MAPL Manual

Max Suarez, Atanas Trayanov, Arlindo da Silva, Purnendu Chakraborty

March 14, 2014

Contents

Chapter 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 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. Figure 1 represents a hierarchical (tree) implementation of the component-based GEOS-5 software where each box is an ESMF component performing some specific function and the root of the tree serves as the top level control point.

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 led to two seemingly contradictory conclusions: all implementations are different and 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

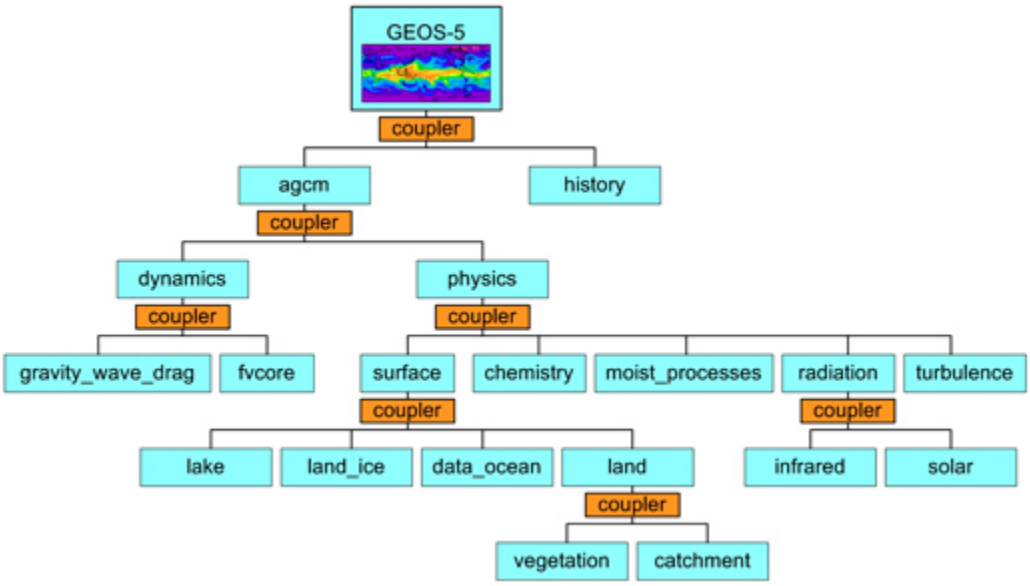


Figure 1.1: Structure of the GEOS-5 atmospheric general circulation model

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 middle-ware layer (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 middle-ware-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.

The documentation is arranged as follows: In Chapter 2, we review relevant aspects of ESMF, followed by a chapter that aims to introduce readers, already familiar with ESMF,

quickly into MAPL through examples that increase in complexity, demonstrating the salient features of MAPL. The following chapter provides are more detailed description of MAPL followed by the MAPL API (Application Programming Interface) and the source codes for tutorials in the appendix.

Chapter 2

ESMF - A review of aspects relevant to MAPL

Leaf and Composite components are defined here.

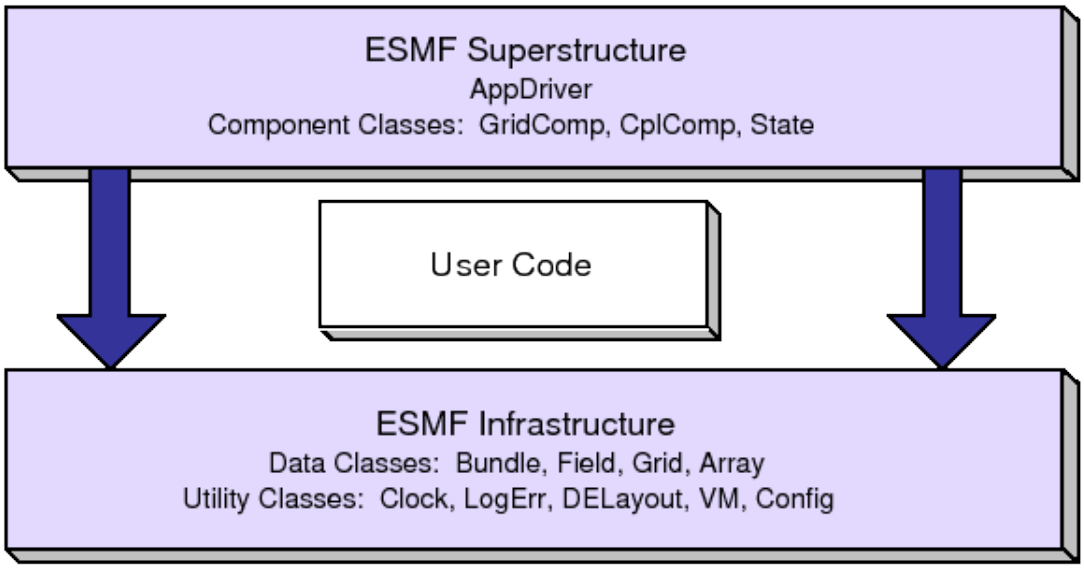
The Earth System Modeling Framework (ESMF) [1] is a software package designed to provide some of the essential functions needed by parallel, scalable earth system models in a machine-independent way. ESMF is implemented as a collection of very general programming classes that can be used both to construct ESMF components and to connect them to one another. These classes thus support modelers in building interoperable and portable codes. This design is illustrated by the ESMF 'sandwich' diagram (Figure 2), where the user's computational code sits between the two ESMF layers.

The simplest ESMF implementation consists 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 actually be done without using any 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_Field`s) 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.

An ESMF component (represented by a box in Figure 1, e.g. `solar`) consists of four (**or just one — SetServices!?!?!**) public component interface functions performing specific roles:

Figure 2.1: 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. Taken from [2]



.

`SetServices`: A component's `SetServices` function is called when an instance (object) of the component is created and is the *only required public interface* of the component. It takes the instantiated component as the first argument, and an integer return code as the second. The goal of `SetServices` is to register with the framework the component's user-defined routines that satisfy the `Initialize`, `Run` and `Finalize` requirements.

`Initialize`: A component's `Initialize` function is called to configure an instance (object) of the component (allocate space, initialize data etc.). In addition to the component's instance, the arguments to this function include two `ESMF_State`s (one `Import` and one `Export`) and an `ESMF_Clock`. The `ESMF_State` variables are used to pass data between components. The `Clock` is used to pass the simulation time counter to the component instance.

`Run`: A component's `Run` function is called to carry out a cycle of the iteration that makes up the kernel of a component's computational algorithm. This function contains the kernel of user code and is called repeatedly as part of the component instance's life cycle. It takes the same argument list as `Initialize`.

`Finalize`: A component's `Finalize` function is called to terminate the the component instance cleanly (release space, write results etc.). It takes the same argument list as `Initialize`.

Some important aspects of the ESMF API that are relevant to MAPL are:

- Data structures added to an `ESMF_State` can have *arbitrary meta tags* associated with them
- An `ESMF_State` can contain an `ESMF_State` variable allowing *recursive nesting* of `ESMF_State` variables.
- *Hierarchical organization of gridded components*: `ESMF_GridComp`s can be simple containers for user code (leaf components) or they can contain other gridded `ESMF_GridComp`s (composite 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 `Export`s of one component and `Import`s of another.

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 middle-ware layer implemented in MAPL includes the following design elements:

1. Aides in constructing a component's `IRF` methods
2. Provides 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. MAPL extends the `ESMF_State` concept to a component's `Internal` state, and help it manage its persistent data.
3. Facilitates the use of `ESMF_Field`s and thus of the ESMF Infrastructure layer
4. Facilitates 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 - 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.

Chapter 3

MAPL

Contents

3.1 Overview

The MAPL library can be divided into the following sub-systems:

Mapl_Core is a collection of routines and conventions used to build `ESMF_GridComp`s (or to wrap legacy codes as `ESMF_GridComp`s). In particular, it includes the means of describing a component's `Import` and `Export` states as well as the new `Internal` state.

