias to the same ESMF DELayout object in memory as delayout2. If delayout2 is invalid, then delayout1 will be equally invalid after the assignment.
The arguments are:

delayout1 The `ESMF_DELayout` object on the left hand side of the assignment.

delayout2 The `ESMF_DELayout` object on the right hand side of the assignment.

44.6.2 ESMF_DELayoutOperator(==) - DELayout equality operator

INTERFACE:

```
   interface operator(==)
     if (delayout1 == delayout2) then ... endif
               OR
     result = (delayout1 == delayout2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_DELayout), intent(in) :: delayout1
type(ESMF_DELayout), intent(in) :: delayout2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether delayout1 and delayout2 are valid aliases to the same ESMF DELayout object in memory. For a more general comparison of two ESMF DELayouts, going beyond the simple alias test, the ESMF_DELayoutMatch() function (not yet implemented) must be used.
The arguments are:

delayout1 The `ESMF_DELayout` object on the left hand side of the equality operation.

delayout2 The `ESMF_DELayout` object on the right hand side of the equality operation.

44.6.3 ESMF_DELayoutOperator(/=) - DELayout not equal operator

INTERFACE:

```
interface operator(/=)
  if (delayout1 /= delayout2) then ... endif
            OR
  result = (delayout1 /= delayout2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_DELayout), intent(in) :: delayout1
type(ESMF_DELayout), intent(in) :: delayout2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether delayout1 and delayout2 are *not* valid aliases to the same ESMF DELayout object in memory. For a more general comparison of two ESMF DELayouts, going beyond the simple alias test, the ESMF_DELayoutMatch() function (not yet implemented) must be used.
The arguments are:

delayout1 The `ESMF_DELayout` object on the left hand side of the non-equality operation.

delayout2 The `ESMF_DELayout` object on the right hand side of the non-equality operation.

44.6.4 ESMF_DELayoutCreate - Create DELayout object

INTERFACE:

```
! Private name; call using ESMF_DELayoutCreate()
function ESMF_DELayoutCreateDefault(deCount, deGrouping, &
  pinflag, petList, vm, rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                     intent(in),  optional :: deCount
    integer, target,             intent(in),  optional :: deGrouping(:)
    type(ESMF_Pin_Flag),         intent(in),  optional :: pinflag
    integer, target,             intent(in),  optional :: petList(:)
    type(ESMF_VM),               intent(in),  optional :: vm
    integer,                     intent(out), optional :: rc
```

RETURN VALUE:

```
    type(ESMF_DELayout) :: ESMF_DELayoutCreateDefault
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an `ESMF_DELayout` object on the basis of optionally provided restrictions. By default a DELayout with deCount equal to petCount will be created, each DE mapped to a single PET. However, the number of DEs as well grouping of DEs and PETs can be specified via the optional arguments.
The arguments are:

[deCount] Number of DEs to be provided by the created DELayout. By default the number of DEs equals the number of PETs in the associated VM context. Specifying a `deCount` smaller than the number of PETs will result in unassociated PETs. This may be used to share VM resources between DELayouts within the same ESMF component. Specifying a `deCount` greater than the number of PETs will result in multiple DE to PET mapping.

[deGrouping] This optional argument must be of size deCount. Its content assigns a DE group index to each DE of the DELayout. A group index of -1 indicates that the associated DE isn't member of any particular group. The significance of DE groups is that all the DEs belonging to a certain group will be mapped against the *same* PET. This does not, however, mean that DEs belonging to different DE groups must be mapped to different PETs.

[pinflag] This flag specifies which type of resource DEs are pinned to. The default is to pin DEs to PETs. Alternatively it is also possible to pin DEs to VASs. See section 44.2.1 for a list of valid pinning options.

[petList] List specifying PETs to be used by this DELayout. This can be used to control the PET overlap between DELayouts within the same ESMF component. It is erroneous to specify PETs that are not within the provided VM context. The default is to include all the PETs of the VM.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

44.6.5 ESMF_DELayoutCreate - Create DELayout from petMap

INTERFACE:

```
   ! Private name; call using ESMF_DELayoutCreate()
   function ESMF_DELayoutCreateFromPetMap(petMap, pinflag, &
     vm, rc)
```

ARGUMENTS:

```
     integer,                      intent(in)            :: petMap(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Pin_Flag),          intent(in),  optional :: pinflag
     type(ESMF_VM),                intent(in),  optional :: vm
     integer,                      intent(out), optional :: rc
```

RETURN VALUE:

```
     type(ESMF_DELayout) :: ESMF_DELayoutCreateFromPetMap
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Create an ESMF_DELayout with exactly specified DE to PET mapping.
This ESMF method must be called in unison by all PETs of the VM. Calling this method from a PET not part of the VM or not calling it from a PET that is part of the VM will result in undefined behavior. ESMF does not guard against violation of the unison requirement. The call is not collective, there is no communication between PETs.
The arguments are:

petMap List specifying the DE-to-PET mapping. The list elements correspond to DE 0, 1, 2, ... and map against the specified PET of the VM context. The size of the petMap argument determines the number of DEs in the created DELayout. It is erroneous to specify a PET identifier that lies outside the VM context.

[pinflag] This flag specifies which type of resource DEs are pinned to. The default is to pin DEs to PETs. Alternatively it is also possible to pin DEs to VASs. See section 44.2.1 for a list of valid pinning options.

[vm] Optional ESMF_VM object. The VM of the current context is the typical and default value.

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

44.6.6 ESMF_DELayoutDestroy - Release resources associated with DELayout object

INTERFACE:

```
   subroutine ESMF_DELayoutDestroy(delayout, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Destroy an ESMF_DELayout object.
The arguments are:

delayout ESMF_DELayout object to be destroyed.

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

44.6.7 ESMF_DELayoutGet - Get DELayout internals

INTERFACE:

```
   subroutine ESMF_DELayoutGet(delayout, vm, deCount, petMap, &
     vasMap, oneToOneFlag, pinflag, localDeCount, localDeList, &
     vasLocalDeCount, vasLocalDeList, rc)
```

ARGUMENTS:

```
     type(ESMF_DELayout),      intent(in)            :: delayout
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_VM),            intent(out), optional :: vm
     integer,                  intent(out), optional :: deCount
     integer, target,          intent(out), optional :: petMap(:)
     integer, target,          intent(out), optional :: vasMap(:)
     logical,                  intent(out), optional :: oneToOneFlag
     type(ESMF_Pin_Flag),      intent(out), optional :: pinflag
     integer,                  intent(out), optional :: localDeCount
     integer, target,          intent(out), optional :: localDeList(:)
     integer,                  intent(out), optional :: vasLocalDeCount
     integer, target,          intent(out), optional :: vasLocalDeList(:)
     integer,                  intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Access to DELayout information.
The arguments are:

delayout Queried ESMF_DELayout object.

[vm] Upon return this holds the ESMF_VM object on which the delayout is defined.

[deCount] Upon return this holds the total number of DEs.

[petMap] Upon return this holds the list of PETs against which the DEs are mapped. The petMap argument must at least be of size deCount.

[vasMap] Upon return this holds the list of VASs against which the DEs are mapped. The vasMap argument must at least be of size deCount.

[oneToOneFlag] Upon return this holds .TRUE. if the specified ESMF_DELayout describes a 1-to-1 mapping between DEs and PETs, .FALSE. otherwise.

[pinflag] Upon return this flag will indicate the type of DE pinning. See section 44.2.1 for a list of valid pinning options.

[localDeCount] Upon return this holds the number of DEs associated with the local PET.

[localDeList] Upon return this holds the list of DEs associated with the local PET. The provided argument must at least be of size localDeCount.

[vasLocalDeCount] Upon return this holds the number of DEs associated with the local VAS.

[vasLocalDeList] Upon return this holds the list of DEs associated with the local VAS. The provided argument must at least be of size `vasLocalDeCount`.

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

44.6.8 ESMF_DELayoutPrint - Print DELayout internals

INTERFACE:

```
   subroutine ESMF_DELayoutPrint(delayout, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Prints internal information about the specified `ESMF_DELayout` object to `stdout`.

The arguments are:

delayout Specified `ESMF_DELayout` object.

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

44.6.9 ESMF_DELayoutServiceComplete - Close service window

INTERFACE:

```
   recursive subroutine ESMF_DELayoutServiceComplete(delayout, de, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

The PET who's service offer was accepted for `de` must use `ESMF_DELayoutServiceComplete` to close the service window.
The arguments are:

delayout Specified `ESMF_DELayout` object.

de DE for which to close service window.

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

44.6.10 ESMF_DELayoutServiceOffer - Offer service for a DE in DELayout

INTERFACE:

```
   recursive function ESMF_DELayoutServiceOffer(delayout, de, rc)
```

ARGUMENTS:

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

RETURN VALUE:

```
     type(ESMF_ServiceReply_Flag) :: ESMF_DELayoutServiceOffer
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Offer service for a DE in the ESMF_DELayout object. This call together with ESMF_DELayoutServiceComplete() provides the synchronization primitives between the PETs of an ESMF multi-threaded VM necessary for dynamic load balancing via a work queue approach.
The calling PET will either receive ESMF_SERVICEREPLY_ACCEPT if the service offer has been accepted by DELayout or ESMF_SERVICEREPLY_DENY if the service offer was denied. The service offer paradigm is different from a simple mutex approach in that the DELayout keeps track of the number of service offers issued for each DE by each PET and accepts only one PET's offer for each offer increment. This requires that all PETs use ESMF_DELayoutServiceOffer() in unison. See section 44.2.2 for the potential return values.
The arguments are:

delayout Specified ESMF_DELayout object.

de DE for which service is offered by the calling PET.

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

44.6.11 ESMF_DELayoutValidate - Validate DELayout internals

INTERFACE:

```
   subroutine ESMF_DELayoutValidate(delayout, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the delayout is internally consistent. The method returns an error code if problems are found.
The arguments are:

delayout Specified ESMF_DELayout object.

