An Introduction to MAPL

Max Suarez, Atanas Trayanov, Arlindo da Silva, Chris Hill, Paul Schopf

June 12, 2007

1 Introduction

This document describes MAPL, a software layer that establishes usage standards and software tools for building ESMF compliant components. This package (1) facilitates the porting of existing codes to ESMF, (2) provides tools and a straightforward recipe for building new ESMF components, and (3) provides much greater interoperability between compliant components than between current ESMF compliant components.

As the Earth System Modeling Framework (ESMF) has become available, several groups have been involved in prototyping its use in climate and weather prediction models and in data assimilation systems. Existing programs have been converted to use the superstructure of the framework at MIT, NCAR, GFDL, Goddard, NCEP and the DoD. (see http://www.esmf.ucar.edu/impacts). One of the most complete attempts to use ESMF has been the development of the GEOS-5 AGCM, a model targeted by the MAP announcement. GEOS-5 has been built “from the ground up” using the latest available versions of ESMF superstructure and infrastructure.

All of these efforts have produced much constructive feedback to the ESMF core development team, and have helped refine the design and improve the implementation of the framework. They have also served to identify the most important directions for future extensions. Comparing the various implementations leads to two seemingly contradictory conclusions: that all implementations are different and that much of what they do is the same. Both conclusions were anticipated, since ESMF is a general framework designed to meet a wide variety of needs. This generality is an important strength of the ESMF design, but it also implies that there are many different ways of using ESMF—even when performing very similar tasks. Other observations from this early experience were that each group, within its own implementations, repeatedly needed functions that provided higher level functionality than that provided by the basic ESMF tools, and that the core methods of ESMF components (Run, Initialize, and Finalize) looked very similar in all their implementations.

The MAPL package arose as a response to this early experience, particularly during the construction of GEOS-5. It is based on the observation that much of the work done in these initial implementations can be standardized; thus, reducing the labor of constructing
ESMF applications in the future, as well as increasing their interoperability. In its initial implementation, MAPL provides:

- Specific conventions and best practices for the utilization of ESMF in climate models
- A middleware layer (i.e., between the model and ESMF) that facilitates the adoption of ESMF by climate models.

This enhancement in usability of ESMF must come at the cost of reduced generality. To make the framework more usable for our applications, we make assumptions and place requirements on the applications that ESMF, with its goal of generality, could not. MAPL does this “on top of” ESMF and as a separate layer through which the application uses ESMF for some of its functions (although for most things, applications will continue to use ESMF directly). We feel that this middleware-layer approach is the right way to get the usability and interoperability that climate model components require of the framework, without sacrificing ESMF’s generality and extensibility.

2 Review of Relevant Aspects of ESMF

The Earth System Modeling Framework (ESMF) (DeLuca et al, 2004) is a software package designed to provide some of essential functions needed by parallel, scalable earth system models in a machine-independent way. ESMF is implemented as a collection of very general programming classes that can be used both to construct ESMF components (infrastructure layer) and to connect them to one another (superstructure layer). These classes thus support modelers in building interoperable and portable codes. This design is illustrated by the ESMF “sandwich” diagram, where the user’s computational code sits between the two ESMF layers.

The simplest ESMF implementations consist of building a Gridded Component (an ESMF superstructure class) that encapsulates the user code, interfacing it to the framework by defining the ESMF callable methods (Initialize, Run and Finalize, hereafter, IRF methods). This can be done making little or no use of the ESMF Infrastructure—a strategy that fails to capitalize on some of ESMF’s greatest strengths. Such “encapsulation” implementations have dominated the early adoptions of ESMF.

More sophisticated implementations put user data in ESMF Infrastructure objects (primarily ESMF.Fields) which can then be manipulated by a wide array of ESMF methods to facilitate the coupling of components with different data structures (i.e., that are on different grids) and to insulate the user from the architecture-specific implementation layers that are used for inter-process or inter-processor communication, I/O, etc.

In designing ESMF, a deliberate decision was made to have the framework provide these services in a very general way, and not to prejudge how future models would use it or what programming models would best suit future computer architectures. This generality is an important strength of ESMF, but it is also an impediment to many users that would prefer
a more specific formulation for porting existing codes or a better defined recipe for building new codes with ESMF. The generality also impacts the interoperability of applications, since the ESMF interfaces to the IRF methods are general purpose, and they carry little information (other than the grid definition) about the physical content of the data moving in and out of the gridded component.

The middleware layer implemented in MAPL includes the following design elements:

1. Provide easy-to-use tools for describing the contents of a component’s import and export states, as well as adopting conventions for what must be described. But in no way specifying what the contents must be.

2. Facilitate the coupling of components into complex applications. This requires a means of describing the connectivity between components and of using the description of the import and export states to couple components.

3. Provide aids in constructing a component’s IRF methods

4. Facilitate the use of ESMF.Fields and thus of the ESMF Infrastructure layer

5. Extend the ESMF.State concept to a component’s internal state, and help it manage its persistent data.

Two other ESMF concepts are worth highlighting: the hierarchical organization of gridded components and the coupler component. ESMF gridded components can be, as described above, simple containers for user code. But ESMF also allows gridded components to contain other gridded components. These composite components can themselves contain user code or can simply serve as a way of grouping closely related components. The notion of composite components allows a straightforward way of organizing applications as a hierarchy of components. ESMF does not require a hierarchical organization, but it is the most natural way of connecting ESMF components. ESMF also defines the notion of Coupler Components. These are similar to gridded components, but are not intended for user code; rather, they house the transformations necessary to convert between exports of one component and imports of another. MAPL adopts the hierarchical organization as its architecture for making complex applications and uses both composite gridded components and ESMF coupler components to establish connections between members of the hierarchy.

3 Overview of MAPL

We organize the MAPL description around how it deals with the relevant design elements discussed, even though there is not a one-to-one correspondence between these elements and the actual MAPL code.

The first of these we refer to as MAPL-Core. This includes those elements useful in building a single MAPL gridded component. In particular, it includes the means of describing a component’s Import and Exports states. Having a component built in this ways, with well
described states, allows us to use the second element, MAPL.Connect to organize them into a hierarchy. The distinction between these two elements, which in the MAPL code are mostly within the MAPL.Generic module, is important. One may use MAPL.Core alone as a means of facilitating the introduction of ESMF, with no intention of ever coupling the component to a MAPL hierarchy. A component so constructed is a perfectly good ESMF component, and other than having to access the MAPL library to build and execute, it is not special in any way. The code in an application instanciating it would not need to know it was built with MAPL machinery.