Mapl_Connect is a collection of routines and conventions used for organizing MAPL-ESMF Gridded Components into a MAPL hierarchy.

Mapl_History is an ESMF Gridded Component that sits inside MAPL and *can* be instantiated to provide data writing services for a MAPL hierarchy.

Mapl_ExtData is an ESMF Gridded Component that sits inside MAPL and *can* be instantiated to provide data services to the IMPORT states of MAPL components in a hierarchy.

Mapl_Utils is a set of support utillities for commonly performed tasks in global climate models. MAPL itself uses some of these, but, like MAPL_History, MAPL components or applications need not use them.

Mapl_CFIO is an I/O layer for ESMF `Field`s, `Bundle`s and `State`s that uses CF (Climate and Forecast) compliant methods to read from (or write to) NetCDF, HDF or GrADS files. It relies internally on MAPL and must be built with it and can be used even if one is not using the rest of MAPL.

The distinction between MAPL_Core and MAPL_Connect, 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 contructed is a perfectly good ESMF component and, other than having to access the MAPL library to build and execute, is not special in any way. The code in an application instantiating it would not need to know it was built with MAPL machinery.

3.2 Building a Mapl Gridded Component: Mapl_Core

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

1. It makes it easier to write a component's `IRF` methods. In fact in some cases they need not be written at all.

2. It adds an `Internal` (`IN`) `ESMF_State` to the component, supplementing the `Import`(`IM`) and `Export`(`EX`) states required by ESMF.

3. It provides a means of describing the contents of the three states so that MAPL can help manage them.

4. It adopts ground rules for the behavior of a component and its treatment of the three states.

5. It defines a standard recipe for writing MAPL-based ESMF Gridded Components.

Each of these items is discussed in more detail in the following subsections. It will be helpful to refer to the complete MAPL example in section 3.10.

3.2.1 Writing the `IRF` method

After writing some gridded components, one realizes that, except 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 (**or two!?!?!**) ways of doing this.

a) The first way is to use the generic versions of `SetServices` and the `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.

b) A second way of using MAPL is to simply call the generic versions of the methods from the component-specific versions, allowing them to perform the boilerplate functions.

c) **Should this really be included!?!?!** 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 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 `ESMF_State` '`Internal`' that we will discuss below. We will refer to these three states as the `IM`/`EX`/`IN` states. Note that they are all ordinary `ESMF_State`s. From the description of the three states (3.2.3) provided in the data services, 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 connectivity of children components, creating the appropriate couplers, registering their services, and executing their `IRF` methods.

3.2.2 The new `Internal` (`IN`) State

ESMF requires that the control is passed back to it at the end of a component's `Run` method. In the spirit of having as unintrusive a design as possible, *ESMF says nothing about a component's internal state* which will probably be needed during subsequent executions of the component's `Run`. MAPL provides a mechanism to place parts of its true internal state in an `ESMF_State` called an `Internal` state, that is similar to the `IM/EX` states.

Since it is desirable that gridded components be as object-oriented as possible, the framework has to allow them to be fully instantiatable. This requires that whatever the component defines as its internal state be *attached* (in the object-oriented paralance) to the instance of the `ESMF_GridComp`. ESMF provides such a mechanism - effectively a hook on which a component can hang the current instance of its internal state.

Accordingly, this new `IN` 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 `ESMF_GridComp` and, in principle, is accessible only through MAPL and can be queried.

All of the mechanisms for registering and manipulating data that are already available in MAPL for the `IM/EX` States, are extended to the `IN` 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 `SetServices` are, of course, automatically allocated, checkpointed, and restarted by the MAPL Initialize and `Finalize` methods.

3.2.3 Description of State contents

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 (`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 is exchanged between the components. These states can contain only ESMF objects (primarily `ESMF_Field`s and other `ESMF_State`s), but ESMF says nothing about how they are to be used. MAPL assumes that `IM/EX` states consist only of `ESMF_Field`s and other `ESMF_State`s. It also adopts the convention that, by default, items in its `Export` state are not modified by other components and that a component cannot modify items in its `Import` state (this default behavior can be changed by adding a 'FRIENDLY_TO' attribute to an `IM` 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:

Adding `Import` state:

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

Adding `Export` state:

```
call MAPL_AddExportSpec  (GC,                                &
                          SHORT_NAME = 'U',                  &
                          LONG_NAME  = 'eastward_wind',      &
                          UNITS      = 'm s-1',              &
                          DIMS       = MAPL_DimsHorzVert,    &
                          VLOCATION  = MAPL_VLocationCenter, &
                          RC         = STATUS)
```

Adding `Internal` state:

```
call MAPL_AddInternalSpec(GC,                                &
                          SHORT_NAME = 'PKZ',                &
                          LONG_NAME  = 'pressure_to_kappa',  &
                          UNITS      = 'Pa$^\kappa$',        &
                          PRECISION  = ESMF_KIND_R8,         &
                          DIMS       = MAPL_DimsHorzVert,    &
                          VLOCATION  = MAPL_VLocationCenter, &
                          RC         = STATUS)
```

Note that some of the attributes being set for the `ESMF_Field`s, such as units, 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 described below.

3.2.4 Rules for Components

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

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

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

Rule 3 Every ESMF_GridComp has a configuration (that stores parameters). A MAPL gridded component will expect it to be open (accessible!?!?!) 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_GridComp can be simple (called a *leaf*) or composite.

Rule 6 The component must obey all MAPL rules pertaining to its grid, as defined below (3.2.4.3).

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 gridded 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 codes show simple MAPL components.

3.2.4.1 Example 1: Using the Generic Component

MAPL has built-in ESMF_GridComps. The most fundamental 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_GridComp.

The following example is a main program that runs MAPL_Generic for a year. It also *illustrates the basic steps that an ESMF main program (called Cap) contains.* This is a fully-compliant ESMF_GridComp. It has a public SetServices taken from MAPL, and this is its only public object (method?). Of course, it does nothing; but it can be run as a null component anywhere an ESMF_GridComp 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 a natural grid - the ESMF_Grid is assumed to be given to it when the instance of the ESMF_GridComp is created. It has IM/EX/IN states, which are silently created by the implicit generic methods; but all three state are empty.

```
Program Example1

  use ESMF
  use MAPL_Mod, 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(ESMF_TimeInterval):: DT
  integer                :: RC

  RC = ESMF_SUCCESS

  ! Initialize ESMF
  !----------------
  call ESMF_Initialize(defaultCalendar=ESMF_CALKIND_GREGORIAN, 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( name='MyClock', timeStep=DT, &
           startTime=StartTime, stopTime=StopTime, userRC=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, importState=Import, exportState=Export, clock=Clock, RC)
  if(RC==ESMF_FAILURE) call exit(RC)

  ! Time loop
  do while (.not. ESMF_ClockIsDone(Clock))
    ! Run
    !----
    call ESMF_GridCompRun(GC, importState=Import, exportState=Export, clock=Clock, userRC=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 grid comp
  !-------------------
  call ESMF_GridCompFinalize(GC, importState=Import, exportState=Export, clock=Clock, userRC=RC)
  if(RC==ESMF_FAILURE) call exit(RC)

  ! Don't we need to destroy the components!?!?!
  !-----------------------------------------

  ! Finalize ESMF
  !--------------
  call ESMF_Finalize (rc=rc)
  if(RC==ESMF_FAILURE) call exit(RC)

  ! All Done
  !---------
  call exit(RC)

end Program Example1
```