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

45 VM Class

45.1 Description

The ESMF VM (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component.

In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods. The VM communication calls are very similar to MPI. Data references in VM communication calls must be provided as raw, language specific, one-dimensional, contiguous data arrays. The similarity between VM and MPI communication calls is striking and there are many equivalent point-to-point and collective communication calls. However, unlike MPI, the VM communication calls support communication between threaded PETs in a completely transparent fashion.

Many ESMF applications do not interact with the VM class directly very much. The resource management aspect is wrapped completely transparent into the ESMF Component concept. Often the only reason that user code queries a Component object for the associated VM object is to inquire about resource information, such as the localPet or the petCount. Further, for most applications the use of higher level communication APIs, such as provided by Array and Field, are much more convenient than using the low level VM communication calls.

The basic elements of a VM are called PETs, which stands for Persistent Execution Threads. These are equivalent to OS threads with a lifetime of at least that of the associated component. All VM functionality is expressed in terms of PETs. In the simplest, and most common case, a PET is equivalent to an MPI process. However, ESMF also supports multi-threading, where multiple PETs run as Pthreads inside the same virtual address space (VAS).

The resource management functions of the VM class become visible when a component, or the driver code, creates sub-components. Section 14.4.5 discusses this aspect from the Superstructure perspective and provides links to the relevant Component examples in the documentation.

There are two parts to resource management, the parent and the child. When the parent component creates a child component, the parent VM object provides the resources on which the child is created with ESMF_GridCompCreate() or ESMF_CplCompCreate(). The optional petList argument to these calls limits the resources that the parent gives to a specific child. The child component, may specify - during its optional ESMF_<Grid/Cpl>CompSetVM() method - how it wants to arrange the inherited resources in its own VM. After this, all standard ESMF methods of the Component, including ESMF_<Grid/Cpl>CompSetServices(), will execute in the child VM. Notice that the ESMF_<Grid/Cpl>CompSetVM() routine, although part of the child Component, must execute *before* the child VM has been started up. It runs in the parent VM context. The child VM is created and started up just before the user-written set services routine, specified as an argument to ESMF_<Grid/Cpl>CompSetServices(), is entered.

45.2 Use and Examples

The concept of the ESMF Virtual Machine (VM) is so fundamental to the framework that every ESMF application uses it. However, for many user applications the VM class is transparently hidden behind the ESMF Component concept and higher data classes (e.g. Array, Field). The interaction between user code and VM is often only indirect. The following examples provide an overview of where the VM class can come into play in user code.

45.2.1 Global VM

This complete example program demonstrates the simplest ESMF application, consisting of only a main program without any Components. The global VM, which is automatically created during the ESMF_Initialize() call, is obtained using two different methods. First the global VM will be returned by ESMF_Initialize() if the optional vm argument is specified. The example uses the VM object obtained this way to call the VM print method. Second, the global VM can be obtained anywhere in the user application using the ESMF_VMGetGlobal() call. The identical VM is returned and several VM query methods are called to inquire about the associated resources.

```
program ESMF_VMDefaultBasicsEx

  use ESMF
```

```
  implicit none

  ! local variables
  integer:: rc
  type(ESMF_VM):: vm
  integer:: localPet, petCount, peCount, ssiId, vas


  call ESMF_Initialize(vm=vm, defaultlogfilename="VMDefaultBasicsEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  ! Providing the optional vm argument to ESMF_Initialize() is one way of
  ! obtaining the global VM.


  call ESMF_VMPrint(vm, rc=rc)


  call ESMF_VMGetGlobal(vm=vm, rc=rc)
  ! Calling ESMF_VMGetGlobal() anywhere in the user application is the other
  ! way to obtain the global VM object.


  call ESMF_VMGet(vm, localPet=localPet, petCount=petCount, peCount=peCount, &
    rc=rc)
  ! The VM object contains information about the associated resources. If the
  ! user code requires this information it must query the VM object.


  print *, "This PET is localPet: ", localPet
  print *, "of a total of ",petCount," PETs in this VM."
  print *, "There are ", peCount," PEs referenced by this VM"

  call ESMF_VMGet(vm, localPet, peCount=peCount, ssiId=ssiId, vas=vas, rc=rc)


  print *, "This PET is executing in virtual address space (VAS) ", vas
  print *, "located on single system image (SSI) ", ssiId
  print *, "and is associated with ", peCount, " PEs."

  call ESMF_Finalize(rc=rc)

end program
```

45.2.2 Getting the MPI Communicator from an VM object

Sometimes user code requires access to the MPI communicator, e.g. to support legacy code that contains explict MPI communication calls. The correct way of wrapping such code into ESMF is to obtain the MPI intra-communicator out of the VM object. In order not to interfere with ESMF communications it is advisable to duplicate the communicator before using it in user-level MPI calls. In this example the duplicated communicator is used for a user controlled `MPI_Barrier()`.

```
  integer:: mpic
```

```
integer:: mpic2
```

```
call ESMF_VMGet(vm, mpiCommunicator=mpic, rc=rc)
! The returned MPI communicator spans the same MPI processes that the VM
! is defined on.
```

```
call MPI_Comm_dup(mpic, mpic2, ierr)
! Duplicate the MPI communicator not to interfere with ESMF communications.
! The duplicate MPI communicator can be used in any MPI call in the user
! code. Here the MPI_Barrier() routine is called.
call MPI_Barrier(mpic2, ierr)
```

45.2.3 Nesting ESMF inside a user MPI application

It is possible to nest an ESMF application inside a user application that explicitly calls `MPI_Init()` and `MPI_Finalize()`. The `ESMF_Initialize()` call automatically checks whether MPI has already been initialized, and if so does not call `MPI_Init()` internally. On the finalize side, `ESMF_Finalize()` can be instructed to *not* call `MPI_Finalize()`, making it the responsibility of the outer code to finalize MPI.

```
call MPI_Init(ierr)
! User code initializes MPI.
```

```
call ESMF_Initialize(defaultlogfilename="VMUserMpiEx.Log", &
                  logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
! ESMF_Initialize() does not call MPI_Init() if it finds MPI initialized.
```

```
call ESMF_Finalize(endflag=ESMF_END_KEEPMPI, rc=rc)
! Calling with endflag=ESMF_END_KEEPMPI instructs ESMF_Finalize() to keep
! MPI active.
```

```
call MPI_Finalize(ierr)
! It is the responsibility of the outer user code to finalize MPI.
```

45.2.4 Nesting ESMF inside a user MPI application on a subset of MPI ranks

The previous example demonstrated that it is possible to nest an ESMF application, i.e. `ESMF_Initialize()`...`ESMF_Finalize()` inside `MPI_Init()`...`MPI_Finalize()`. It is not necessary that all MPI ranks enter the ESMF application. The following example shows how the user code can pass an MPI communicator to `ESMF_Initialize()`, and enter the ESMF application on a subset of MPI ranks.

```
call MPI_Init(ierr)
! User code initializes MPI.
```

```
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
! User code determines the local rank.
```

```
! User code prepares MPI communicator "esmfComm" that only contains
! rank 0 and 1.


if (rank < 2) then
  call ESMF_Initialize(mpiCommunicator=esmfComm, &
  defaultlogfilename="VMUserMpiCommEx.Log", &
                   logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  ! Only call ESMF_Initialize() on rank 0 and 1, passing the prepared MPI
  ! communicator that spans these ranks.


  call ESMF_Finalize(endflag=ESMF_END_KEEPMPI, rc=rc)
  ! Finalize ESMF without finalizing MPI. The user application will call
  ! MPI_Finalize() on all ranks.


endif


call MPI_Finalize(ierr)
! User code finalizes MPI.
```

45.2.5 Send/Recv

The VM layer provides MPI-like point-to-point communication. Use `ESMF_VMSend()` and `ESMF_VMRecv()` to pass data between two PETs. The following code sends data from PET 'src' and receives it on PET 'dst'. Both PETs must be part of the same VM. The sendData and recvData arguments must be 1-dimensional arrays.

```
if (localPet==src) &
  call ESMF_VMSend(vm, sendData=localData, count=count, dstPet=dst, rc=rc)


if (localPet==dst) &
  call ESMF_VMRecv(vm, recvData=localData, count=count, srcPet=src, rc=rc)
```

45.2.6 Scatter and Gather

The VM layer provides MPI-like collective communication. `ESMF_VMScatter()` scatters data located on `root` PET across all the PETs of the VM. `ESMF_VMGather()` provides the opposite operation, gathering data from all the PETs of the VM onto `root` PET.

```
call ESMF_VMScatter(vm, sendData=array1, recvData=array2, count=nsize, &
  rootPet=scatterRoot, rc=rc)
! Both sendData and recvData must be 1-d arrays.


call ESMF_VMGather(vm, sendData=array2, recvData=array1, count=nsize, &
  rootPet=gatherRoot, rc=rc)
! Both sendData and recvData must be 1-d arrays.
```