A third element is MAPL.History, which is an ESMF gridded component that sits inside MAPL and can be instanciated to provide data writing services for a MAPL hierarchy. MAPL.Utils is a set of support utilities for tasks commonly needed in global models. MAPL itself uses some of these, but, like MAPL.History, MAPL components or application need not use them.

Finally, MAPL.CFIO is a partial I/O layer for ESMF components. It relies internally on MAPL, and so must be built with it, but it can be used even if one is not using the rest of MAPL. MAPL.History is built on top of MAPL.CFIO.

### 3.1 Building a MAPL Gridded Component: MAPL.Core

MAPL’s initial intention, and still its core function, is to provide assistance in writing ESMF Gridded components. It does this in the following ways:

- It makes it easier to write the component’s IRF methods. In fact in some cases they need not be written at all.
- It adds an Internal ESMF State to the component, supplementing the Import and Export states required by ESMF.
- It provides a means of describing the contents of the three states so that MAPL can help manage them.
- It adopts groundrules for the behavior of a component and its treatment of the three states.
- It defines a standard recipe for writing MAPL-based ESMF Gridded Components.

**Writing the IRF method:** After writing a few gridded components, one realizes that, except for the actual insertion of the user code, most SetServices and IRF methods are very similar, and that it would be economical to generalize this “boilerplate” code. MAPL provides three ways of doing this.

The first way is to use the generic versions of SetServices and of the three IRF methods provided by MAPL as the component’s methods. When MAPL_GenericSetServices is invoked, it registers the three Generic IRF methods. If not overridden, these become the component’s actual methods.
A second way of using \textit{MAPL} is to simply call the generic versions of the methods from the component-specific versions, allowing them to perform the boilerplate functions.

A third way is to simply use the source code of the generic versions as templates for the specific versions. Taking this approach is dangerous and not allowed for \textit{MAPL}-compliant components.

So what do the Generic IRF methods do? This will be described in detail in subsequent sections, but simply stated, they manage the IM/EX States and a third \texttt{ESMF\_State} that we will be discussing below, the \texttt{Generic\_Internal} state. We will refer to these three state as the IM/EX/IN states. Note that they are all ordinary \texttt{ESMF\_States}. From the description of the three states provided in the data services, \textit{MAPL} is able to create, allocate, initialize, and destroy all items in these states; it can also checkpoint and restart the internal and import states. The IRF methods also implement the connectivity of children components, creating the appropriate couplers, registering their services, and executing their IRF methods.

The new Internal State: In the spirit of having as unintrusive a design as possible, ESMF says nothing about a component’s internal state. But, since it is desirable that gridded components themselves be as object-oriented as possible, the framework has to allow them to be fully instanciable. This requires that whatever the component defines as its internal state be attached to the \texttt{ESMF\_GriddedComponent} instance. ESMF provides such a mechanism—effectively a hook on which a component can hang the current instance of its internal state.

\textit{MAPL} maintains this approach, but in addition allows the component to place some parts of its true internal state in an \texttt{ESMF\_State} analogous to the IM/EX states. This new state does not appear explicitly in the argument list of IRF methods, as is the case with the IM/EX states; instead it is attached to the \texttt{ESMF\_GriddedComponent} and, in principle, is accessible only through \textit{MAPL}.

All of the mechanisms for registering and manipulating data that are already available in \textit{MAPL} for the IM/EX States, are extended to the new Internal state. The default accessibility rules for this state are that its items can be written only by the component and can be read only by its parent. All data registered in this state by the component’s \texttt{SetServices} are, of course, automatically allocated, checkpointed, and restarted by the \textit{MAPL} Initialize and Finalize methods.

\textit{Description of State contents:} As discussed above, the simplest ESMF gridded component consists of the IRF methods encapsulating the user’s computational code. These methods are private to the component, but are callable by the framework; in fact, they can only be called by the framework. This is accomplished by having in each component a public method (\texttt{SetServices}) that tells the framework what functions it can perform (initialize the component, run it, etc.). The framework can then invoke these functional services when they are required.

The interface to these services is prescribed by ESMF and includes Import and Export (IM/EX) States, through which all data exchange between the components occurs. These states can contain only ESMF objects (primarily \texttt{ESMF\_Fields} and other \texttt{ESMF\_States}), but
ESMF says nothing about how they are to be used. *MAPL* assumes that IM/EX states consist only of Fields and other States. It also adopts the convention that, by default, items in its Export state are not modified by other components and that it cannot modify items in its Import state.

A major innovation in *MAPL* is a means of describing the contents of the IM/EX states. *MAPL* takes the view that a component, in addition to giving the framework access to its functional services, should also tell the framework about its data services, i.e., what it needs from others and what it can provide. *MAPL* extends the use of *SetServices* to accomplish this. The *SetServices* method of a *MAPL*-based gridded component will contain *spec calls* like the following:

```call MAPL_AddImportSpec(STATE, &
    SHORT_NAME = 'PLE', &
    LONG_NAME = 'air_pressure', &
    UNITS = 'Pa', &
    DIMS = MAPL_DimsHorzVert, &
    VLOCATION = MAPL_VLocationEdge, &
                    RC=STATUS )
```

Note that some of the attributes being set for this Field, such as units, are likely reflect assumptions made by the component and are usually static; others may be set at run time, say from a configuration file.

The information provided in setting data services is used by *MAPL*, to allocate and initialize the states, to couple to other components, and to help build the component’s IRF methods, as we will see below.

*Rules for Components:*

The first thing to clarify is what we mean by a *MAPL*-based ESMF_GriddedComponent. The following general rules apply to *MAPL*-compliant components:

**Rule 1** The component must be a fully-compliant ESMF_GriddedComponent. This implies that its only public object is *SetServices* and it registers IRF methods with ESMF.

**Rule 2** Associated with each instance of a *MAPL*-compliant ESMF_GriddedComponent there is an ESMF_Grid that *MAPL* will use to allocate data.

**Rule 3** Every ESMF_GriddedComponent has a configuration. A *MAPL* gridded component will expect it to be open when *SetServices* is called.

**Rule 4** Components can be run sequentially or concurrently; however, their Run methods must return control at RUN_DT intervals.