3.2.4.2 Example 2: HelloWorldMod

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

```
module HelloWorldMod

  ! We always have this preamble
  !-----------------------------
```

```
  use ESMF
  use MAPL_Mod

  implicit none

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

contains

  ! a simple SetServices to register our custom
  ! run method (run_hello) with MAPL.
  !-----------------------------------------
  subroutine SetServices(gc,rc)
    ! input/output parameters
    !------------------------
    type(ESMF_GridComp), intent(INOUT) :: gc      ! gridded component
    integer, optional,    intent( OUT) :: rc      ! return code

    ! register custom run method
    call MAPL_GridCompSetEntryPoint(gc, ESMF_SETRUN, run_hello, rc)

    !IMPORTANT step - call GenericSetServices
    call MAPL_GenericSetServices(gc, rc)
  end subroutine SetServices

  ! The Run method
  !---------------
  subroutine run_hello(gc, import, export, clock, rc )

    ! input/output parameters
    !------------------------
    type(ESMF_GridComp),  intent(inout) :: gc     ! gridden component
    type(ESMF_State),     intent(inout) :: import ! import state
    type(ESMF_State),     intent(inout) :: export ! export state
    type(ESMF_Clock),     intent(inout) :: clock  ! the clock
    integer, optional,    intent(  out) :: rc     ! return code
                                                  ! 0 - all is well

    ! local
    !------
    type(ESMF_Config)                   :: cf     ! config
    character(len=ESMF_MAXSTR)          :: comp_name
    real                                :: dt     ! time step

    ! Get my name
    !------------
    call ESMF_GridCompGet(gc, name=comp_name, config=cf)

    ! Query configuration to get time step
    !-------------------------------------
    call ESMF_ConfigGetAttribute(cf, dt, label='run_dt:')
```

```
      print *, 'Hello World. I am ', trim(comp_name), &
               ', and my timestep is ',dt

      ! All done - successfully
      !-------------------------
      rc = ESMF_SUCCESS

    end subroutine run_hello

  end module HelloWorldMod
```

This example needs a custom `Run` method (`run_hello`). 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. Normally, we would also register data and connectivities at this point, but in this example, we have none. Also note that `MAPL_GenericSetServices` is called at the end, after all registration with MAPL is completed. We rely on `MAPL_GenericSetServices` to do the heavy work.

The `Run` method is simple, but it does illustrate that every instance of an `ESMF_GridComp` 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 (`ESMF_Config` - see section 3.3.3) in the gridded component, from which we are getting the time step. This is also typical of MAPL components and is crucial to the successful use of this and other examples is the. MAPL treats the configuration in the component object like an environment from which it can always query for predefined metadata. MAPL *requires* certain configuration variables to be set in order to properly execute any application.

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

3.2.4.3 Additional Rules for Grids and States

Most `MAPL_GridComp`s 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.

In its current implementation, MAPL severely restricts the nature of `ESMF_Grid`s reflecting in part the state of ESMF's own development. We will discuss this at length later (**where!?!?!**).

The following are some of the grid related rules:

Rule 10 A component's grid must be fully formed before `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 (attached) with each instance of a MAPL-compliant `ESMF_GridComp`.

A component can operate on data on various grids. These can be `ESMF_Grid`s or grids defined with the user's own conventions and ESMF Infrasructure 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 `ESMF_State` related rules:

Rule 13 Items in the `IM/EX/IN` states must be one of `ESMF_State`s, `ESMF_Bundle`s or `ESMF_Field`s.

Rule 14 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 instantiatable state. `Private` states must, therefore, be attached to the `ESMF_GridComp`.

Rule 18 `Private` states must be 'named' states when attached to the `ESMF_GridComp`. MAPL uses 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 `IN` 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 `EX` state, and it is an error to add another item with the same name to the Export. **Why don't we make this a regular EX state!?!?!**

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.

3.2.5 The recipe for writing a `MAPL_GridComp`

Writing an `ESMF_GridComp` 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 (an `ESMF_GridComp` that is a simple container for user code) and defer the discussion of how to extend the recipe to composite components and to putting together hierarchies to the MAPL_Connect section 3.3.

3.2.5.1 Writing a `SetServices`

Every non-trivial `MAPL_GridComp` has a `SetServices` from which `MAPL_GenericSetServices` is called, as illustrated in Example 2 (3.2.4.2). In this section we provide a complete recipe for writing `SetServices` and explain the role of `MAPL_GenericSetServices` (for variable declarations, please refer to the actual code).

The minimum we must do in `SetServices` is registering the private `IRF` methods (`Run` in Example 2) and then call `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_Utils: only used for *optional error handling*)

```
Iam = "SetServices"
call ESMF_GridCompGet( gc, name=COMP_NAME, RC=STATUS )
Iam = trim(COMP_NAME) // Iam
```

2. If using a `private` state, allocate it and put it in the gridded component with a unique name. **Why can't we just use an `IN` state!?!?!**

```
allocate( dyn_internal_state, stat=status )
wrap%dyn_state => dyn_internal_state
call ESMF_UserCompSetInternalState ( gc,'FVstate',wrap,status )
```

3. Register any custom `IRF` methods with MAPL . MAPL will register them with ESMF. *This step is present in practically all components.*

```
call MAPL_GridCompSetEntryPoint ( gc, ESMF_SETINIT,  Initialize, rc=status)
call MAPL_GridCompSetEntryPoint ( gc, ESMF_SETRUN,   Run1, rc=status)
call MAPL_GridCompSetEntryPoint ( gc, ESMF_SETRUN,   Run2, rc=status)
call MAPL_GridCompSetEntryPoint ( gc, ESMF_SETFINAL, Finalize, rc=status)
call MAPL_GridCompSetEntryPoint ( gc, ESMF_SETREADRESTART, Coldstart, rc=status)
```

4. Set Data Services for the gridded component. 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 explain the setting of data services in detail below (3.2.5.1.2).

```
call MAPL_AddImportSpec ( gc,                &
     SHORT_NAME = 'DUDT',                    &
     LONG_NAME  = 'eastward_wind_tendency', &
     UNITS      = 'm s-2',                   &
     DIMS       = MAPL_DimsHorzVert,         &
     VLOCATION  = MAPL_VLocationCenter,      &
     RC=STATUS )
```

   Similarly for `Export` and `Internal` variables.

5. Call `MAPL_GenericSetServices`. *This is required.* We discuss what it does next (3.2.5.1.1).

```
call MAPL_GenericSetServices( GC, RC=STATUS )
```

6. Set the Profiling timers (MAPL_Utils). *This, of course, is optional.*

```
call MAPL_TimerAdd( GC, name="INITIALIZE", RC=STATUS )
```

For an example of a `SetServices` routine, please refer to section 3.10.

3.2.5.1.1 What does MAPL_GenericSetServices do?

As we showed in Example 1 (3.2.4.1), MAPL_GenericSetServices can be used as a component's SetServices, but this is not particularly useful. Its *typical use* is as a set-up routine for MAPL and is one of the last things called 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 any of ESMF_GridComp's IRF methods that have not been registered to the generic versions.
- Deals with the children. This is discussed further in MAPL_Connect 3.3.

3.2.5.1.2 Data Services

A crucial aspect of writing a MAPL component is describing the three states (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 the three states. This will allow MAPL to create, initialze, 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 of MAPL_AddInternalSpec is as follows:

```
subroutine MAPL_AddInternalSpec(GC,                &
                                SHORT_NAME,        &
                                LONG_NAME,         &
                                UNITS,             &
                                DIMS,              &
                                VLOCATION,         &
                                DATATYPE,          &
                                NUM_SUBTITILES,    &
                                REFRESH_INTERVAL,  &
                                AVERAGING_INTERVAL, &
                                DEFAULT,           &
                                RESTART,           &
                                HALOWIDTH,         &
                                PRECISION,         &
                                FRIENDLYTO,        &
```

```
                                ADD2EXPORT,         &
                                ATTR_RNAMES,        &
                                ATTR_INAMES,        &
                                ATTR_RVALUES,       &
                                ATTR_IVALUES,       &
                                UNGRIDDED_DIMS,     &
                                RC)


type (ESMF_GridComp)              , intent(INOUT)   :: GC
character (len=*)                 , intent(IN)      :: SHORT_NAME
character (len=*)  , optional     , intent(IN)      :: LONG_NAME
character (len=*)  , optional     , intent(IN)      :: UNITS
integer            , optional     , intent(IN)      :: DIMS
integer            , optional     , intent(IN)      :: DATATYPE
integer            , optional     , intent(IN)      :: VLOCATION
integer            , optional     , intent(IN)      :: NUM_SUBTILES
integer            , optional     , intent(IN)      :: REFRESH_INTERVAL
integer            , optional     , intent(IN)      :: AVERAGING_INTERVAL
integer            , optional     , intent(IN)      :: PRECISION
real               , optional     , intent(IN)      :: DEFAULT
logical            , optional     , intent(IN)      :: RESTART
character (len=*)  , optional     , intent(IN)      :: HALOWIDTH
character (len=*)  , optional     , intent(IN)      :: FRIENDLYTO
logical            , optional     , intent(IN)      :: ADD2EXPORT
character (len=*)  , optional     , intent(IN)      :: ATTR_INAMES(:)
character (len=*)  , optional     , intent(IN)      :: ATTR_RNAMES(:)
integer            , optional     , intent(IN)      :: ATTR_IVALUES(:)
real               , optional     , intent(IN)      :: ATTR_RVALUES(:)
integer            , optional     , intent(IN)      :: UNGRIDDED_DIMS(:)
integer            , optional     , intent(OUT)     :: RC
```