45.2.7 AllReduce and AllFullReduce

Use `ESMF_VMAllReduce()` to reduce data distributed across the PETs of a VM into a result vector, returned on all the PETs. Further, use `ESMF_VMAllFullReduce()` to reduce the data into a single scalar returned on all PETs.

```
  call ESMF_VMAllReduce(vm, sendData=array1, recvData=array2, count=nsize, &
    reduceflag=ESMF_REDUCE_SUM, rc=rc)
  ! Both sendData and recvData must be 1-d arrays. Reduce distributed
  ! sendData element by element into recvData and return in on all PETs.


  call ESMF_VMAllFullReduce(vm, sendData=array1, recvData=result, &
    count=nsize, reduceflag=ESMF_REDUCE_SUM, rc=rc)
  ! sendData must be 1-d array. Fully reduce the distributed sendData
  ! into a single scalar and return it in recvData on all PETs.
```

45.2.8 VM and Components

The following example shows the role that the VM plays in connection with ESMF Components. A single Component is created in the main program. Through the optional `petList` argument the driver code specifies that only resources associated with PET 0 are given to the `gcomp` object.
When the Component code is invoked through the standard ESMF Component methods Initialize, Run, or Finalize the Component's VM is automatically entered. Inside of the user-written Component code the Component VM can be obtained by querying the Component object. The VM object will indicate that only a single PET is executing the Component code.

```
module ESMF_VMComponentEx_gcomp_mod


  recursive 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(ESMF_VM):: vm

    ! get this Component's vm
    call ESMF_GridCompGet(gcomp, vm=vm)

    ! the VM object contains information about the execution environment of
    ! the Component

    call ESMF_VMPrint(vm, rc=rc)

    rc = 0
  end subroutine !--------------------------------------------------------------


  recursive 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(ESMF_VM):: vm

    ! get this Component's vm
    call ESMF_GridCompGet(gcomp, vm=vm)

    ! the VM object contains information about the execution environment of
    ! the Component

    call ESMF_VMPrint(vm, rc=rc)

    rc = 0
  end subroutine !--------------------------------------------------------------

  recursive 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(ESMF_VM):: vm

    ! get this Component's vm
    call ESMF_GridCompGet(gcomp, vm=vm)

    ! the VM object contains information about the execution environment of
    ! the Component

    call ESMF_VMPrint(vm, rc=rc)

    rc = 0
  end subroutine !--------------------------------------------------------------

end module


program ESMF_VMComponentEx
  use ESMF
  use ESMF_VMComponentEx_gcomp_mod
  implicit none

  ! local variables


  gcomp = ESMF_GridCompCreate(petList=(/0/), rc=rc)


  call ESMF_GridCompSetServices(gcomp, mygcomp_register, rc=rc)


  call ESMF_GridCompInitialize(gcomp, rc=rc)


  call ESMF_GridCompRun(gcomp, rc=rc)
```

```
  call ESMF_GridCompFinalize(gcomp, rc=rc)

  call ESMF_GridCompDestroy(gcomp, rc=rc)

  call ESMF_Finalize(rc=rc)

end program
```

45.3 Restrictions and Future Work

1. **Fortran array section syntax is not supported** for source and destination arguments in VM communication calls. Just as for MPI, the VM communication calls require contiguous data arrays. Fortran array sections are not guaranteed to be contiguous. This fundamental restriction is not likely to change in future releases.

2. **Non-blocking `Reduce()` operations *not* implemented.** None of the reduce communication calls have an implementation for the non-blocking feature. This affects:
   - `ESMF_VMAllFullReduce()`,
   - `ESMF_VMAllReduce()`,
   - `ESMF_VMReduce()`.

3. **Limitations when using `mpiuni` mode.** In `mpiuni` mode non-blocking communications are limited to one outstanding message per source-destination PET pair. Furthermore, in `mpiuni` mode the message length must be smaller than the internal ESMF buffer size.

4. **Alternative communication paths not accessible.** All user accessible VM communication calls are currently implemented using MPI-1.2. VM's implementation of alternative communication techniques, such as shared memory between threaded PETs and POSIX IPC between PETs located on the same single system image, are currently inaccessible to the user. (One exception to this is the `mpiuni` case for which the VM automatically utilizes a shared memory path.)

5. **Data arrays in VM comm calls are *assumed shape* with rank=1.** Currently all dummy arrays in VM comm calls are defined as *assumed shape* arrays of rank=1. The motivation for this choice is that the use of assumed shape dummy arrays guards against the Fortran copy in/out problem. However it may not be as flexible as desired from the user perspective. Alternatively all dummy arrays could be defined as *assumed size* arrays, as it is done in most MPI implementations, allowing arrays of various rank to be passed into the comm methods.

45.4 Design and Implementation Notes

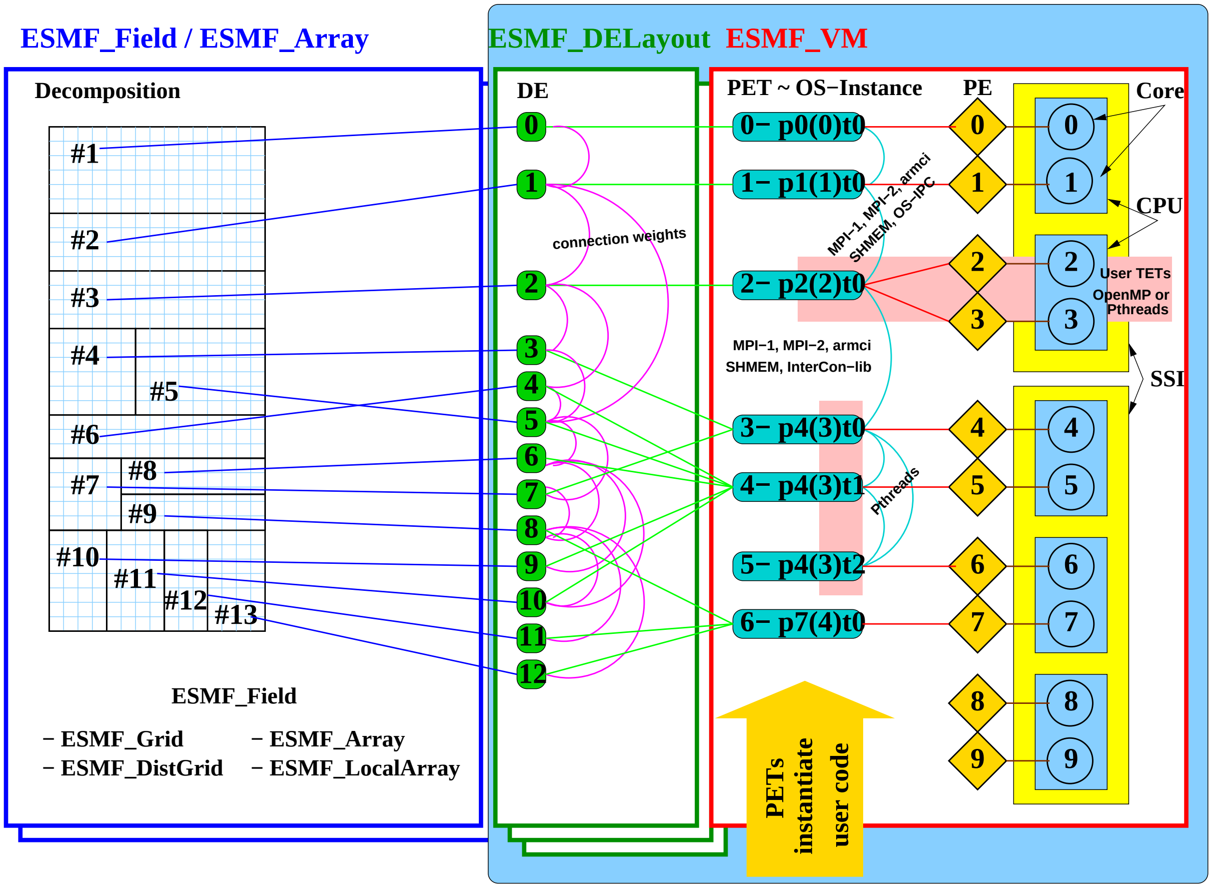
The VM class provides an additional layer of abstraction on top of the POSIX machine model, making it suitable for HPC applications. There are four key aspects the VM class deals with.

1. Encapsulation of hardware and operating system details within the concept of Persistent Execution Threads (PETs).

2. Resource management in terms of PETs with a guard against over-subscription.

3. Topological description of the underlying configuration of the compute resources in terms of PETs.

4. Transparent communication API for point-to-point and collective PET-based primitives, hiding the many different communication channels and offering best possible performance.



Definition of terms used in the diagram

- PE: A processing element (PE) is an alias for the smallest physical processing unit available on a particular hardware platform. In the language of today's microprocessor architecture technology a PE is identical to a core, however, if future microprocessor designs change the smallest physical processing unit the mapping of the PE to actual hardware will change accordingly. Thus the PE layer separates the hardware specific part of the VM from the hardware-independent part. Each PE is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.

- Core: A Core is the smallest physical processing unit which typically comprises a register set, an integer arithmetic unit, a floating-point unit and various control units. Each Core is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.

- CPU: The central processing unit (CPU) houses single or multiple cores, providing them with the interface to system memory, interconnects and IO. Typically the CPU provides some level of caching for the instruction and data streams in and out of the Cores. Cores in a multi-core CPU typically share some caches. Each CPU is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.