**Rule 5** A *MAPL*-compliant ESMF_GriddedComponent can be simple or composite.

**Rule 6** The component must obey all *MAPL* rules pertaining to its grid, as defined below.
Rule 7 The component must obey all MAPL access rules to the IM/EX/IN states, as defined below.

Rule 8 The MAPL_GenericSetServices, MAPL_GenericInitialize, and MAPL_GenericFinalize methods must be invoked once, and only once, for each instance of the grid-ded component.

Rule 9 Component instances must have unique names of the form: “first[:last]”. Neither first nor last name can have a colon. Example: Ens01:TURBULENCE.

The following Fortran 95 module shows the simplest MAPL component. This is a fully-compliant ESMF_GriddedComponent. It has a public SetServices taken from MAPL, and this is its only public object. Of course, it does nothing; but it can be run as a null component anywhere an ESMF_GriddedComponent can be run. Since it uses the generic IRF methods, it has a single stage of each. The rules about grids and states are not too relevant, but it has a natural grid—the ESMF_Grid we assume was given to it when the instance of the ESMF_GriddedComponent is created. It has IM/EX/IN states, which are silently created by the implicit generic methods; but all three state are empty.

3.2 Example 1: Using the Generic Component

MAPL has built-in ESMF_GriddedComponents. The most fundaments of these is the MAPL_Generic component, whose SetServices and IRF methods we normally use in building other components. It is possible, however, to instantiate MAPL_Generic itself. Currently such an instance is useful only as a null leaf component, which does nothing. Nevertheless, it is a perfectly valid ESMF_GriddedComponent. The example is a main program that runs MAPL_Generic for a year. It also illustrates the basic steps that an ESMF main program (or Cap) contains. As one can see, even this basic program is fairly complex. We will see later how MAPL can also help in reducing this complexity.

Program Example1

```fortran
use ESMF
use MAPL :: only, SetServices->MAPL_GenericSetServices

type(ESMF_GridComp) :: GC
type(ESMF_State) :: Import, Export
type(ESMF_Clock) :: Clock
type(ESMF_Time) :: StartTime
type(ESMF_Time) :: StopTime
type(ESMFTimeInterval):: DT
integer :: RC
rc = ESMF_SUCCESS
!

! Initialize ESMF
!

    call ESMF_Initialize (rc=rc)
    if(RC==ESMF_FAILURE) call exit(RC)
```
! Initial and final time of run and time step
!--------------------------------------------

call ESMF_TimeSet( StartTime, YY = 2007, rc=RC )
if(RC==ESMF_FAILURE) call exit(RC)

call ESMF_TimeSet( StopTime, YY = 2008, rc=RC )
if(RC==ESMF_FAILURE) call exit(RC)

call ESMF_TimeIntervalSet( DT, S=1800, rc=RC )
if(RC==ESMF_FAILURE) call exit(RC)

! Create the Clock
!-----------------

clock = ESMF_ClockCreate( "MyClock",timeStep=DT, StartTime=StartTime,
StopTime=StopTime, rc=STATUS )
if(RC==ESMF_FAILURE) call exit(RC)

! Create the gridded component
!-----------------------------

GC = ESMF_GridCompCreate(name='ExampleGC',rc=rc)
if(RC==ESMF_FAILURE) call exit(RC)

! SetServices
!------------

call ESMF_GridCompSetServices ( GC, SetServices, RC)
if(RC==ESMF_FAILURE) call exit(RC)

! Initialize
!-----------

call ESMF_GridCompInitialize ( GC, Import, Export, Clock, RC)
if(RC==ESMF_FAILURE) call exit(RC)

do while (.not. ESMF_ClockIsDone(Clock))

! Run
!----

call ESMF_GridCompRun ( GC, Import, Export, Clock, RC)
if(RC==ESMF_FAILURE) call exit(RC)

! Tick the Clock
!---------------

call ESMF_ClockAdvance(Clock, rc=RC)
if(RC==ESMF_FAILURE) call exit(RC)
enddo

! Finalize
!--------

call ESMF_GridCompFinalize ( GC, Import, Export, Clock, RC)
if(RC==ESMF_FAILURE) call exit(RC)

! Finalize ESMF


3.3 **Example 2: HelloWorldMod**

The second example illustrates a more typical use of MAPL to help write a gridded component.

```fortran
module HelloWorldMod
  ! We always have this preamble
  !----------------------------
  use ESMF
  use MAPL

  implicit none

  ! Make sure only SetServices id public.
  ! This is a hallmark of ESMF gridded components.
  !-------------------------------

  private
  public SetServices

  contains

  ! We write a simple SetServices to register our custom
  ! run method with MAPL. We rely on MAPL_GenericSetServices
  ! to do the heavu work. Normally, we would also register
  ! data and connectivities at this point, but in this example
  ! we have none.
  !------------------

  subroutine SetServices(GC,RC)
    type(ESMF_GridComp), intent(INOUT) :: GC
    integer, optional, intent( OUT) :: RC

    call MAPL_GridCompSetEntryPoint ( GC, ESMF_SETRUN, Run, &
                                        ESMF_SINGLEPHASE, RC )
    call MAPL_GenericSetServices ( GC, RC )
  end subroutine SetServices
```

--------

call ESMF_Finalize (rc=rc)
if(RC==ESMF_FAILURE) call exit(RC)
! All Done
!---------
call exit(RC)
end Program Example1
! The Run method
!-----------------

subroutine Run (GC, IMPORT, EXPORT, CLOCK, RC )
type(ESMF_GridComp), intent(INOUT) :: GC
    type(ESMF_State), intent(INOUT) :: IMPORT
    type(ESMF_State), intent(INOUT) :: EXPORT
    type(ESMF_Clock), intent(INOUT) :: CLOCK
    integer, optional, intent( OUT) :: RC
    type(ESMF_Config) :: CF
    character(len=ESMF_MAXSTR) :: COMP_NAME
    real :: DT

call ESMF_GridCompGet ( GC, NAME=COMP_NAME, config=CF)
call ESMF_ConfigGetAttribute( CF, DT, Label="DT:")
print *, 'Hello World. I am ', trim(COMP_NAME), 
       ', and my timestep is ',DT

end subroutine Run

end module HelloWorldMod

This example needs a custom a Run method. Since this method can only be registered in a
SetServices that is in the module, we must also write an explicit SetServices. Notice that
the registration of the Run method is with MAPL, not directly with ESMF. The component
does not explicitly register Initialize and Finalize methods, so the generic ones will be used.
Notice also that MAPL_GenericSetServices is called at the end, after all registration with
MAPL is completed.