Only the `ESMF_GridComp` object `GC` 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. For a description of the remaining optional arguments (as well as interfaces to `MAPL_AddImportSpec` and `MAPL_ExportSpec`), please see the MAPL Reference Manual.

3.2.5.2 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 invokes `MAPL_GenericInitialize`. In this section we provide a complete recipe for writing an `Initialize` routine and explain exactly what `MAPL_GenericInitialize` does.

The main reason to write a component-specific `Initalize` is to *handle a* ***`private`*** *internal state*. If all internal state variables can be put in the MAPL `IN` 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.

The following is a complete recipe in the order they would normally appear. **Add commands for each step!?!?!**

1. Get the instance name and setup traceback handle (MAPL_Utils: used for optional error handling)

   ```
   Iam = "Initialize"
   call ESMF_GridCompGet(GC, name=COMP_NAME, CONFIG=CF, RC=STATUS )
   Iam = trim(COMP_NAME) // Iam
   ```

2. Get the MAPL object from the `ESMF_GridComp`

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

   ```
   call MAPL_GetObjectFromGC(GC, MAPL,  RC=STATUS )
   ```

3. If profiling, turn on timer (MAPL_Utils)

   ```
   call MAPL_TimerOn(MAPL,"TOTAL")
   call MAPL_TimerOn(MAPL,"INITIALIZE")
   ```

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

   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 Initalize. 'Resource' in MAPL parlance is the same as 'Attribute' in ESMF.

   ```
   ! can use call ESMF_ConfigGetAttribute(cf, ...) but
   ! the preferred way is the following
   !-------------------------------------------------
   call MAPL_GetResource(MAPL, ...)
   ```

5. Get the component's `private Internal` state from the `ESMF_GridComp`

   If you are writing your own Initialize you will almost certainly be using a ***`private`*** *internal state.*

   ```
   call ESMF_UserCompGetInternalState(GC, 'FVstate', wrap, status)
   state => wrap%dyn_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 - where!?!?!), the grids on those restarts will overide 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.

   ```
   call MAPL_GenericInitialize(GC, IMPORT, EXPORT, CLOCK,  RC=STATUS)
   ```

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 overide 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

For an example of an `Initialize` routine, please refer to section 3.10.

3.2.5.2.1 What does MAPL_GenericInitialize do?

MAPL_GenericInitialize does most of the instance-specific initializations of the MAPL objects. It also creates, and 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: **where is the list!?!?!**

3.2.5.3 Writing a Finalize Method

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

3.2.5.3.1 What does MAPL_GenericFinalize do?

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, and possibly deallocates items in the IM/EX(?)/IN states. MAPL_GenericFinalize also calls the children's Finalize routines.

3.3 Building complex applications: Mapl_Connect

MAPL adopts ESMF's natural hierarchical topology for component connectivity, following the model illustrated in Figure 1. The *leaf* components (no children: 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*). In a typical application, a *composite* component (parent) spawns other (children) components. In our MAPL example (3.10), the parent gridded component GEOS_AgcmSimpleGridComp spawns two children components FVdycore_GridCompMod and GEOS_hsGridCompMod. The registration of the children with MAPL is accomplished by the following calls in the parent's SetServices.

```
dyn = MAPL_AddChild( gc, name='FVDYNAMICS', ss=DYN_SetServices, rc=status)
phs = MAPL_AddChild( gc, name='HSPHYSICS',  ss=PHS_SetServices, rc=status)
```

Each parent's constituent components (its *children*) can then be connected to each other by ESMF couplers (ESMF_CplComp). *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 couplings - whether physical or automatable - occur between *siblings*. This simplifies the placement of couplers, which is important since 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 '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. The following call, made by the parent component, would let MAPL know that it needs certain connectivity services between these children; MAPL will provide these by **automatically** generating the appropriate couplers (`ESMF_CplComp`) (**Or does it just swap pointers!?!?!**), extracting some of the needed information from the data services provided by the children. Once again, this is done in `SetServices`.

```
call MAPL_AddConnectivity                          &
     (GC,                                          &
      SHORT_NAME  = (/ 'DUDT', 'DVDT', 'DTDT' /), &
      SRC_ID      =  PHS,                          &
      DST_ID      =  DYN,                          &
      RC=STATUS)
```

Here, `DUDT`, `DVDT` and `DTDT` are `Import` states of `FVdycore_GridComp` and `Export` states of `GEOS_hsGridComp`. After all connections between the children are processed, their `Import` states may still contain 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 `Import`s are satisfied. **Unresolved** `Import`s at the parent level have to be terminated. In the Held-Suarez example, the child `DYN` of the parent component `GEOS_AgcmSimple` has unresolved `Import`s `PHIS`, `DPEDT` which are terminated in the `SetServices` routine of `GEOS_AgcmSimple`:

```
call MAPL_TerminateImport(GC, SHORT_NAME = (/PHIS ,DPEDT/), &
                          CHILD = DYN, RC=STATUS)
```

In order to have the cousin's `Export` available to the parents, MAPL places all of the children's `Export`s in the parent's `Export` state. This also continues recursively up the hierarchy.

3.3.1 What `MAPL_GenericSetServices` Does with the Children

- Allocates an `ESMF_GridComp` and an `Import` and `Export` state for each child
- Creates each child's `ESMF_GridComp` using the inherited grid and configuration. The i^{th} 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 `SetServices`. [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] - **EXPLAIN**.
- 'Wires' the children. This resolves all child `Import`s that are satisfied by siblings. All such connections must have been added explicitly in `SetServices`.
- Propagates each child's `Export` state to the component's `Export` state.
- Propagate the childrens's unresolved `Import`s to the component's `Import` state.

3.3.2 Rules for Mapl 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 three children: `Root`, `ExtData`, and `History`. The application component creates and initializes the configuration.

3.3.3 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 behavior of an application is controlled through three resource (or configuration) files. The `MAPL_Cap` (main program) opens the configuration files for itself and its three children (`Root` and `History`). These have the default names `CAP.rc`, `ROOT.rc`, `ExtData.rc`, and `HISTORY.rc`. They must be present in the run directory at run time. The name of `MAPL_Cap`'s own resource file is fixed as `Cap.rc`, since this is the resource from which the application 'boots up'. The other two may be renamed in `Cap.rc`. Table (3.1) lists the resources in the `Cap.rc`.

Name	Description	Units	Default
CF_FILE:	Name of ROOT's config file	none	'Root.rc'
CF_FILE:	Name of HISTORY's config file	none	'HISTORY.rc'
TICK_FIRST:	Determines when clock is advanced	1 or 0	none
BEG_YY:	Beginning year (integer)	year	1
BEG_MM:	Beginning month (integer 1-12)	month	1
BEG_DD:	Beginning day of month (integer 1-31)	day	1
BEG_H:	Beginning hour of day (integer 0-23)	hour	0
BEG_M:	Beginning minute (integer 0-59)	minute	0
BEG_S:	Beginning second (integer 0-59)	second	0
END_YY:	Ending year (integer)	year	1
END_MM:	Ending month (integer 1-12)	month	1
END_DD:	Ending day of month (integer 1-31)	day	1
END_H:	Ending hour of day (integer 0-23)	hour	0
END_M:	Ending minute (integer 0-59)	minute	0
END_S:	Ending second (integer 0-59)	second	0
RUN_DT:	App Clock Interval (the Heartbeat)	second	none
LATLON:	1 -> regular lat-lon; 0 -> custom grid	0 or 1	1
NX:	Processing elements in 1st dimension	none	1
NY:	Processing elements in 2nd dimension	none	1
IM_WORLD:	Grid size in 1st dimension	none	none
JM_WORLD:	Grid size in 2nd dimension	none	none
LM:	Grid size in 3rd dimension	none	1
GRIDNAME:	Optional grid name	none	'APPGRID'
IMS:	Gridpoints in each PE along 1st dimension	none	IMS
JMS:	Gridpoints in each PE along 2nd dimension	none	JMS
POLEEDGE:	1->gridedge at pole; 0->gridpoint at pole	0 or 1	0
LON0:	Longituce of center of first gridbox	degree	-90.

Table 3.1: List of resources in `Cap.rc`

An example configuration file (`CAP.rc`) for Example 2 (3.2.4.2) is:

```
NX:     2
NY:     2
IM:     72
JM:     46
LM:     72
BEG_YY: 1991
BEG_MM: 03
BEG_DD: 01
```

```
BEG_H:  0
END_YY: 1991
END_MM: 03
END_DD: 02
END_H:  0
RUN_DT: 1800
ROOT_RC_FILE: HelloWorld.rc
```

An example `HelloWorld.rc` configuration file is simply:

```
RUN_DT: 1800
```

This would perform a one day simulation with 30 minute time steps on a $4^\circ \times 5^\circ$ grid, using a 2×2 decomposition element layout.

For an `ESMF_GridComp`, the configuration may be obtained by querying using the standard ESMF interface, as shown in the run method of Example 2 (3.2.4.2). It can also be queried through the MAPL object by calling `MAPL_GetResource`. *This is the preferred way.* When the configuration is queried this way, 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, MAPL would first look for `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, `DT:`; finally, if this also fails, it would set it to the default value, which in the example is the application's time step, `RUN_DT`.

3.4 Doing Diagnostics: Mapl_History

`MAPL_HistoryGridCompMod` is an internal MAPL gridded component used to manange output streams from a MAPL hierarchy. It writes Fields in the Export states of all MAPL components in a hierarchy to file collections during the course of a run. It also has the some limited capability to interpolate the fields horizontally and/or vertically beofore outputing them.

It is usually one of the two gridded components in the "cap" or main program of a MAPL application, the other being the root of the MAPL hierarchy it is servicing. It is instanciated and all its registered methods are run automatically by `MAPL_Cap,` if that is used. If writing a custom cap, `MAPL_HistoryGridCompMod`'s SetServices can be called anytime after ESMF is initialized. Its Initialize method should be executed before entering the time loop, and its Run method at the bottom of each time loop, after advancing the Clock. 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 MAPL

component, but it generally makes no sense to checkpoint and restart these.

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

`MAPL_HistoryGridCompMod` uses `MAPL_CFIO` for creating and writing its files; it thus obeys all `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 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 `EXDID:` attribute value. A filename extension may also be added to identify the type of file (e.g., .nc4).

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

The extension is not added automatically, it is up to the user to add the appropriate one. If the format is CFIO or CFIOasync and the extension is absent or .nc a NETCDF4 classic file will be produced. Is the extentions is .nc4 a NETCDF4 file will be produced. If it is "flat", the data files have whatever extension you provide 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 ("CFIO", "CFIOasync", "flat"). "CFIO" uses MAPL_CFIO and produces netcdf output. "CFIOasync" uses MAPL_CFIO but delegates the actual I/O to the MAPL_CFIOServer (see MAPL_CFIOServer documenation for details). Default = "flat".

`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 ($\leq$ frequency) for time-averaged diagnostics. Default = `frequency`; ignored if `mode` is 'instantaneous'.

`ref_date` Integer (YYYYMMDD) reference date for *frquency*; 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

`end_time` Integer (HHMMSS) ending time 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 ouput stream. Transforms betwee 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`($\cdot$), `pow`($\cdot$,*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.

`conservative` Set to a non-zero integer to turn on conservative regridding when going from a native cube-sphere grid to lat-lon output. Default: 0

`deflate` Set deflate level (0-9) of NETCDF output when format is CFIO or CFIOasync. Default: 0

`subset` Optional subset (lonMin lonMax latMin latMax) for the output when performing non-conservative cube-sphere to lat-lon regridding of the output.

`chunksize` Optional user specified chunking of NETCDF output when format is CFIO or CFIOasync, (Lon chunksize, Lat chunksize, Lev chunksize, Time chunksize)

The following is a sample HISORY.rc take from the FV_HeldSuarez test.

```
EXPID:  fvhs_example
EXPDSC: 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:     'CFIO',
dynamics_vars_eta.frequency:  240000,
dynamics_vars_eta.duration:   240000,
dynamics_vars_eta.fields:     'T_EQ'     , 'HSPHYSICS'          ,
                              'U'        , 'FVDYNAMICS'         ,
                              'V'        , 'FVDYNAMICS'         ,
                              'T'        , 'FVDYNAMICS'         ,
                              'PLE'      , 'FVDYNAMICS'         ,
                       ::



dynamics_vars_p.template:   '%y4%m2%d2_%h2%n2z',
dynamics_vars_p.format:     'flat',
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 0.7, (SHOULD BE IN ONE LINE??)
dynamics_vars_p.fields:     'T_EQ'     , 'HSPHYSICS'          ,
                            'U'        , 'FVDYNAMICS'         ,
                            'V'        , 'FVDYNAMICS'         ,
                            'T'        , 'FVDYNAMICS'         ,
                            'PLE'      , 'FVDYNAMICS'         ,
                       ::
```

In addition to fields in the EXPORT state of components in the MAPL hierarchy, the user may also specify new fields that will be evaluted using the MAPL arithmetic parser. In place of the variable name when defining fields the user enters an expression that will be evaluated by the MAPL parser (see the documentation for the parser to learn what legal expressions are). In the expression the user can use any field in the collection as a variable in the expression, provided it is an actual field in a MAPL component. In other words new arithemtic fields CAN NOT be used as variables in other expressions in the collection.

On this line the user must specify two more pieces of information. A component name is needed just as in the real fields but this is just a placeholder and will be ignored. Finally a name for the new field is needed. This will be the name of the variable on the resultant file produced by MAPL_HistoryGridCompMod. An example with some comments below makes this clear:

```
  EXPID:  fvhs_example
  EXPDSC: fvhs_(ESMF07_EXAMPLE)_5x4_Deg

  COLLECTIONS:
        'dynamics_vars_eta'
        ::

  dynamics_vars_eta.template:   '%y4%m2%d2_%h2%n2z',
  dynamics_vars_eta.format:     'CFIO',
  dynamics_vars_eta.frequency:  240000,
  dynamics_vars_eta.duration:   240000,
  dynamics_vars_eta.fields:     'T_EQ'     , 'HSPHYSICS'           ,
                                'U'        , 'FVDYNAMICS'          ,
                                'V'        , 'FVDYNAMICS'          ,
                                'sqrt(U*U+V*V)', 'FVDYNAMCIS', 'Wind_Magnitude',
                                'U*2.0e2', 'FVDYNAMICS', 'U_times_two',
                          ::
# T_EQ, U, and V are actual fields in the export state of MAPL components.
# note two new fields that are functions of the real fields in the collection.
# note that in the line 'sqrt(U*U+V*V)', 'FVDYNAMCIS', 'Wind_Magnitude',
# the FVDYNAMICS entry is just a placeholder. It is not used.
# it would be illegal to use Wind_Magnitude in an expression as a variable.
```

3.5 Doing Diagnostics Asynchronously: Mapl_CFIOServer

The MAPL_CFIOServer is a MAPL component that may run concurrently with the GEOS5 model. The server provides a capability to offload I/O in MAPL_History (via CFIO of course) to special nodes that have been set aside for I/O rather than performing the I/O on the nodes running the GEOS5 model as is normally done. Essentially it allows I/O to be performed asynchronously with the model computation assuming it is used properly. The I/O server is started from MAPL_Cap where the global mpi communicator is split. Some processes will run the model as normal and others will run the I/O server, at the discretion of the user. The advantage here is that computation can be overlapped with the I/O rather than having to wait for the History to finish before going on to the next step. The MAPL_CFIOServer runs a master process that continually pulls for I/O requests coming from MAPL_History to use the I/O server to process a collection. All other processes on

the I/O server are considered "workers". If it receives a request then a worker process on the server gets assigned to handle this collection. If no workers are free the request will block in `MAPL_History` until one becomes free. When CFIO is run the data is sent to the worker process on the I/O server rather than written by CFIO. The data on the I/O server is not immediately written but buffered in memory until all levels of the collection have been received. As soon as CFIO has sent all the data to the `MAPL_CFIOServer` the model is free to leave `MAPL_History` and to go on to the next model timestep, independent of any I/O occuring on the server.

The I/O server is normally turned off by default. To use it the user must do three things. They must supply a namelist file to start the server and control the relative number of nodes being dedicated to the model and the I/O server. Additionally it must be specified in the History.rc file that the collection will be written with the I/O server. This is accomplished by changing the format keyword of the collection from "CFIO" to "CFIOasync". Not all collections need to be written with the I/O server. If the format is still "CFIO" and the I/O server has been started the I/O proceeds as normal. Finally the model must be started with a number of processors greater than the product of the layout as it normally would be. For example if the layout is NX = 4 and NY = 24, the GEOS executable would be started on 96 processors. If you want to run the I/O server with this layout you would need to start the model on more than 96 processors, consistent with the namelist file. The namelist file must be named ioserver.nml and has the following format:

```
&ioserver
nnodes = 24 ! number of nodes for used for the model
CoresPerNode = 16  ! cores per node
MaxMem = 26000 ! maximum memory that can be used per node on io server in megabytes
/
```

The first line of the namelist file is the number of nodes that will be used to run the model (NX*NY). Next you must specify the number of cores per node you will be running on. Finally you must specify the maximum memory that can be used per node on the I/O server in megabytes. The I/O server is run on any extra nodes.

The following are some things to consider to make efficient use of the I/O server. The first consideration is that for small jobs, the I/O server is almost certainly not efficient. The reason is that any nodes used by the I/O server could always be used to run the model itself. The incremental speedup when using these nodes for the full model is probably greater than devoting them to speedup just the I/O. Only testing can tell you this. Assuming your problem is I/O bound then make sure the following is true to see the full benefits. First devote sufficient number of nodes such that each time History runs you have at least as many workers as collections and that all the data to be written in a step can be buffered in memory on the I/O nodes. Ideally after History runs and sends the data to the I/O nodes, by the time History runs again the I/O should have occured on the I/O nodes. If any of these are not true you could run into a case where History will have to wait for a free worker on the I/O server, thus negating the purpose of the I/O server.

3.6 Connecting Import Fields to Data on File: Mapl_ExtData

MAPL_ExtDataGridCompMod is an internal MAPL gridded component used to fulfill imports fields in a MAPL hierarchy from netcdf files on disk. It is usually one of the three gridded components in the "cap" or main program of a MAPL application, the others being the root of the MAPL hierarchy it is servicing and MAPL_HistoryGridCompMod. It is instanciated and all its registered methods are run automatically by MAPL_Cap, if that is used. MAPL_ExtDataGridCompMod will provide data to fields in the Import states of MAPL components that are not satisfied by a MAPL_AddConnectivity call in the MAPL hierarchy. In a MAPL application fields added to the Import state of a component are passed up the MAPL hierarchy looking for a connectivity to another component that will provide data to fill the import. If a connectivity is not found these fields will eventually reach the "cap". At this point any fields that have not have their connectivity satisfied are passed to the MAPL_ExtDataGridCompMod through its Export state. MAPL_ExtDataGridCompMod is in essence a provider of last resort for Import fields that need to be filled with data.

The user provides a resource file available to the MAPL_ExtDataGridCompMod GC. At its heart this resource file provides a connection between a field name and a variable name in a netcdf file on disk. The component receives a list of fields that need to be filled and parses the resource file to determine if MAPL_ExtDataGridCompMod can fill a variable of that name. We will refer to each field name-file variable combination as a primary export. Each primary export is an announcment that MAPL_ExtDataGridCompMod is capable of filling a field named A with data contained in variable B on file xyz. Note that the field name in each primary export does not need to actually be a field that needs to be filled by the model. The component only processes primary exports that are needed The resource file should be viewed as an anncountment of what MAPL_ExtDataGridCompMod can provide. In addition to simply announcing what MAPL_ExtDataGridCompMod can provide the user can specify other information such as how frequently to update the data from disk. This could be at every step, just once when starting the model run, or at a particular time each days. MAPL_ExtDataGridCompMod also allows data to be shifted and scaled.

MAPL_ExtDataGridCompMod uses MAPL_CFIO to perform the actual file IO and thus the IO capabilities are limited to files MAPL_CFIO is capable of reading.

MAPL_ExtDataGridCompMod provides an additional method to fill fields. The user can specify any number of derived exports. Each derived export should once again be viewed as an announcement that MAPL_ExtDataGridCompMod has the capability to fill a field with a given name. Instead of the data coming from disk now the user provides a mathematical expression used to evaluate the field which could include fields from the primary exports. The function is any character string that obeys the rules of the MAPL_NewArthParserMod. Any primary export can be used as a variable in the expression, regardless of whether that particular primary export is needed to fulfill an import. Care must be taken however to ensure that the fields are conformal (i.e. on the same grid). For example each field in the expression must have the same number of levels. Likewise attempting to fill a 2D field from a 3D field is nonsensical (see the description of MAPL_NewArthParserMod for an exception).

Any derived exports not needed to fulfill a field are ignored when processing the resource file. As a simple example suppose we have two primary exports with field names A and B. We could define a derived export C that is the sum of A and B.

Finally the component provides the user the capability of creating masks based on a Lat-Lon bounding box. The resulting mask is a real value inside the box and another outside the box. The mask is genereated when the component is initialized and my be used in subsequent derived expressions.

The ExtData.rc file has the following structure. Each entry for the primary exports, derived exports, or mask entry with space separated arguements.