- SSI: A single system image (SSI) spans all the CPUs controlled by a single running instance of the operating system. SMP and NUMA are typical multi-CPU SSI architectures. Each SSI is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.

- TOE: A thread of execution (TOE) executes an instruction sequence. TOE's come in two flavors: PET and TET.

- PET: A persistent execution thread (PET) executes an instruction sequence on an associated set of data. The PET has a lifetime at least as long as the associated data set. In ESMF the PET is the central concept of abstraction provided by the VM class. The PETs of an VM object are labeled from 0 to N-1 where N is the total number of PETs in the VM object.

- TET: A transient execution thread (TET) executes an instruction sequence on an associated set of data. A TET's lifetime might be shorter than that of the associated data set.
- OS-Instance: The OS-Instance of a TOE describes how a particular TOE is instantiated on the OS level. Using POSIX terminology a TOE will run as a single thread within a single- or multi-threaded process.
- Pthreads: Communication via the POSIX Thread interface.
- MPI-1, MPI-2: Communication via MPI standards 1 and 2.
- armci: Communication via the aggregate remote memory copy interface.
- SHMEM: Communication via the SHMEM interface.
- OS-IPC: Communication via the operating system's inter process communication interface. Either POSIX IPC or System V IPC.
- InterCon-lib: Communication via the interconnect's library native interface. An example is the Elan library for Quadrics.

The POSIX machine abstraction, while a very powerful concept, needs augmentation when applied to HPC applications. Key elements of the POSIX abstraction are processes, which provide virtually unlimited resources (memory, I/O, sockets, ...) to possibly multiple threads of execution. Similarly POSIX threads create the illusion that there is virtually unlimited processing power available to each POSIX process. While the POSIX abstraction is very suitable for many multi-user/multi-tasking applications that need to share limited physical resources, it does not directly fit the HPC workload where over-subscription of resources is one of the most expensive modes of operation.
ESMF's virtual machine abstraction is based on the POSIX machine model but holds additional information about the available physical processing units in terms of Processing Elements (PEs). A PE is the smallest physical processing unit and encapsulates the hardware details (Cores, CPUs and SSIs).
There is exactly one physical machine layout for each application, and all VM instances have access to this information. The PE is the smallest processing unit which, in today's microprocessor technology, corresponds to a single Core. Cores are arranged in CPUs which in turn are arranged in SSIs. The setup of the physical machine layout is part of the ESMF initialization process.
On top of the PE concept the key abstraction provided by the VM is the PET. All user code is executed by PETs while OS and hardware details are hidden. The VM class contains a number of methods which allow the user to prescribe how the PETs of a desired virtual machine should be instantiated on the OS level and how they should map onto the hardware. This prescription is kept in a private virtual machine plan object which is created at the same time the associated component is being created. Each time component code is entered through one of the component's registered top–level methods (Initialize/Run/Finalize), the virtual machine plan along with a pointer to the respective user function is used to instantiate the user code on the PETs of the associated VM in form of single- or multi-threaded POSIX processes.
The process of starting, entering, exiting and shutting down a VM is very transparent, all spawning and joining of threads is handled by VM methods "behind the scenes". Furthermore, fundamental synchronization and communication primitives are provided on the PET level through a uniform API, hiding details related to the actual instantiation of the participating PETs.
Within a VM object each PE of the physical machine maps to 0 or 1 PETs. Allowing unassigned PEs provides a means to prevent over-subscription between multiple concurrently running virtual machines. Similarly a maximum of one PET per PE prevents over-subscription within a single VM instance. However, over-subscription is possible by subscribing PETs from different virtual machines to the same PE. This type of over-subscription can be desirable for PETs associated with IO work loads expected to be used infrequently and to block often on IO requests.
On the OS level each PET of a VM object is represented by a POSIX thread (Pthread) either belonging to a single– or multi–threaded process and maps to at least 1 PE of the physical machine, ensuring its execution. Mapping a single PET to multiple PEs provides resources for user–level multi–threading, in which case the user code inquires how many PEs are associated with its PET and if there are multiple PEs available the user code can spawn an equal number of threads (e.g. OpenMP) without risking over-subscription. Typically these user spawned threads are short-lived and

used for fine-grained parallelization in form of TETs. All PEs mapped against a single PET must be part of a unique SSI in order to allow user–level multi–threading!
In addition to discovering the physical machine the ESMF initialization process sets up the default global virtual machine. This VM object, which is the ultimate parent of all VMs created during the course of execution, contains as many PETs as there are PEs in the physical machine. All of its PETs are instantiated in form of single-threaded MPI processes and a 1:1 mapping of PETs to PEs is used for the default global VM.
The VM design and implementation is based on the POSIX process and thread model as well as the MPI-1.2 standard. As a consequence of the latter standard the number of processes is static during the course of execution and is determined at start-up. The VM implementation further requires that the user starts up the ESMF application with as many MPI processes as there are PEs in the available physical machine using the platform dependent mechanism to ensure proper process placement.
All MPI processes participating in a VM are grouped together by means of an MPI_Group object and their context is defined via an MPI_Comm object (MPI intra-communicator). The PET local process id within each virtual machine is equal to the MPI_Comm_rank in the local MPI_Comm context whereas the PET process id is equal to the MPI_Comm_rank in MPI_COMM_WORLD. The PET process id is used within the VM methods to determine the virtual memory space a PET is operating in.
In order to provide a migration path for legacy MPI-applications the VM offers accessor functions to its MPI_Comm object. Once obtained this object may be used in explicit user-code MPI calls within the same context.

45.5 Class API

45.5.1 ESMF_VMAssignment(=) - VM assignment

INTERFACE:

```
interface assignment(=)
vm1 = vm2
```

ARGUMENTS:

```
type(ESMF_VM) :: vm1
type(ESMF_VM) :: vm2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Assign vm1 as an alias to the same ESMF VM object in memory as vm2. If vm2 is invalid, then vm1 will be equally invalid after the assignment.
The arguments are:

vm1 The `ESMF_VM` object on the left hand side of the assignment.

vm2 The `ESMF_VM` object on the right hand side of the assignment.

45.5.2 ESMF_VMOperator(==) - VM equality operator

INTERFACE:

```
interface operator(==)
  if (vm1 == vm2) then ... endif
            OR
  result = (vm1 == vm2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_VM), intent(in) :: vm1
type(ESMF_VM), intent(in) :: vm2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether vm1 and vm2 are valid aliases to the same ESMF VM object in memory. For a more general comparison of two ESMF VMs, going beyond the simple alias test, the ESMF_VMMatch() function (not yet implemented) must be used.
The arguments are:

vm1 The `ESMF_VM` object on the left hand side of the equality operation.

vm2 The `ESMF_VM` object on the right hand side of the equality operation.

45.5.3 ESMF_VMOperator(/=) - VM not equal operator

INTERFACE:

```
interface operator(/=)
  if (vm1 /= vm2) then ... endif
            OR
  result = (vm1 /= vm2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_VM), intent(in) :: vm1
type(ESMF_VM), intent(in) :: vm2
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Test whether vm1 and vm2 are *not* valid aliases to the same ESMF VM object in memory. For a more general comparison of two ESMF VMs, going beyond the simple alias test, the ESMF_VMMatch() function (not yet implemented) must be used.
The arguments are:

vm1 The `ESMF_VM` object on the left hand side of the non-equality operation.

vm2 The `ESMF_VM` object on the right hand side of the non-equality operation.

45.5.4 ESMF_VMAllFullReduce - Fully reduce data across VM, result on all PETs

INTERFACE:

```
    subroutine ESMF_VMAllFullReduce(vm, sendData, recvData, &
      count, reduceflag, syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                      intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      <type>(ESMF_KIND_<kind>),         intent(out)           :: recvData
      integer,                            intent(in)            :: count
      type(ESMF_Reduce_Flag),            intent(in)            :: reduceflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),              intent(in),  optional :: syncflag
      type(ESMF_CommHandle),             intent(out), optional :: commhandle
      integer,                            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective `ESMF_VM` communication call that reduces a contiguous data array of <type><kind> across the `ESMF_VM` object into a single value of the same <type><kind>. The result is returned on all PETs. Different reduction operations can be specified.

This method is overloaded for: `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

TODO: The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with `syncflag = ESMF_SYNC_NONBLOCKING` error code `ESMF_RC_NOT_IMPL` will be returned and an error will be logged.

The arguments are:

vm `ESMF_VM` object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Single data variable to be received. All PETs must specify a valid result variable.

count Number of elements in sendData. Must be the same on all PETs.

reduceflag Reduction operation. See section 9.35 for a list of valid reduce operations.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is `ESMF_SYNC_BLOCKING`. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument `syncflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommWaitAll()` may be used to block on all currently queued communication calls of the VM context.