The Run method is simple, but it does illustrate that every instance of an ESMF_GriddedComponent
has a name, and the IRF methods can access it to know which instance they are working on.
Notice also that we have assumed that there is an open configuration in the gridded
component, from which we are getting the time step. This is also typical of MAPL components.
MAPL treats the configuration in the component object like an environment from which it
can always query for predefined metadata.

The situation illustrated by this example is quite common. Most simple components will
follow this template: defining a Run method, having a SetServices that registers it and
calls MAPL_GenericSetServices, and defaulting the Initialize and Finalize methods.

Additional Rules for Grids and States

Most MAPL_GriddedComponent gridded components will receive a fully populated grid from
its parent. Some, however, may need be written to receive an empty grid that they populate
themselves or to replace the grid they receive with one of their own creation.

Its current implementation, MAPL severely restricts the nature of ESMF_Grids reflecting
in part the state of ESMF’s own development. We will discuss this at length later.
The following are some of the grid related rules:

**Rule 10** A component’s grid must be fully formed when `MAPL_GenericInitialize` is invoked.

**Rule 11** Once `MAPL_GenericInitialize` is invoked, the grid may not be changed and must remain as the instance’s `ESMF_Grid`.

**Rule 12** An instance’s grid can be either an `ESMF_Grid` or a `MAPL_LocationStream` that has an associated `ESMF_Grid`. Thus there is always an `ESMF_Grid` associated with each instance of a `MAPL`-compliant `ESMF_GriddedComponent`.

A component can operate on data on various grids. These can be `ESMF_Grids` or grids defined with the user’s own conventions and ESMF Infrastructure can be used to manipulate this data internally. But to the outside world and to `MAPL` a `MAPL`-compliant component should “look” as though it has only one grid.

The following are the state related rules:

**Rule 13** Items in the IM/EX/IN states must be either `ESMF_States`, `ESMF_Bundles`, or `ESMF_Fields`.

**Rule 14** A `MAPL` places items in the IM/EX/IN states only through appropriate “spec” calls from its `SetServices`.

**Rule 15** All items the component places in the IM/EX/IN states must be defined on its grid. If the grid is a `MAPL_LocationStream`, these items can be either at locations or on the associated `ESMF_Grid`. Only in this sense can a component appear to expose two grids.

**Rule 16** In addition to the ESMF internal state that `MAPL` places in the component, a component can have any number of privately defined “internal” states. We will refer to these as the component’s private states.

**Rule 17** The private states, together with IN, fully define the component’s instanciatable state. Private states must, therefore, be attached to the `ESMF_GriddedComponent`.

**Rule 18** Private states must be “named” states when attached to the `ESMF_GriddedComponent`. `MAPL` used the unnamed internal state in the component for its own purposes.

**Rule 19** Items in the IM/EX/IN states may have `MAPL` and user attributes.

**Rule 20** Items in the Internal state can be given a `FRIENDLY_TO MAPL` attribute that consists of a list of other component’s names. `MAPL` then places these items in the component’s Export state, and it is an error to add another item with the same name to the Export.

**Rule 21** Items in the IM state are “read-only” to the component, unless the component’s name appears in the item’s `FRIENDLY_TO` attribute.

**Rule 22** Items in the EX state can be assumed to be “read-only” to other components, unless a non-empty `FRIENDLY_TO` is present.
Rule 23 Components can only create or modify the FRIENDLY_TO attribute of items in its Import state.

Rule 24 Values of all MAPL attributes can be set only in SetServices.

Note that the restriction on items being on the component’s grid applies only to the items explicitly placed in the states by the component; MAPL itself may place other items in these states that are not “visible” to the component. It is in this sense that the component “looks” as though it has a single grid, even when its children use different grids.

The recipe for writing a MAPL:

Writing an ESMF.GriddedComponent consists of writing a SetServices and at least one phase of each of the registered IRF methods. MAPL provides a recipe for each of these tasks. We will focus first on the writing of a leaf component and defer the discussion of how to extend the recipe to composite components and to putting together hierarchies to the MAPL.Connect section.

3.4 Writing a SetServices

Every non-trivial MAPL has a SetServices from which MAPL.GenericSetServices is called, as illustrated in Example 2. In this section we provide a complete recipe for writing SetServices and explain exactly what MAPL.GenericSetServices does.

The minimum we must do in SetServices is registering any private IRF methods we are writing (Run in Example 2) and then calling MAPL.GenericSetServices. Everything else is optional. The following is a complete list in the order in which they would normally appear:

1. Get instance name and set-up traceback handle (MAPL_Util)

   *This is only for using the optional error handling.*

2. If it is using a private internal state, allocate it and put it in the gridded component with a unique name.

3. Register any custom IRF methods with MAPL. MAPL will register them with ESMF.

   *This step is present in practically all components.*

4. Set Data Services for the GC

   *Data services are the heart of MAPL and practically all components will have to do some state description. An exception would be a composite component that serves only as a container for its children. We will explain the setting of data services in detail on the following sections.*

6. Set the Profiling timers (MAPL_Util)

This, of course, is optional.

3.4.1 What MAPL_GenericSetServices Does

As we showed in Example 1, MAPL_GenericSetServices, can be used as a component’s SetServices, but this is not very useful. Its usual use is as a set-up routine for MAPL called as one of the last things from the component’s own SetServices (step 5 above).

MAPL_GenericSetServices performs the following tasks:

- If the MAPL object does not exist in the component, it allocates it and places it in the component. Usually the object already exists at this point.
- Sets the any of the ESMF_GriddedComponent’s IRF methods that have not been registered to the generic versions.
- Deals with the children. This is discussed in MAPL_Connect.

Most of this is straightforward, but a few points merit further discussion, especially the setting of data services and the treatment of the children.

3.4.2 Data Services

A crucial aspect of writing a MAPL component is describing the three states (Import/Export/Internal – IM/EX/IN). These are all ESMF_States. The IM/EX states are those passed in the calls to the IRF methods. The IN state is attached to the MAPL object by MAPL. In SetServices we must describe all items in all three states. This will allow MAPL to create, initialize, and otherwise manipulate these data.