```
PrimaryExports:
field_name Units field_dims level_type climatology
    refresh_template shift scale Name_On_File FileTemplate
    reference_time frequency_units frequency
::
Masks::
mask_name mask_function
::
# the only currently supported mask function is the bounding box called as
# bbox(field_name,lat_min,lat_max,lon_min,lon_max,real_in,real_out)
# field_name is a field_name from the primary export list the function will
# use the grid from this field as the grid of the new maks we create
# lat_min,lat_max, lon_min, and lon_max are the corners of the box
# a real number (see mapl_parser for legal syntax for this). Inside the box
# the mask will be this number. If it is set to "undef" then the value will
# be MAPL_Undef
# a real number, same as above but mask will be this number outside of box
DerivedExports::
field_name expression refresh_template
::
```

Currently for the primary exports, the last 3 keywords in each entry are optional, however all 3 must be specified if used.

The following is a description of a primary export entry:

`field_name` Name of the field to be filled. This should be the same name as the field in the Import state `MAPL_ExtDataGridCompMod` is attempting to fill.

`Units` Units - this is a place holder for the time being and while text must be there is not used

`field_dims` Field dimension - 'xyz' if 3D and 'xy' if 2D

`level_type` 'c' if MAPL_VLocationCenter and 'e' if MAPL_VlocationEdge 'NA' if 2D

`climatology` 'Y' if variable represents climatological data. 'N' if not.

`refresh_template` Refresh template. See section on the refresh template below for more details and function.

`shift` number to shift data, currently the data is always shifted so enter 0.0 if you want no shifting

`scale` number to scale data, currently the data is always scaled so enter 1.0 if you want no scaling

`name_on_file` name of the variable on the file

`FileTemplate` The full path to the file. The actual filename can be the real file name or a grads style template. In addition you can simply set the import to a constant by specifying the entry as /dev/null:real_constant. if no constant is specified after /dev/null with the colon the import is set to zero.

`reference_time` Reference time in the form YYYY-MM-DDThh:mm:ss

`frequency_units` Units of the frequency of the primary export. Valid entries are years, months, days, hours, minutes

`frequency` An integer frequency of the file for this primary export.

The following are descriptions of a mask entry:

`mask_name` Name of the mask

`mask_function` The function used to generate the mask. Currently the only supported function is the bounding box which is specified as bbox(field_name,lat_min,lat_max,lon_min,lon_max,real_in,real_out). The arguements are as follows: field_name is an input field whose grid will be used to create the mask. The next four arguments define the corners of a rectangular bounding box by specifiy the maximum latitudes and longitudes. Finally real_in and real_out are numbers specify what the value of the mask will be inside and outside of the box. In addition to specifying a real number the user may specify "UNDEF" which will make the value MAPL_UNDEF

The following is a description of a derived export entry:

`field_name` Name of the field to be filled

`expression` Legal expression understood by the MAPL parser, for more information see details on legal expressions in the parser documentation. Legal variable names are any field_name in the primary export list or mask_name in the mask list. Remember that the fields in an expression must have the same grid, otherwise an error will result.

`refresh_template` Same options as with the primary export

The Refresh Template controls when the data in the field being satisfied by Extdata is updated and currently has three options:

The first option is set the refresh template to '-'. In this case the field will only be updated once, the first time the run method of `MAPL_ExtDataGridCompMod` is called. The data will be read from the file obtained from evaluating the file template at the current model time.

The second option is to provide a template of the form %y4-%m2-%d2T%h2:%n2:00 where y4, m2, d2, h2, and n2 are the year,month, day, time, and minute. The user then substitutes the time they wish the variable to be periodically updated. This is best illustrated by examples. For example to update the variable daily at 21Z the refresh template should read %y4-%m2-%d2T21:00:00. At 21Z every day `MAPL_ExtDataGridCompMod` will attempt to read a variable from the file obtained by apply the current time to the file template. To update the variable monthly on the 1st of the month at 21Z the refresh template should read %y4-%m2-01T21:00:00.

Finally the refresh_template can be set to '0'. In this case the field will be updated at every time step in the model run. Simply because the field is being updated at every timestep does not mean there has to be data on file for each timestep. If you have the data on file at a larger frequency than the model timestep ExtData can interpolate between the times for which data exists, even if that data is distributed among multiple files.

In order to understand how to use Extdata when the refresh_template is '0' it is helpful to understand what it does it in this case. The basic idea is that for a given model time ExtData tries to find two times, the left and right bracketing time, on the available files that bracket the current time. It then reads the data at those times and performs a linear interpolation to get the data at the current time. It stores the bracketing data for later times to minimize the IO and updates the bracketing data as neccessary when the current time passes the right bracketing time. The data need not reside all on one file.

To get the data Extdata needs to the know what files are available and where to look for the data. Extdata accomplishes this by assuming that the files containing the data are timestamped starting at a reference time at some frequency when the refresh_template

is '0'. For example you could have a daily file, a monthly file, or a file every two hours starting at 13:30 on a particular day, as long as there is data on the files that covers the time you want to run you application and that any times on the file fall within the range of the timestamp. Thus if you have a file with the template myfile.%y4%m2%d2.nc4, one particular file might be myfile.20010414.nc4 and any times in this file must be sometime on the 14th of march 2001.

There are two ways to specify the reference time and frequency. The first is to use the three optional keywords for a primary export. As described in the section on primary exports you specify a reference time, and a frequency with some units. When ExtData tries to find the bracketing data it starts by finding a file with a timestamp closest to but less than or equal to the current time. If it does not find a bracketing time there it checks the file with the timestamp at the frequency interval before and after the file it just checked.

The second is to let ExtData figure out the frequency from the file template itself and leaving the final three entries in the primary export blank. In that case it determines the rightmost token in the grads template and uses that as the frequency. For example if the template is filename.%y4%m2%d2.nc4 the frequency will be assumed to be days. The reference time is taken to be the application start time at the beginning with any time units less than the frequency assumed to be zero. So using the same file template again if we start on 2001-04-14 at 21z the reference time will be set to 2001-04-14 at 0z.

To hopefully make this clearer here are several concrete examples showing how it might be used.

Suppose we have data on daily files and each file has data at 0Z, 6Z, 12Z, and 18Z. The file template is myfile.%y4%m2%d2.nc4 and we leave the reference time and frequency blank.

```
if the clock starts at 21Z on January 1, 2001 these are the initial bracketing
times data is read from

myfile.20010101.nc4
0Z
6Z
12Z
18Z    Left bracket time
myfile.20010102.nc4
0Z     Right bracket time
6Z
12Z
18Z

When the time passes 0Z on 2nd of month we must update bracket time
and now the new bracketing times, for example
when it is 2Z on 2nd of month the data for the following times will be in memory
```

```
myfile.20010101.nc4
0Z
6Z
12Z
18Z
myfile.20010102.nc4
0Z     Left bracket time
6Z     Right bracket time
12Z
18Z
```

Now suppose we have data on files every two hours with one time per file, the time on file being the same as the timestamp on the file. The first file is at 2001-01-01 at 01:30:00 Then the reference time is 2001-01-01T01:30:00, with a frequency of 2 hours. The file template must be of the form myfile.%y4%m2%d2_%h2%n2z.nc4

```
if the clock starts at 2Z on January 1, 2001 and we have a series of files.
The initial bracket times are:

myfile.20010101_0130.nc4 Left bracket Time
myfile.20010101_0330.nc4 Right bracket Time
myfile.20010101_0530.nc4

Now when it is 4Z the bracket times are

myfile.20010101_0130.nc4
myfile.20010101_0330.nc4 Left bracket Time
myfile.20010101_0530.nc4 Right bracket Time
```

Now suppose we have monthly climatologies. The file template is of the form myfile.%y4%m2.nc4 Each file has data for one time, at noon on the 15th of the month. We leave the reference time and frequency blank.

```
if the clock starts at 2001-04-01 0z then initial bracket times and data stored are
myfile.200103.nc4
2001-03-15T12:00:00 left bracket time
myfile.200104.nc4
2001-04-15T12:00:00 right bracket time