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

45.5.5 ESMF_VMAllGather - Gather data across VM, result on all PETs

INTERFACE:

```
    subroutine ESMF_VMAllGather(vm, sendData, recvData, count, &
      syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                      intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                            intent(in)            :: count
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),               intent(in),  optional :: syncflag
      type(ESMF_CommHandle),              intent(out), optional :: commhandle
      integer,                            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that gathers contiguous data from all PETs of an ESMF_VM object into an array on all PETs.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Contiguous data array for data to be received. All PETs must specify a valid recvData argument.

count Number of elements to be gathered from each PET. Must be the same on all PETs.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.6 ESMF_VMAllGatherV - GatherV data across VM, result on all PETs

INTERFACE:

```
    subroutine ESMF_VMAllGatherV(vm, sendData, sendCount, &
      recvData, recvCounts, recvOffsets, syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                      intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      integer,                            intent(in)            :: sendCount
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                            intent(in)            :: recvCounts(:)
      integer,                            intent(in)            :: recvOffsets(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),               intent(in),  optional :: syncflag
      type(ESMF_CommHandle),              intent(out), optional :: commhandle
      integer,                            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that gathers contiguous data from all PETs of an ESMF_VM object into an array on all PETs.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with syncflag = ESMF_SYNC_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

sendCount Number of sendData elements to send from local PET to all other PETs.

recvData Single data variable to be received. All PETs must specify a valid result variable.

recvCounts Number of recvData elements to be received from corresponding source PET.

recvOffsets Offsets in units of elements in recvData marking the start of element sequence to be received from source PET.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.7 ESMF_VMAllReduce - Reduce data across VM, result on all PETs

INTERFACE:

```
    subroutine ESMF_VMAllReduce(vm, sendData, recvData, count, &
      reduceflag, syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                    intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                          intent(in)            :: count
      type(ESMF_Reduce_Flag),           intent(in)            :: reduceflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),             intent(in),  optional :: syncflag
      type(ESMF_CommHandle),            intent(out), optional :: commhandle
      integer,                          intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective `ESMF_VM` communication call that reduces a contiguous data array across the `ESMF_VM` object into a contiguous data array of the same <type><kind>. The result array is returned on all PETs. Different reduction operations can be specified.

This method is overloaded for: `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

TODO: The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with `syncflag = ESMF_SYNC_NONBLOCKING` error code `ESMF_RC_NOT_IMPL` will be returned and an error will be logged.

The arguments are:

vm `ESMF_VM` object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Single data variable to be received. All PETs must specify a valid result variable.

count Number of elements in sendData and recvData. Must be the same on all PETs.

reduceflag Reduction operation. See section 9.35 for a list of valid reduce operations.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is `ESMF_SYNC_BLOCKING`. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument `syncflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommWaitAll()` may be used to block on all currently queued communication calls of the VM context.

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

45.5.8 ESMF_VMAllToAllV - AllToAllV communications across VM

INTERFACE:

```
    subroutine ESMF_VMAllToAllV(vm, sendData, sendCounts, &
      sendOffsets, recvData, recvCounts, recvOffsets, syncflag, &
      commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                    intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      integer,                          intent(in)            :: sendCounts(:)
      integer,                          intent(in)            :: sendOffsets(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                          intent(in)            :: recvCounts(:)
      integer,                          intent(in)            :: recvOffsets(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),             intent(in),  optional :: syncflag
      type(ESMF_CommHandle),            intent(out), optional :: commhandle
      integer,                          intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that performs a total exchange operation, sending pieces of the contiguous data buffer semdData to all other PETs while receiving data into the contiguous data buffer recvData from all other PETs.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with syncflag = ESMF_SYNC_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

sendCounts Number of sendData elements to send from local PET to destination PET.

sendOffsets Offsets in units of elements in sendData marking to start of element sequence to be send from local PET to destination PET.

recvData Single data variable to be received. All PETs must specify a valid result variable.

recvCounts Number of recvData elements to be received by local PET from source PET.

recvOffsets Offsets in units of elements in recvData marking to start of element sequence to be received by local PET from source PET.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.9 ESMF_VMBarrier - VM wide barrier

INTERFACE:

```
   subroutine ESMF_VMBarrier(vm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective `ESMF_VM` communication call that blocks calling PET until all PETs of the VM context have issued the call.

The arguments are:

vm `ESMF_VM` object.

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

45.5.10 ESMF_VMBroadcast - Broadcast data across VM

INTERFACE:

```
    subroutine ESMF_VMBroadcast(vm, bcstData, count, rootPet, &
      syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                     intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(inout)         :: bcstData(:)
      integer,                           intent(in)            :: count
      integer,                           intent(in)            :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),              intent(in),  optional :: syncflag
      type(ESMF_CommHandle),             intent(out), optional :: commhandle
      integer,                           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective `ESMF_VM` communication call that broadcasts a contiguous data array from `rootPet` to all other PETs of the `ESMF_VM` object.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`, `ESMF_TYPEKIND_LOGICAL`, `ESMF_TYPEKIND_CHARACTER`.

The arguments are:

vm `ESMF_VM` object.

bcstData Contiguous data array. On `rootPet` `bcstData` holds data that is to be broadcasted to all other PETs. On all other PETs `bcstData` is used to receive the broadcasted data.

count Number of elements in sendData and recvData. Must be the same on all PETs.

rootPet PET that holds data that is being broadcast.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is `ESMF_SYNC_BLOCKING`. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument `syncflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommWaitAll()` may be used to block on all currently queued communication calls of the VM context.

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

45.5.11 ESMF_VMCommWait - Wait for non-blocking VM communication to complete

INTERFACE:

```
   subroutine ESMF_VMCommWait(vm, commhandle, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Wait for non-blocking VM communication specified by the `commhandle` to complete.

The arguments are:

vm `ESMF_VM` object.

commhandle Handle specifying a previously issued non-blocking communication request.

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

45.5.12 ESMF_VMCommWaitAll - Wait for all non-blocking VM comms to complete

INTERFACE:

```
   subroutine ESMF_VMCommWaitAll(vm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Wait for *all* pending non-blocking VM communication within the specified VM context to complete.

The arguments are:

vm ESMF_VM object.

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

45.5.13 ESMF_VMGather - Gather data from across VM

INTERFACE:

```
    subroutine ESMF_VMGather(vm, sendData, recvData, count, rootPet, &
      syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                     intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                           intent(in)            :: count
      integer,                           intent(in)            :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),              intent(in),  optional :: syncflag
      type(ESMF_CommHandle),             intent(out), optional :: commhandle
      integer,                           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that gathers contiguous data from all PETs of an ESMF_VM object (including rootPet) into an array on rootPet.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Contiguous data array for data to be received. Only the recvData array specified by the rootPet will be used by this method.

count Number of elements to be send from each PET to rootPet. Must be the same on all PETs.

rootPet PET on which data is gathereds.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument `syncflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommWaitAll()` may be used to block on all currently queued communication calls of the VM context.

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

45.5.14 ESMF_VMGatherV - GatherV data from across VM

INTERFACE:

```
    subroutine ESMF_VMGatherV(vm, sendData, sendCount, recvData, &
      recvCounts, recvOffsets, rootPet, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                     intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      integer,                           intent(in)            :: sendCount
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                           intent(in)            :: recvCounts(:)
      integer,                           intent(in)            :: recvOffsets(:)
      integer,                           intent(in)            :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective `ESMF_VM` communication call that gathers contiguous data from all PETs of an `ESMF_VM` object into an array on rootPet.

This method is overloaded for: `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

TODO: The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with `syncflag = ESMF_SYNC_NONBLOCKING` error code `ESMF_RC_NOT_IMPL` will be returned and an error will be logged.

The arguments are:

vm `ESMF_VM` object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

sendCount Number of `sendData` elements to send from local PET to all other PETs.

recvData Single data variable to be received. All PETs must specify a valid result variable.

recvCounts Number of `recvData` elements to be received from corresponding source PET.

recvOffsets Offsets in units of elements in `recvData` marking the start of element sequence to be received from source PET.

rootPet PET on which data is gathered.

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

45.5.15 ESMF_VMGet - Get VM internals

INTERFACE:

```
   ! Private name; call using ESMF_VMGet()
   subroutine ESMF_VMGetDefault(vm, localPet, petCount, &
     peCount, mpiCommunicator, pthreadsEnabledFlag, openMPEnabledFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_VM),      intent(in)            :: vm
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(out), optional :: localPet
     integer,            intent(out), optional :: petCount
     integer,            intent(out), optional :: peCount
     integer,            intent(out), optional :: mpiCommunicator
     logical,            intent(out), optional :: pthreadsEnabledFlag
     logical,            intent(out), optional :: openMPEnabledFlag
     integer,            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal information about the specified `ESMF_VM` object.

The arguments are:

vm Queried `ESMF_VM` object.

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

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

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

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

[pthreadsEnabledFlag] `.TRUE.` ESMF has been compiled with Pthreads.

`.FALSE.` ESMF has not been compiled with Pthreads.

[openMPEnabledFlag] `.TRUE.` ESMF has been compiled with OpenMP.

`.FALSE.` ESMF has not been compiled with OpenMP.

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

45.5.16 ESMF_VMGet - Get VM PET local internals

INTERFACE:

```
   ! Private name; call using ESMF_VMGet()
   subroutine ESMF_VMGetPetLocalInfo(vm, pet, peCount, ssiId, &
     threadCount, threadId, vas, rc)
```

ARGUMENTS:

```
     type(ESMF_VM), intent(in)            :: vm
     integer,       intent(in)            :: pet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,       intent(out), optional :: peCount
     integer,       intent(out), optional :: ssiId
     integer,       intent(out), optional :: threadCount
     integer,       intent(out), optional :: threadId
     integer,       intent(out), optional :: vas
     integer,       intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get internal information about a specific PET within an `ESMF_VM` object.

The arguments are:

vm Queried `ESMF_VM` object.

pet Queried PET id within the specified `ESMF_VM` object.

[peCount] Upon return this holds the number of PEs associated with the specified PET in the `ESMF_VM` object.

[ssiId] Upon return this holds the id of the single-system image (SSI) the specified PET is running on.