MAPL assumes that items in these states are either ESMF.Fields or ESMF.Bundles. Each item is described by a call to MAPL_AddXXXSpec, where 'XXX' stands for either Import, Export or Internal. These calls do not modify these states or create the items; they merely update tables of item specifications for the three states. The interface is as follows:

```plaintext
subroutine MAPL_AddInternalSpec(MAPL , &
  SHORT_NAME, &
  LONG_NAME, &
  UNITS, &
  DIMS, &
  VLOCATION, &
  DATATYPE, &

```
REFRESH_INTERVAL,  &
AVERAGING_INTERVAL, &
DEFAULT,  &
HALOWIDTH,  &
PRECISION,  &
FRIENDLYTO,  &

RC  )

Only the MAPL object MAPL and the SHORT_NAME are required. The latter is the handle used to access the variable; it is also the name used for the variable by MAPL in checkpoint files. The remaining, optional arguments are as follows:

LONG_NAME  A longer descriptive name; we try to use the CF standard name. Default: (empty string)

UNITS  Default: (empty string)
DIMS
This describes the structure of the arrays. It can take on any of the following integer values:

MAPL_DimsUnknown

MAPL_DimsHorzOnly 1- or 2-D arrays in the horizontal

MAPL_DimsVertOnly 1-D arrays in the vertical

MAPL_DimsHorzVert 2- or 3-D arrays with one or two horizontal dimensions and a vertical dimension. Because of temporary ESMF constraints, the vertical dimension is assumed to be the last. This constraint will be removed later.

MAPL_DimsTileOnly 1-D array of “tile” or location stream variables.

MAPL_DimsTileTile 2-D array of tile by subtile variables.

Default: none

Even though DMD is optional, it makes little sense not to specify it.

DATATYPE
This is the type of item. It can be any of the following integer values:

MAPL_FieldItem

MAPL_BundleItem

Default: MAPL_FieldItem

VLOCATION
Location of the variable in a vertically staggered grid. Edge variables will be dimensioned starting at 0. It can take on the following integer values:

MAPL_VLocationNone

MAPL_VLocationEdge

MAPL_VLocationCenter

Default: MAPL_VLocationNone

REFRESH_INTERVAL
For IMPORT variables only. Data will be valid only on this interval, given in integer seconds. Currently the time origin is the time when this call is made; in future this should be defined in the argument list. Default: 0
AVERAGING_INTERVAL  For IMPORT variables only. Data will be an average over this interval immediately preceding the valid time. This is also in integer seconds. If zero, the value will be instantaneous. What is meant by “preceeding” is a bit tricky, since it may or may not include the update for the current timestep, depending on whether the source component(s) were already called or not. This is discussed further in the checkpointing and coupling sections. Default: 0

DEFAULT  Value to which variable is set when allocated. Default: 0.0

HALOWIDTH  The width of the halo region. Currently it is the same on all dimensions. Note that “haloing” is currently defined only for variables with two horizontal dimensions. Default: '0'

FRIENDLYTO  For INTERNAL variables only. A colon separated list of component types that the variable is “Friendly” to. Default: (blank string)

3.5 Writing an Initialize Method

Every MAPL component must make a call to MAPL_GenericInitialize. This can be done by letting the method default or by writing a component-specific Initialize method that invoked MAPL_GenericInitialize. In this section we provide a complete recipe for writing an Initialize and explain exactly what MAPL_GenericInitialize does.

The main reason to write a component-specific Initialize is to handle a private internal state. If all internal state variables can be put in the MAPL INTERNAL and checkpointed, using MAPL_GenericInitialize should suffice, at least for a simple component. A composite component may have other considerations; these will be discussed in later sections.

1. Get the name from the ESMF_GriddedComponent and set-up traceback handle (MAPL_Utils)

   *This is only for using the optional error handling.*

2. Get the MAPL object from the ESMF_GriddedComponent

   *It will almost certainly be convenient to query this object during Initialization.*

3. If you are doing profiling turn on timer (MAPL_Utils)

4. If you will use the configuration, get it from the ESMF_GriddedComponent

   *The configuration is to a component what the environment is to a UNIX process. We use it to keep all parameters, and so it is likely to be needed in Initialize.*

5. Get the component’s private internal state from the ESMF_GriddedComponent
If you are writing your own Initialize you will almost certainly be using a private internal state.

6. If you are changing the grid, it has to be done before invoking MAPL_GenericInitialize.

   Remember, by default the component’s natural grid will be the one it was given at creation. If an INTERNAL and/or an IMPORT state is being restarted (as described in the next section), the grids on those restarts will override whatever is present when MAPL_GenericInitialize is called in the next step. So it only makes sense to change the grid if you are not doing restarts in MAPL_GenericInitialize. After returning from MAPL_GenericInitialize, the natural grid cannot be changed.

7. Invoke MAPL_GenericInitialize

   This will do the automatic state initializations as described below. In the case of a composite component, it will also initialize the children.

8. If you have put items that need to be explicitly initialized in the MAPL INTERNAL state, get it from the MAPL object

   Items in the MAPL INTERNAL state that were checkpointed will be restored by MAPL_GenericInitialize; other items will be set to their DEFAULT value. We need access to INTERNAL only if we wish to override these in Initialize. An example of this would be setting static arrays, like map factors, Coriolis, etc.

9. Query the MAPL object for information you need to do initialization

    You probably need to know what the world looks like, so get LATS and LONS.

10. Query the configuration for parameters you need to do initialization

11. Get pointers from the MAPL INTERNAL and/or the private internal states.

    These are the quantities you need to initialize.

12. Do the Initialization

    For INTERNAL items, you are overriding MAPL’s initialization, which was either from a restart or a default; for a private state you are on your own.

13. If you are profiling, turn off timer

3.6 What MAPL_GenericInitialize Does

MAPL_GenericInitialize does most of the instance-specific initializations of the MAPL objects. It also creates, an possibly allocates and initializes, items in the IM/EX/IN states. MAPL_GenericInitialize also makes the final decision on what will be the natural grid. And, as is the case for all generic IRF methods, it calls the children’s Initialize. The following list discusses these tasks in more detail:
3.6.1 Writing a Finalize Method

Finalize parallels the Initialize. It is usually only needed if there is a private internal state.