[threadCount] Upon return this holds the number of PETs in the specified PET"s thread group.

[threadId] Upon return this holds the thread id of the specified PET within the PET"s thread group.

[vas] Virtual address space in which this PET operates.

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

45.5.17 ESMF_VMGetGlobal - Get Global VM

INTERFACE:

```
   subroutine ESMF_VMGetGlobal(vm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get the global `ESMF_VM` object. This is the VM object that is created during `ESMF_Initialize()` and is the ultimate parent of all VM objects in an ESMF application. It is identical to the VM object returned by `ESMF_Initialize(..., vm=vm, ...)`.
The `ESMF_VMGetGlobal()` call provides access to information about the global execution context via the global VM. This call is necessary because ESMF does not created a global ESMF Component during

ESMF_Initialize() that could be queried for information about the global execution context of an ESMF application.
Usage of ESMF_VMGetGlobal() from within Component code is strongly discouraged. ESMF Components should only access their own VM objects through Component methods. Global information, if required by the Component user code, should be passed down to the Component from the driver through the Component calling interface.

The arguments are:

vm Upon return this holds the ESMF_VM object of the global execution context.

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

45.5.18 ESMF_VMGetCurrent - Get Current VM

INTERFACE:

```
   subroutine ESMF_VMGetCurrent(vm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

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

The arguments are:

vm Upon return this holds the ESMF_VM object of the current execution context.

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

45.5.19 ESMF_VMPrint - Print VM internals

INTERFACE:

```
   subroutine ESMF_VMPrint(vm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Print internal information about the specified `ESMF_VM` to `stdout`.

The arguments are:

vm Specified `ESMF_VM` object.

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

45.5.20 ESMF_VMRecv - Receive data from srcPet

INTERFACE:

```
    subroutine ESMF_VMRecv(vm, recvData, count, srcPet, &
      syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                  intent(in)            :: vm
      integer(ESMF_KIND_I4), target, intent(out)           :: recvData(:)
      integer,                        intent(in)            :: count
      integer,                        intent(in)            :: srcPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),           intent(in),  optional :: syncflag
      type(ESMF_CommHandle),          intent(out), optional :: commhandle
      integer,                        intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Receive contiguous data from `srcPet` within the same `ESMF_VM` object.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`, `ESMF_TYPEKIND_LOGICAL`, `ESMF_TYPEKIND_CHARACTER`.

The arguments are:

vm `ESMF_VM` object.

recvData Contiguous data array for data to be received.

count Number of elements to be received.

srcPet Sending PET.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.21 ESMF_VMReduce - Reduce data from across VM

INTERFACE:

```
    subroutine ESMF_VMReduce(vm, sendData, recvData, count, &
      reduceflag, rootPet, syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                      intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                            intent(in)            :: count
      type(ESMF_Reduce_Flag),             intent(in)            :: reduceflag
      integer,                            intent(in)            :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),               intent(in),  optional :: syncflag
      type(ESMF_CommHandle),              intent(out), optional :: commhandle
      integer,                            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that reduces a contiguous data array across the ESMF_VM object into a contiguous data array of the same <type><kind>. The result array is returned on rootPet. Different reduction operations can be specified.

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

TODO: The current version of this method does not provide an implementation of the *non-blocking* feature. When calling this method with syncflag = ESMF_SYNC_NONBLOCKING error code ESMF_RC_NOT_IMPL will be returned and an error will be logged.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. All PETs must specify a valid source array.

recvData Single data variable to be received. All PETs must specify a valid result variable.

count Number of elements in sendData and recvData. Must be the same on all PETs.

reduceflag Reduction operation. See section 9.35 for a list of valid reduce operations.

rootPet PET on which reduced data is returned.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.22 ESMF_VMScatter - Scatter data across VM

INTERFACE:

```
    subroutine ESMF_VMScatter(vm, sendData, recvData, count, &
      rootPet, syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                     intent(in)             :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)             :: sendData(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)            :: recvData(:)
      integer,                           intent(in)             :: count
      integer,                           intent(in)             :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),              intent(in),  optional :: syncflag
      type(ESMF_CommHandle),             intent(out), optional :: commhandle
      integer,                           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that scatters contiguous data from the rootPet to all PETs across the ESMF_VM object (including rootPet).

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8, ESMF_TYPEKIND_LOGICAL.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. Only the sendData array specified by the rootPet will be used by this method.

recvData Contiguous data array for data to be received. All PETs must specify a valid destination array.

count Number of elements to be send from rootPet to each of the PETs. Must be the same on all PETs.

rootPet PET that holds data that is being scattered.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.23 ESMF_VMScatterV - ScatterV across VM

INTERFACE:

```
    subroutine ESMF_VMScatterV(vm, sendData, sendCounts, &
      sendOffsets, recvData, recvCount, rootPet, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                      intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      integer,                            intent(in)            :: sendCounts(:)
      integer,                            intent(in)            :: sendOffsets(:)
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                            intent(in)            :: recvCount
      integer,                            intent(in)            :: rootPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Collective ESMF_VM communication call that scatters contiguous data from the rootPet to all PETs across the ESMF_VM object (including rootPet).

This method is overloaded for: ESMF_TYPEKIND_I4, ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send. Only the sendData array specified by the rootPet will be used by this method.

sendCounts Number of sendData elements to be send to corresponding receive PET.

sendOffsets Offsets in units of elements in sendData marking the start of element sequence to be send to receive PET.

recvData Single data variable to be received. All PETs must specify a valid result variable.

recvCount Number of recvData elements to receive by local PET from rootPet.

rootPet PET that holds data that is being scattered.

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

45.5.24 ESMF_VMSend - Send data to dstPet

INTERFACE:

```
    subroutine ESMF_VMSend(vm, sendData, count, dstPet, &
      syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                     intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      integer,                           intent(in)            :: count
      integer,                           intent(in)            :: dstPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),              intent(in),  optional :: syncflag
      type(ESMF_CommHandle),             intent(out), optional :: commhandle
      integer,                           intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Send contiguous data to dstPet within the same ESMF_VM object.

The arguments are:

vm ESMF_VM object.

sendData Contiguous data array holding data to be send.

count Number of elements to be send.

dstPet Receiving PET.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is ESMF_SYNC_BLOCKING. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument syncflag). The commhandle can be used in ESMF_VMCommWait() to block the calling PET until the communication call has finished PET-locally. If no commhandle was supplied to a non-blocking call the VM method ESMF_VMCommWaitAll() may be used to block on all currently queued communication calls of the VM context.

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

45.5.25 ESMF_VMSendRecv - Send and Recv data to and from PETs

INTERFACE:

```
    subroutine ESMF_VMSendRecv(vm, sendData, sendCount, dstPet, &
      recvData, recvCount, srcPet, syncflag, commhandle, rc)
```

ARGUMENTS:

```
      type(ESMF_VM),                     intent(in)            :: vm
      <type>(ESMF_KIND_<kind>), target, intent(in)            :: sendData(:)
      integer,                          intent(in)            :: sendCount
      integer,                          intent(in)            :: dstPet
      <type>(ESMF_KIND_<kind>), target, intent(out)           :: recvData(:)
      integer,                          intent(in)            :: recvCount
      integer,                          intent(in)            :: srcPet
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Sync_Flag),             intent(in),  optional :: syncflag
      type(ESMF_CommHandle),            intent(out), optional :: commhandle
      integer,                          intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Send contiguous data to `dstPet` within the same `ESMF_VM` object while receiving contiguous data from `srcPet` within the same `ESMF_VM` object. The `sendData` and `recvData` arrays must be disjoint!

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`, `ESMF_TYPEKIND_LOGICAL`, `ESMF_TYPEKIND_CHARACTER`.

The arguments are:

vm `ESMF_VM` object.

sendData Contiguous data array holding data to be send.

sendCount Number of elements to be send.

dstPet PET that holds `recvData`.

recvData Contiguous data array for data to be received.

recvCount Number of elements to be received.

srcPet PET that holds `sendData`.

[syncflag] Flag indicating whether this call behaves blocking or non-blocking. The default is `ESMF_SYNC_BLOCKING`. See section 9.44 for a complete list of options.

[commhandle] If present, a communication handle will be returned in case of a non-blocking request (see argument `syncflag`). The `commhandle` can be used in `ESMF_VMCommWait()` to block the calling PET until the communication call has finished PET-locally. If no `commhandle` was supplied to a non-blocking call the VM method `ESMF_VMCommWaitAll()` may be used to block on all currently queued communication calls of the VM context.

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

45.5.26 ESMF_VMValidate - Validate VM internals

INTERFACE:

```
   subroutine ESMF_VMValidate(vm, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Validates that the `vm` is internally consistent. The method returns an error code if problems are found.
The arguments are:

vm Specified `ESMF_VM` object.

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

45.5.27 ESMF_VMWtime - Get floating-point number of seconds

INTERFACE:

```
  subroutine ESMF_VMWtime(time, rc)
```

ARGUMENTS:

```
    real(ESMF_KIND_R8), intent(out)           :: time
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get floating-point number of seconds of elapsed wall-clock time since some time in the past.

The arguments are:

time Time in seconds.

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

45.5.28 ESMF_VMWtimeDelay - Delay execution

INTERFACE:

```
  subroutine ESMF_VMWtimeDelay(delay, rc)
```

ARGUMENTS:

```
    real(ESMF_KIND_R8), intent(in)            :: delay
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Delay execution for amount of seconds.