3.6.2 What MAPL_GenericFinalize Does

MAPL_GenericFinalize does most of the instance-specific finalizations of the MAPL objects. It checkpoints the Import and Export states if a checkpoint file has been provided. It also destroys, an possibly deallocates items in the IM/IN states. MAPL_GenericInitialize also makes the It calls the children’s Finalize.

4 Building complex applications: MAPL_Connect

MAPL adopts ESMF’s natural hierarchical topology for component connectivity, following the model illustrated in Figure 3. In a typical application, the leaf components at the bottom of the figure contain the bulk of the computational code. These are things like physical parameterizations or dynamical cores, and they are grouped in composite components (their parents), where the physical coupling is performed. Each parent’s constituent components (its children) can be connected to each other by ESMF couplers. It is in these couplers that the more automatable coupling functions, such as grid transformation, accumulation, etc., are performed.

Note that in this hierarchical scheme all coupling, whether physical or automatable, occurs between siblings. This simplifies the placement of couplers, which is important to us because we want this to be done automatically by MAPL, but it does require some means of making connections between “cousins.” This is done by adopting some rules that define the parent-child relationship. Since a parent component’s “owns” its children components and their IM/EX states (it declares them), it has access to them. In MAPL, we take advantage of this by having the parent explicitly declare what connections it wants between its children’s import and export states. Once again, this is done in SetServices. The following call, which is made by the parent of ESMF_GridedComponents called SURFACE and MOIST, would let MAPL know that it needs certain connectivity services between these children; MAPL will provide these by automatically generating the appropriate couplers, extracting some of the needed information from the data services provided by the children.

```fortran
call MAPL_AddConnectivity ( GC, &
   SRC_NAME = ('PCU', 'PLS', 'SNO'), &
   DST_NAME = ('CU_prec', 'LS_prec', 'SnowFall'),&
   SRC_ID = 'SURFACE', &
   DST_ID = 'MOIST', &
   RC = STATUS )
```

After all connections between the children are processed, their import states may still con-
tain some unsatisfied items (such as those that would be provided by cousins). MAPL adds these to the parent’s Import state. This occurs recursively up the hierarchy until, in a well-coupled application, all imports are satisfied. In order to have the cousin’s export available to the parents, MAPL places all of the children’s exports in the parent’s Export state. This also continues recursively up the hierarchy.

4.1 What MAPL_GenericSetServices Does with the Children

- Allocates an ESMF_GriddedComponent and an IMPORT and EXPORT state for each child
- Creates each child’s ESMF_GriddedComponent using the inherited grid and configuration. The Ith child is named GCNames(I).
- Creates each child’s Import and Export states. These are named GCNames(I)="/IMPORT" and GCNames(I)="/EXPORT"
- Invokes each child’s set services. These are chosen from the five possible externals specified, depending on the value of SSptr(I). By convention, if SSptr is not present, there can be at most as many children as optional externals, and these are associated in the order they appear in GCNames and the argument list.
- “Wires” the children. This resolve all child imports that are satisfied by siblings. All such connections must have been added explicitly in SetServices (step 4 above).
- Propagates each child’s export state to the component’s export state.
- Propagated the children’s unresolved imports to the component’s import state.

4.2 Rules for the Application

Rule 25  Every MAPL application will have one and only one Root component, which will be an ancestor of every component except the History component.

Rule 26  The Cap component is the main program; it has no parent and exactly two children: Root and History. The application component creates and initializes the configuration.

4.3 The Configuration

MAPL requires that the application’s configuration be propagated down from parents to children, and that it be present in the component as soon as the component is created. It effectively treats the configuration as though it was a UNIX environment available to all components in an application.

The configuration may be obtained from the ESMF_GriddedComponent and queried using the standard ESMF interface, as shown in the run method of Example 2. It can also be queried
through the \textit{MAPL} object by calling \texttt{MAPL\_GetResource}, and this is the preferred way of doing it. When the configuration is queried this way, \textit{MAPL} first tries to match a label that has been made instance-specific by prepending the instance’s full name and an underscore to the specified label; in Example 2, \textit{MAPL} would first look for \texttt{trim(COMP\_NAME)}/${'_DT: '}$. If this is not found, it would then look for a type-specific label by prepending only the last name, if the instance has one. If this fails, it would look for the unqualified label, \texttt{DT:}; finally, if this also failed, it would set it to the default value, which in the example is the application’s time step, \texttt{RUN\_DT}.

5 Doing Diagnostics: \textit{MAPL\_History}

\texttt{MAPL\_HistoryGridCompMod} is an internal \textit{MAPL} gridded component used to manage output streams from a \textit{MAPL} hierarchy. It writes Fields in the Export states of all \textit{MAPL} components in a hierarchy to file collections during the course of a run. It also has some limited capability to interpolate the fields horizontally and/or vertically before outputting them.

It is usually one of the two gridded components in the “cap” or main program of a \textit{MAPL} application, the other being the root of the \textit{MAPL} hierarchy it is servicing. It is instanciated and all its registered methods are run automatically by \texttt{MAPL\_Cap}, if that is used. If writing a custom cap, \texttt{MAPL\_HistoryGridCompMod}’s \texttt{SetServices} can be called anytime after ESMF is initialized. Its \texttt{Initialize} method should be executed before entering the time loop, and its \texttt{Run} method at the bottom of each time loop, after advancing the \texttt{Clock}. \texttt{Finalize} simply cleans-up memory.

The component has no true export state, since its products are diagnostic file collections. It does have both Import and Internal states, which can be treated as in any other \textit{MAPL} component, but it generally makes no sense to checkpoint and restart these.

The behavior of \texttt{MAPL\_HistoryGridCompMod} is controlled through its configuration, which as in any \textit{MAPL} gridded component, is open and available in the GC. It is placed there by the cap and usually contained in a \texttt{HISTORY.rc} file.

\texttt{MAPL\_HistoryGridCompMod} uses \texttt{MAPL\_CFIO} for creating and writing its files; it thus obeys all \texttt{MAPL\_CFIO} rules. In particular, an application can write either Grads style flat files together with the Grads .ctl file description files, or one of two self-describing format (netcdf or HDF), which ever is linked with the application.

Each collection to be produced is described in the \texttt{HISTORY.rc} file and can have the following properties:

- Its fields may be ”instantaneous” or ”time-averaged”, but all fields within a collection use the same time discretization.
- A beginning and an end time may be specified for each collection.
- Collections are a set of files with a common name template.
• Files in a collection have a fixed number of time groups in them.

• Data in each time group are "time-stamped"; for time-averaged data, the center of the averaging period is used.

• Files in a collection can have time-templated names. The template values correspond to the times on the first group in the file.

The body of the HISTORY.rc file usually begins with two character string attributes under the config labels EXPID: and EXPDSC: that are identifiers for the full set of collections. These are followed by a list of collection names under the config label COLLECTIONS:. Note the conventional use of colons to terminate labels in the HISTORY.rc.

The remainder of the file contains the attributes for each collection. Attribute labels consist of the attribute name with the collection name prepended; the two are separated by a ".".

Attributes are listed below. A special attribute is collection.fields: which is the label for the list of fields that will be in the collection. Each item (line) in the field list consists of a comma separated list with the field’s name (as it appears in the corresponding ESMF field in the EXPORT of the component), the name of the component that produces it, and the alias to use for it in the file. The alias may be omitted, in which case it defaults to the true name.

Files in a collection are named using the collection name, the template attribute described below, and the EXID: attribute value. A filename extension may also be added to identify the type of file (e.g., .hdf).

[expid.]collection[.template][.ext]

The appended extension depends on the mode attribute below. If mode is "HDF", the extension is always .hdf, even when using a netcdf library. If it is "GrADS", the data files have no extension and the “control file” has the .ctl extension, but with no template. The expid is always prepended, unless it is an empty string.

The following are the valid collection attributes:

• template Character string defining the time stamping template that is appended to collection to create a particular file name. The template uses GrADS convensions. The default value depends on the duration of the file.

• descr Character string describing the collection. Defaults to "expdsc".

• format Character string to select file format ("GrADS" or "HDF"). "HDF" implies whatever IO library is linked, netcdf or HDF. Default = "GrADS".

• frequency Integer (HHHHMMSS) for the frequency of time groups in the collection. Default = 060000.
mode  Character string equal to "instantaneous" or "time-averaged". Default = "instantaneous".

acc_interval  Integer (HHHHMMSS) for the acculation interval (= frequency) for time-averaged diagnostics. Default = frequency; ignored if mode is "instantaneous".

ref_date  Integer (YYYYMMDD) reference date for frequency; also the beginning date for the collection. Default is the Start date on the Clock.

ref_time  Integer (HHMMSS) Same a ref_date.

end_date  Integer (YYYYMMDD) ending date to stop diagnostic output. Default: no end

duration  Integer (HHHHMMSS) for the duration of each file. Default = 00000000 (everything in one file).

resolution  Optional resolution (IM JM) for the output stream. Transforms between two regulate LogRect grid in index space. Default is the native resolution.

xyoffset  Optional Flag for output grid offset when interpolating. Must be between 0 and 3. (Cryptic Meaning: 0:DcPc, 1:DePc, 2:DcPe, 3:DePe). Ignored when resolution results in no interpolation (native). Default: 0 (DatelinCenterPoleCenter).

levels  Optional list of output levels (Default is all levels on Native Grid). If vvars is not specified, these are layer indeces. Otherwise see vvars, vunits, vscale.

vvars  Optional field to use as the vertical coordinate and functional form of vertical interpolation. A second argument specifies the component the field comes from. Example 1: the entry 'log(PLE)', 'DYN' uses PLE from the DYN component as the vertical coordinate and interpolates to levels linearly in its log. Example 2: 'THETA', 'DYN' a way of producing isentropic output. Only log(.), pow(.real number) and straight linear interpolation are supported.

vunit  Character string to use for units attribute of the vertical coordinate in file. The default is the MAPL_CFIO default. This affects only the name in the file. It does not do the conversion. See vscale

vscale  Optional Scaling to convert VVARS units to VUNIT units. Default: no conversion.

regrid_exch  Name of the exchange grid that can be used for interpolation between two LogRect grids or from a tile grid to a LogRect grid. Default: no exchange grid interpolation. irregular grid.

regrid_name  Name of the Log-Rect grid to interpolate to when going from a tile to Field to a gridde output. regrid_exch must be set, otherwise it is ignored.
The following is a sample HISORY.rc take from the FV_HeldSuarez test.

EXPID: fvhs_example
EXPSC: fvhs_(ESMF07_EXAMPLE)_5x4_Deg

COLLECTIONS:
'dynamics_vars_eta'
'dynamics_vars_p'
::

dynamics_vars_eta.template: '%y4%m2%d2_%h2%n2z',
dynamics_vars_eta.format: 'HDF',
dynamics_vars_eta.frequency: 240000,
dynamics_vars_eta.duration: 240000,
::

dynamics_vars_p.template: '%y4%m2%d2_%h2%n2z',
dynamics_vars_p.format: 'Grads',
dynamics_vars_p.frequency: 240000,
dynamics_vars_p.duration: 240000,
dynamics_vars_p.vscale: 100.0,
dynamics_vars_p.vunit: 'hPa',
dynamics_vars_p.vvars: 'log(PLE)' , 'FVDYNAMICS' ,
dynamics_vars_p.levels: 1000 900 850 750 500 300 250 150 100 70 50 30 20 10 7 5 2 1 0,
::

6 Doing I/O: MAPL_CFIO

MAPL_CFIO interfaces CFIO to the ESMF data types. It currently includes read-write support for ESMF Fields and States, and read support for ESMF Fields and Fortran arrays.
It has only four methods: MAPL_CFIODread, MAPL_CFIOWrite, MAPL_CFIORead, MAPL_CFIODestroy. Except for MAPL_Read, all work on the MAPL_CFIIO object. Reading is done directly from a file to the appropriate ESMF object.

MAPL_CFIIO is designed for two modes of I/O: self-describing formats (SDF), of which it supports HDF-4 and netcdf-3, and flat files, which includes support for GrADS readable files. The GrADS support is still under construction. There are also plans to add GRIB support.

In SDF mode, capability of MAPL_CFIIO depends on which library (HDF or netcdf) is linked with the application, since both cannot be used because name conflicts between these two libraries. If netcdf is linked, only netcdf files may be read or written. If HDF is linked, both HDF and netcdf files may be read, but only HDF files may be written.