The arguments are:

delay Delay time in seconds.

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

45.5.29 ESMF_VMWtimePrec - Timer precision as floating-point number of seconds

INTERFACE:

```
   subroutine ESMF_VMWtimePrec(prec, rc)
```

ARGUMENTS:

```
     real(ESMF_KIND_R8), intent(out)           :: prec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Get a run-time estimate of the timer precision as floating-point number of seconds. This is a relatively expensive call since the timer precision is measured several times before the maximum is returned as the estimate. The returned value is PET-specific and may differ across the VM context.

The arguments are:

prec Timer precision in seconds.

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

46 Fortran I/O and System Utilities

46.1 Description

The ESMF Fortran I/O utilities provide portable methods to access capabilities which are often implemented in different ways amongst different environments. Currently, two utility methods are implemented: `ESMF_IOUnitGet()`, to find an unopened unit number within the range of unit numbers that ESMF is allowed to use, and `ESMF_IOUnitFlush()` to flush the I/O buffer associated with a specific Fortran unit.

46.2 Use and Examples

46.2.1 Fortran unit number management

The `ESMF_UtilIOUnitGet()` method is provided so that applications using ESMF can remain free of unit number conflicts — both when combined with other third party code, or with ESMF itself. This call is typically used just prior to an `OPEN` statement:

```
  call ESMF_UtilIOUnitGet (unit=grid_unit, rc=rc)
  open (unit=grid_unit, file='grid_data.dat', status='old', action='read')
```

By default, unit numbers between 50 and 99 are scanned to find an unopened unit number.
Internally, ESMF also uses `ESMF_UtilIOUnitGet()` when it needs to open Fortran unit numbers for file I/O. By using the same API for both user and ESMF code, unit number collisions can be avoided.
When integrating ESMF into an application where there are conflicts with other uses of the same unit number range, such as when hard-coded unit number values are used, an alternative unit number range can be specified. The

ESMF_Initialize() optional arguments IOUnitLower and IOUnitUpper may be set as needed. Note that IOUnitUpper must be set to a value higher than IOUnitLower, and that both must be non-negative. Otherwise ESMF_Initialize will return a return code of ESMF_FAILURE. ESMF itself does not typically need more than about five units for internal use.

```
call ESMF_Initialize (..., IOUnitLower=120, IOUnitUpper=140)
```

All current Fortran environments have preconnected unit numbers, such as units 5 and 6 for standard input and output, in the single digit range. So it is recommended that the unit number range is chosen to begin at unit 10 or higher to avoid these preconnected units.

46.2.2 Flushing output

Fortran run-time libraries generally use buffering techniques to improve I/O performance. However output buffering can be problematic when output is needed, but is "trapped" in the buffer because it is not full. This is a common occurance when debugging a program, and inserting WRITE statements to track down the bad area of code. If the program crashes before the output buffer has been flushed, the desired debugging output may never be seen — giving a misleading indication of where the problem occurred. It would be desirable to ensure that the output buffer is flushed at predictable points in the program in order to get the needed results. Likewise, in parallel code, predictable flushing of output buffers is a common requirement, often in conjunction with ESMF_VMBarrier() calls.
The ESMF_UtilIOUnitFlush() API is provided to flush a unit as desired. Here is an example of code which prints debug values, and serializes the output to a terminal in PET order:

```
type(ESMF_VM) :: vm

integer :: tty_unit
integer :: me, npets

call ESMF_Initialize (vm=vm, rc=rc)
call ESMF_VMGet (vm, localPet=me, petCount=npes)

call ESMF_UtilIOUnitGet (unit=tty_unit)
open (unit=tty_unit, file='/dev/tty', status='old', action='write')
...
call ESMF_VMBarrier (vm=vm)
do, i=0, npets-1
  if (i == me) then
    write (tty_unit, *) 'PET: ', i, ', values are: ', a, b, c
    call ESMF_UtilIOUnitFlush (unit=tty_unit)
  end if
  call ESMF_VMBarrier (vm=vm)
end do
```

46.3 Design and Implementation Notes

46.3.1 Fortran unit number management

When ESMF needs to open a Fortran I/O unit, it calls ESMF_IOUnitGet() to find an unopened unit number. As delivered, the range of unit numbers that are searched are between ESMF_LOG_FORTRAN_UNIT_NUMBER (normally set to 50), and ESMF_LOG_UPPER (normally set to 99.) Unopened unit numbers are found by using the Fortran INQUIRE statement.
When integrating ESMF into an application where there are conflicts with other uses of the same unit number range, an alternative range can be specified in the ESMF_Initialize() call by setting the IOUnitLower and IOUnitUpper arguments as needed. ESMF_IOUnitGet() will then search the alternate range of unit numbers. Note that IOUnitUpper must be set to a value higher than IOUnitLower, and that both must be non-negative. Otherwise ESMF_Initialize will return a return code of ESMF_FAILURE.

Fortran unit numbers are not standardized in the Fortran 90 Standard. The standard only requires that they be non-negative integers. But other than that, it is up to the compiler writers and application developers to provide and use units which work with the particular implementation. For example, units 5 and 6 are a defacto standard for "standard input" and "standard output" — even though this is not specified in the actual Fortran standard. The Fortran standard also does not specifiy which unit numbers can be used, nor does it specify how many can be open simultaneously.
Since all current compilers have preconnected unit numbers, and these are typically found on units lower than 10, it is recommended that applications use unit numbers 10 and higher.

46.3.2 Flushing output

When ESMF needs to flush a Fortran unit, the `ESMF_IOUnitFlush()` API is used to centralize the file flushing capability, because Fortran has not historically had a standard mechanism for flushing output buffers. Most compilers run-time libraries support various library extensions to provide this functionality — though, being non-standard, the spelling and number of arguments vary between implementations. Fortran 2003 also provides for a `FLUSH` statement which is built into the language. When possible, `ESMF_IOUnitFlush()` uses the F2003 `FLUSH` statement. With older compilers, the appropriate library call is made.

46.4 Utility API

46.4.1 ESMF_UtilGetArg - Return a command line argument

INTERFACE:

```
   subroutine ESMF_UtilGetArg(argindex, argvalue, arglength, rc)
```

ARGUMENTS:

```
     integer,      intent(in)            :: argindex
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*), intent(out), optional :: argvalue
     integer,      intent(out), optional :: arglength
     integer,      intent(out), optional :: rc
```

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method returns a copy of a command line argument specified when the process was started. This argument is the same as an equivalent C++ program would find in the argv array.
Some MPI implementations do not consistently provide command line arguments on PETs other than PET 0. It is therefore recommended that PET 0 call this method and broadcast the results to the other PETs by using the `ESMF_VMBroadcast()` method.
The arguments are:

argindex A non-negative index into the command line argument `argv` array. If argindex is negative or greater than the number of user-specified arguments, `ESMF_RC_ARG_VALUE` is returned in the `rc` argument.

[argvalue] Returns a copy of the desired command line argument. If the provided character string is longer than the command line argument, the string will be blank padded. If the string is too short, truncation will occur and `ESMF_RC_ARG_SIZE` is returned in the `rc` argument.

[arglength] Returns the length of the desired command line argument in characters. The length result does not depend on the length of the `value` string. It may be used to query the length of the argument.

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

46.4.2 ESMF_UtilGetArgC - Return number of command line arguments

INTERFACE:

```
   subroutine ESMF_UtilGetArgC(count, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method returns the number of command line arguments specified when the process was started.
The number of arguments returned does not include the name of the command itself - which is typically returned as argument zero.
Some MPI implementations do not consistently provide command line arguments on PETs other than PET 0. It is therefore recommended that PET 0 call this method and broadcast the results to the other PETs by using the `ESMF_VMBroadcast()` method.
The arguments are:

count Count of command line arguments.

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

46.4.3 ESMF_UtilGetArgIndex - Return the index of a command line argument

INTERFACE:

```
   subroutine ESMF_UtilGetArgIndex(argvalue, argindex, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

This method searches for, and returns the index of a desired command line argument. An example might be to find a specific keyword (e.g., -esmf_path) so that its associated value argument could be obtained by adding 1 to the argindex and calling `ESMF_UtilGetArg()`.
Some MPI implementations do not consistently provide command line arguments on PETs other than PET 0. It is therefore recommended that PET 0 call this method and broadcast the results to the other PETs by using the `ESMF_VMBroadcast()` method.
The arguments are:

argvalue A character string which will be searched for in the command line argument list.

[argindex] If the `value` string is found, the position will be returned as a non-negative integer. If the string is not found, a negative value will be returned.

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

46.4.4 ESMF_UtilIOUnitFlush - Flush output on a unit number

INTERFACE:

```
   subroutine ESMF_UtilIOUnitFlush(unit, rc)
```

PARAMETERS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Call the system-dependent routine to force output on a specific Fortran unit number.
The arguments are:

unit A Fortran I/O unit number.

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

46.4.5 ESMF_UtilIOUnitGet - Scan for a free I/O unit number

INTERFACE:

```
   subroutine ESMF_UtilIOUnitGet(unit, rc)
```

ARGUMENTS:

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

STATUS: Backward compatible starting with ESMF 5.2.0r.

DESCRIPTION:

Scan for, and return, a free Fortran I/O unit number. By default, the range of unit numbers returned is between 50 and 99 (parameters `ESMF_LOG_FORTRAN_UNIT_NUMBER` and `ESMF_LOG_UPPER` respectively.) When integrating ESMF into an application where these values conflict with other usages, the range of values may be moved by setting the optional `IOUnitLower` and `IOUnitUpper` arguments in the initial `ESMF_Initialize()` call with values in a safe, alternate, range.
The Fortran unit number which is returned is not reserved in any way. Successive calls without intervening `OPEN` or `CLOSE` statements (or other means of connecting to units), might not return a unique unit number. It is recommended that an `OPEN` statement immediately follow the call to `ESMF_IOUnitGet()` to activate the unit.
The arguments are:

unit A Fortran I/O unit number.

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

Part VI
References

References

[1] A Julian Day and Civil Date Calculator. http://www.numerical-recipes.com/julian.html.

[2] Some notes on the ISO 8601 date and time specification standard. http://en.wikipedia.org/wiki/ISO_8601 http://www.iso.ch/iso/en/prods-services/popstds/datesandtime.html.

[3] International Organization for Standardization. http://www.iso.org/iso/home.html, last accessed on May 16, 2011.

[4] METAFOR - Developing the CIM. http://metaforclimate.eu/Table/Work-Package-2/Developing-the-CIM/, last accessed on May 16, 2011.

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

[6] Eaton, B., J. Gregory, B. Drach, K. Taylor, and S. Hankin. NetCDF Climate and Forecast (CF) Metadata Convention. http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html.

[7] *Extensible Markup Language (XML)*. The World Wide Web Consortium (W3C). http://www.w3.org/XML/.

[8] Fliegel, H.F. and Van Flandern, T.C. A Machine Algorithm for Processing Calendar Dates. *Communications of the ACM*, 11(10):657, 1968.

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.

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

[11] Hatcher, D.A. Simple Formulae for Julian Day Numbers and Calendar Dates. *Q.JlR. astr. Soc.*, 25(1):53–55, 1984.

[12] International Organization for Standardization. Standard 8601:2004, Data elements and interchange formats – Information interchange – Representation of dates and times. http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=40874&COMMID=&scopelist=.

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

[14] William Kahan. *Documents relating to IEEE standard 754 for binary floating-point arithmetic*. University of California, Berkeley. http://HTTP.CS.Berkeley.EDU/ wkahan/ieee754status/.

[15] Meyer, Peter. A good discussion of Gregorian and Julian Calendars. http://www.hermetic.ch/cal_stud/cal_art.html.

[16] Meyer, Peter. A good discussion of Julian Day Numbers. http://www.hermetic.ch/cal_stud/jdn.htm.

[17] *NetCDF Climate and Forecast (CF) Metadata Conventions, Version 1.0-beta3*, August 2001. http://www.cgd.ucar.edu/cms/eaton/netcdf/CF-current.htm.

[18] *NetCDF User's Guide for C, Version 3*. Unidata Program Center, Boulder, Colorado, June 1997. http://www.unidata.ucar.edu/packages/netcdf/guidec/.

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

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

[21] Seidelman, P.K. *Explanatory Supplement to the Astronomical Almanac*. University Science Books, 1992.

[22] Isaac Held Michael Winton Jeff Durachta Sergey Malyshev V. Balaji, Jeff Anderson and Ronald J. Stouffer. The exchange grid: a mechanism for data exchange between earth system components on independent grids. *Parallel Computational Fluid Dynamics: Theory and Applications, Proceedings of the 2005 International Conference on Parallel Computational Fluid Dynamics*, 2006.

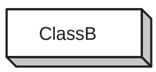
[23] Winkler, Gernot M.R. A good discussion of the Modified Julian Day Calendar. http://tycho.usno.navy.mil/mjd.html.

Part VII
Appendices

47 Appendix A: A Brief Introduction to UML

The schematic below shows the Unified Modeling Language (UML) notation for the class diagrams presented in this *Reference Manual*. For more on UML, see references such as *The Unified Modeling Language Reference Manual*, Rumbaugh et al, [20].

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



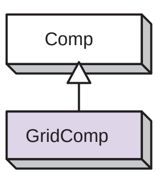
Private class. This type of class does not have methods that should be called by the user. Like a public class it is usually associated with a derived type and a corresponding module.

A line indicates some sort of association among classes.

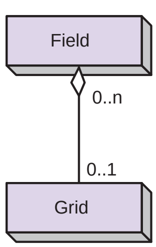
A hollow diamond at one end of a line drawn between classes represents an association called aggregation. Aggregation is a part-whole relationship that can be read as “the class at the end of the line without the diamond is part of the class at the end of the line with the diamond.” The class that is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

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

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



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



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

48 Appendix B: ESMF Error Return Codes

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

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

 ESMF_SUCCESS                0
 ESMF_RC_OBJ_BAD             1
 ESMF_RC_OBJ_INIT            2
 ESMF_RC_OBJ_CREATE          3
 ESMF_RC_OBJ_COR             4
 ESMF_RC_OBJ_WRONG           5
 ESMF_RC_ARG_BAD             6
 ESMF_RC_ARG_RANK            7
 ESMF_RC_ARG_SIZE            8
 ESMF_RC_ARG_VALUE           9
 ESMF_RC_ARG_DUP            10
 ESMF_RC_ARG_SAMETYPE       11
 ESMF_RC_ARG_SAMECOMM       12
 ESMF_RC_ARG_INCOMP         13
 ESMF_RC_ARG_CORRUPT        14
 ESMF_RC_ARG_WRONG          15
 ESMF_RC_ARG_OUTOFRANGE     16
 ESMF_RC_ARG_OPT            17
 ESMF_RC_NOT_IMPL           18
 ESMF_RC_FILE_OPEN          19
 ESMF_RC_FILE_CREATE        20
 ESMF_RC_FILE_READ          21
 ESMF_RC_FILE_WRITE         22
 ESMF_RC_FILE_UNEXPECTED    23
 ESMF_RC_FILE_CLOSE         24
 ESMF_RC_FILE_ACTIVE        25
 ESMF_RC_PTR_NULL           26
 ESMF_RC_PTR_BAD            27
 ESMF_RC_PTR_NOTALLOC       28
 ESMF_RC_PTR_ISALLOC        29
 ESMF_RC_MEM                30
 ESMF_RC_MEM_ALLOCATE       31
 ESMF_RC_MEM_DEALLOCATE     32
 ESMF_RC_MEMC               33
 ESMF_RC_DUP_NAME           34
 ESMF_RC_LONG_NAME          35
 ESMF_RC_LONG_STR           36
 ESMF_RC_COPY_FAIL          37
 ESMF_RC_DIV_ZERO           38
 ESMF_RC_CANNOT_GET         39
 ESMF_RC_CANNOT_SET         40
 ESMF_RC_NOT_FOUND          41
 ESMF_RC_NOT_VALID          42
 ESMF_RC_INTNRL_LIST        43
 ESMF_RC_INTNRL_INCONS      44
 ESMF_RC_INTNRL_BAD         45
 ESMF_RC_SYS                46
```

```
 ESMF_RC_BUSY              47
 ESMF_RC_LIB               48
 ESMF_RC_LIB_NOT_PRESENT   49
 ESMF_RC_ATTR_UNUSED       50
 ESMF_RC_OBJ_NOT_CREATED   51
 ESMF_RC_OBJ_DELETED       52
 ESMF_RC_NOT_SET           53
 ESMF_RC_VAL_WRONG         54
 ESMF_RC_VAL_ERRBOUND      55
 ESMF_RC_VAL_OUTOFRANGE    56
 ESMF_RC_ATTR_NOTSET       57
 ESMF_RC_ATTR_WRONGTYPE    58
 ESMF_RC_ATTR_ITEMSOFF     59
 ESMF_RC_ATTR_LINK         60
 ESMF_RC_BUFFER_SHORT      61

62-499 reserved for future Fortran symmetric return code definitions

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

 ESMC_RC_OBJ_BAD          501
 ESMC_RC_OBJ_INIT         502
 ESMC_RC_OBJ_CREATE       503
 ESMC_RC_OBJ_COR          504
 ESMC_RC_OBJ_WRONG        505
 ESMC_RC_ARG_BAD          506
 ESMC_RC_ARG_RANK         507
 ESMC_RC_ARG_SIZE         508
 ESMC_RC_ARG_VALUE        509
 ESMC_RC_ARG_DUP          510
 ESMC_RC_ARG_SAMETYPE     511
 ESMC_RC_ARG_SAMECOMM     512
 ESMC_RC_ARG_INCOMP       513
 ESMC_RC_ARG_CORRUPT      514
 ESMC_RC_ARG_WRONG        515
 ESMC_RC_ARG_OUTOFRANGE   516
 ESMC_RC_ARG_OPT          517
 ESMC_RC_NOT_IMPL         518
 ESMC_RC_FILE_OPEN        519
 ESMC_RC_FILE_CREATE      520
 ESMC_RC_FILE_READ        521
 ESMC_RC_FILE_WRITE       522
 ESMC_RC_FILE_UNEXPECTED  523
 ESMC_RC_FILE_CLOSE       524
 ESMC_RC_FILE_ACTIVE      525
 ESMC_RC_PTR_NULL         526
 ESMC_RC_PTR_BAD          527
 ESMC_RC_PTR_NOTALLOC     528
 ESMC_RC_PTR_ISALLOC      529
 ESMC_RC_MEM              530
 ESMC_RC_MEM_ALLOCATE     531
 ESMC_RC_MEM_DEALLOCATE   532
 ESMC_RC_MEMC             533
```

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

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

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

 ESMC_RC_OPTARG_BAD       1000
```