7 Miscellaneous Features: MAPL_Utils

Many aspects of the ESMF infrastructure, such as those dealing with time management, error logging, etc., can easily be used directly by modelers. Elements of the infrastructure that involve interfaces to ESMF’s communications layer, which are intended to be among ESMF most powerful methods, are not as easy to adopt. The major hurdle to using these elements of the ESMF infrastructure is that the user pretty much has to put his data into ESMF_Fields, which are the main objects on which the ESMF communication methods work. MAPL facilitates this by creating all elements described in data services as ESMF_Fields or ESMF_Bundles within the three states. In user code these can be extracted directly and manipulated as ESMF_Fields when using ESMF infrastructure, or one extract fortran pointers to the data when interfacing to existing user code.

MAPL provides several features that are not central to its main goals, but which can be very handy. Some of these provide functionality in and instance-specific way by saving metadata in the MAPL object. This save the user the need to deal with such things in his private internal state. The main support is for profiling, error handling, and astronomy. These are very simple and we expect that eventually they will be superceded by ESMF utilities, or remain as simple interfaces to them.

7.1 Error Handling

The error handling utility consists of the three macros:

```c
VERIFY_(STATUS)
RETURN_(ESMF_Success|ESMF_Failure)
ASSERT_(logical expr)
```

These are used by setting the local character string variable Iam to the subroutine name, where possible qualified by the instance’s name, and then using VERIFY_ to test ESMF and
MAPL return codes, RETURN, to exit routines, and ASSERT, for conditional aborts.

7.2 Profiling

The API of the profiling utility consists of three subroutines:

\begin{verbatim}
MAPL_TimerAdd(MAPL, NAME, RC)
MAPL_TimerOn (MAPL, NAME, RC)
MAPL_TimerOff(MAPL, NAME, RC)
\end{verbatim}

where MAPL is the MAPL object and NAME is the string name of a performance meter. Meters are usually registered in SetServices with Add and can then be turned on and off throughout the user code. In generic finalize the results are reported to standard out. Even if the user registers no meters, the performance of the generic IRF methods is reported.

7.3 Astronomy

The astronomy is also simple and easy to use. At any time after the MAPL object is created (i.e., after the call to MAPL_GenericSetServices) it can be queried for an opaque object of type MAPL_SunOrbit. This orbit object can then be used to get the insolation at the top of the atmosphere through the following API:

\begin{verbatim}
MAPL_SunGetInsolation(LONS, LATS, ORBIT, ZTH, SLR, INTV, CLOCK, TIME, RC)
\end{verbatim}

where LONS and LATS can be either one- or two-dimensional Fortran arrays or general ESMF arrays with one or two horizontal dimensions, ORBIT is the predefined object of type MAPL_SunOrbit, and ZTH and SLR are the cosine of the solar zenith angle and the TOA insolation at the given latitudes and longitudes; these are, of course, declared in the same way as LONS and LATS. The remaining arguments are optional and their use is explained in Part II.

By default the orbit created by MAPL uses late 20\textsuperscript{th} century orbital parameters. These can be overridden in the configuration by specifying ECCENTRICITY:, OBLIQUITY, PERIHELION:, AND EQUINOX:. The meaning of these, as well as more complex uses of the astronomy are also explained in the prologues of MAPL_SunMod in Part II.

7.4 Universal Constants

The following universal constants are defined when MAPLMod is used:

\begin{verbatim}
MAPL_PI 3.14159265358979323846
MAPL_GRAV 9.80 m s\textsuperscript{-2}
\end{verbatim}
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAPL_RADIUS</td>
<td>6376.0E3 m</td>
</tr>
<tr>
<td>MAPL_OMEGA</td>
<td>2.0*MAPL_PI/86164.0</td>
</tr>
<tr>
<td>MAPL_ALHL</td>
<td>2.4548E6 J kg⁻¹</td>
</tr>
<tr>
<td>MAPL_ALHS</td>
<td>2.8368E6 J kg⁻¹</td>
</tr>
<tr>
<td>MAPL_ALHF</td>
<td>MAPL_ALHS−MAPL_ALHL</td>
</tr>
<tr>
<td>MAPL_STFBOL</td>
<td>5.6734E−8 W m⁻² K⁻⁴</td>
</tr>
<tr>
<td>MAPL_AIRMW</td>
<td>28.97 kg Kmole⁻¹</td>
</tr>
<tr>
<td>MAPL_H2OMW</td>
<td>18.01 kg Kmole⁻¹</td>
</tr>
<tr>
<td>MAPL_RUNIV</td>
<td>8314.3 J Kmole⁻¹ K⁻¹</td>
</tr>
<tr>
<td>MAPL_KAPPA</td>
<td>2.0/7.0 J Kg⁻¹</td>
</tr>
<tr>
<td>MAPL_RVAP</td>
<td>MAPL_RUNIV/MAPL_H2OMW J K⁻¹ kg⁻¹</td>
</tr>
<tr>
<td>MAPL_RGAS</td>
<td>MAPL_RUNIV/MAPL_AIRMW J K⁻¹ kg⁻¹</td>
</tr>
<tr>
<td>MAPL_CP</td>
<td>MAPL_RGAS/MAPL_KAPPA J K⁻¹ kg⁻¹</td>
</tr>
<tr>
<td>MAPL_P00</td>
<td>100000.0 Pa</td>
</tr>
<tr>
<td>MAPL_CAPWTR</td>
<td>4218. J Kg⁻¹ kg⁻¹</td>
</tr>
<tr>
<td>MAPL_RHOWTR</td>
<td>1000. kg⁻³ m⁻³</td>
</tr>
<tr>
<td>MAPL_NUAIR</td>
<td>1.533E−5 m⁻² S⁻¹ (@ 18C) K</td>
</tr>
<tr>
<td>MAPL_TICE</td>
<td>273.16 K</td>
</tr>
<tr>
<td>MAPL_UNDEF</td>
<td>-999.0</td>
</tr>
<tr>
<td>MAPL_SRFPRS</td>
<td>98470 Pa</td>
</tr>
<tr>
<td>MAPL_KARMAN</td>
<td>0.40</td>
</tr>
<tr>
<td>MAPL_USMIN</td>
<td>1.00</td>
</tr>
<tr>
<td>MAPL_VIREPS</td>
<td>MAPL_AIRMW/MAPL_H2OMW−1.0 m s⁻¹</td>
</tr>
</tbody>
</table>
References