later then time is 2001-04-17 0z then the bracket times and data stored are
```

```
myfile.200104.nc4
2001-04-15T12:00:00 left bracket time
myfile.200105.nc4
2001-05-15T12:00:00 right bracket time
```

3.7 Performing Arithemtic Operations on Fields: Mapl_NewArthParser

`MAPL_NewArthParserMod` is a module that provides a mathematical parsing capabiltiy to MAPL and ESMF components. The module evaluates an ESMF field element by element using an expression which could contain other ESMF fields as variables. Examples of use for this component would be to take the Log of every element in a field or add two fields element by element. The heart of the module is a simple public routine MAPL_StateEval(state,expression,field,rc) with three required arguments. The arguments are as follows:
state is an `ESMF_State` type containing some number of fields.
expression is a character string containing a valid mathematical function string.
field is an `ESMF_Field` type that will be filled using the expression.

The expression can contain the names of any fields in the state as variables.

The following can appear in the expression string
1. The function string can contain the following mathematical operators +, -, *, /, ^ and ()
2. Variable names - these can be any field name in the input state. Parsing of variable names is case sensitive.
3. The following single argument fortran intrinsic functions and user defined functions are implmented: exp, log10, log, sqrt, sinh, cosh, tanh, sin, cos, tan, asin, acos, atan, heav (the Heaviside step function). Parsing of functions is case insensitive.
4. Integers or real constants. To be recognized as explicit constants these must conform to the format

[+|-][nnn][.nnn][e|E|d|D[+|-]nnn]

where nnn means any number of digits. The mantissa must contain at least one digit before or following an optional decimal point. Valid exponent identifiers are 'e', 'E', 'd' or 'D'. If they appear they must be followed by a valid exponent!

Operations are evaluated in the order

1. () expressions in brackets
2. -X unary minux

3. X^Y exponentiation
4. X*Y X/Y multiplicaiton and division
5. A+B X-Y addition and subtraction

One logical requirement is that the fields in the state and the field being filled are on the same grid, including vertical levels. For example, 3D fields in an expression must ALL have been created with the same vertical levels (MAPL_DimsVLocationEdge or MAPL_DimsVLocationCenter). If not an error will result. The one exception is operations involving 3D and 2D fields when the resultant field is a 3D field. In this case, the operation is performed between each level of the 3D field and the 2D field. This is useful if one wanted to scale each level of a 3D field with the same 2D field. Of course the first two dimensions of the 3D field must be same as the 2D field.

The parser also obeys undef arithmetic. Any arithemtic operation involving MAPL_Undef or function of MAPL_Undef results in MAPL_Undef.

The following are several examples of valid expressions. For the examples we will assume that the input state has 4 fields A, B, C, and D.
1. B*2.0e0
2. sqrt(A*A+B*B)
3. A*heav(B)
4. A^(C+D)-2.0e-3

3.8 Doing I/O: Mapl_CFIO

`MAPL_CFIO` interfaces CFIO to the ESMF data types. It currently includes read-write support for `ESMF_Field`s and `ESMF_State`s, and read support for `ESMF_Field`s and `Fortran arrays`. It has only four methods:

1. `MAPL_CFIOCreate`
2. `MAPL_CFIOWrite`
3. `MAPL_CFIORead`
4. `MAPL_CFIODestroy`

Except for `MAPL_Read`, all work on a `MAPL_CFIO` object. Reading is done directly from a file to the appropriate ESMF object.

`MAPL_CFIO` is designed for two modes of I/O: self-describing formats (SDF), of which it supports HDF-4 (not 5!?!?!) 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_CFIO` depends on which of the two libraries (HDF or NetCDF) is linked with the application - both cannot be used because of 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.

3.9 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_Field`s, 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_Field`s or `ESMF_Bundle`s within the three states. In user code these can be extracted directly and manipulated as `ESMF_Field`s when using ESMF infrastructure, or one can extract Fortran pointers to the data when interfacing to existing user code.

MAPL provides several features that, although not central to its main goals, can be very handy. Some of these provide functionality in an instance-specific way by saving metadata in the MAPL object. This saves 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.

3.9.1 Error Handling

The error handling utility consists of the three macros:

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

These are used by setting the local character string variable `Iam` to the soubroutine 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.

3.9.2 Profiling

The API of the profiling utility consists three subroutines:

```
MAPL_TimerAdd(MAPL, NAME, RC)
MAPL_TimerOn (MAPL, NAME, RC)
MAPL_TimerOff(MAPL, NAME, RC)
```

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 `MAPL_GenericFinalize` the results are reported to standard out. Even if the user registers no meters, the performance of the generic `IRF` methods is reported.

3.9.3 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:

```
MAPL_SunGetInsolation(LONS, LATS, ORBIT,ZTH,SLR,INTV,CLOCK,TIME,RC)
```

where `LONS` and `LATS` can be either one- or two-dimensional Fortran arrays or general `ESMF_array`s 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^{th} orbital parameters. These can be overidden in the configuration by specifying `ECCENTRICITY:`, `OBLIQUITY:`, `PERIHELION:`, and `EQUINOX:`. The meanings of these, as well as more complex uses of the astronomy are also explained in the prologues of `MAPL_SunMod` in Part II (!?!?!).

3.9.4 Universal Constants

The following universal constants are defined when `MAPLMod` is used:

Table 3.2: Table of universal constants

MAPL_PI	3.14159265358979323846	---
MAPL_GRAV	9.80	m s^{-2}
MAPL_RADIUS	6376.0e3	m
MAPL_OMEGA	2.0*MAPL_PI/86164.0	s^{-1}
MAPL_ALHL	2.4548e6	J kg^{-1}
MAPL_ALHS	2.8368e6	J kg^{-1}
MAPL_ALHF	MAPL_ALHS-MAPL_ALHL	J kg^{-1}
MAPL_STFBOL	5.6734e-8	W m^{-2} K^{-4}
MAPL_AIRMW	28.97	kg $Kmol^{-1}$
MAPL_H2OMW	18.01	kg $Kmol^{-1}$
MAPL_RUNIV	8314.3	J $Kmol^{-1}$ K^{-1}
MAPL_KAPPA	2.0/7.0	---
MAPL_RVAP	MAPL_RUNIV/MAPL_H2OMW	J K^{-1} kg^{-1}
MAPL_RGAS	MAPL_RUNIV/MAPL_AIRMW	J K^{-1} kg^{-1}
MAPL_CP	MAPL_RGAS/MAPL_KAPPA	J K^{-1} kg^{-1}
MAPL_P00	100000.0	Pa
MAPL_CAPWTR	4218.	J K^{-1} kg^{-1}
MAPL_RHOWTR	1000.	kg m^{-3}
MAPL_NUAIR	1.533e-5	m^2 S^{-1} (@ 18C)
MAPL_TICE	273.16	K
MAPL_UNDEF	-999.0	---
MAPL_SRFPRS	98470.	Pa
MAPL_KARMAN	0.40	---
MAPL_USMIN	1.00	m s^{-1}
MAPL_VIREPS	MAPL_AIRMW/MAPL_H2OMW-1.0	---

3.10 A complete MAPL example - Held-Suarez benchmark for FVdycore

Provide a description of this application — how the application is structured, what the components are etc.

Appendix A

MAPL Application Programming Interface (API)

Contents

A.1 Module MAPL_CapMod — Implements the top entry point for MAPL components

USES:

```
use ESMF
use MAPL_BaseMod
use MAPL_ConstantsMod
use MAPL_ProfMod
use MAPL_MemUtilsMod
use MAPL_IOMod
use MAPL_CommsMod
use MAPL_GenericMod
use MAPL_LocStreamMod
use ESMFL_Mod
use MAPL_ShmemMod
use MAPL_HistoryGridCompMod, only : Hist_SetServices  = > SetServices
use MAPL_HistoryGridCompMod, only : HISTORY_ExchangeListWrap
use MAPL_ExtDataGridCompMod, only : ExtData_SetServices  = > SetServices
use MAPL_CFIOServerMod
```

PUBLIC MEMBER FUNCTIONS:

```
public MAPL_Cap
```

DESCRIPTION:

The main program (or, in ESMF lingo, the Cap) of any ESMF application is provided by the user. In MAPL, it initiates the execution of each of the sub-hierarchies of the application (SetServices, Initialize, Run, Finalize, and the new Record). Usually, each of these, except Run and Record, is executed only once.

In MAPL applications, the Cap contains the time loop. The hierarchy of Run methods is called each time through the loop, returning control to the Cap after each round trip down and back up the hierarchy. The Run hierarchy must be invoked once and only once each time through the time loop.

The time loop advances the current time of the Application Clock – the ESMF Clock that is passed down to all registered methods of all components in the hierarchy. MAPL applications require that the Application Clock be "ticked" after the Run method is invoked,

but before the Record. The time interval of the Application Clock is called the heartbeat in MAPL.

Since the Cap is a main program, it is not an ESMF Gridded Component. A MAPL component's cap, however, has "children" and treats them much as any Composite Component would. In particular, it registers them with MAPL by invoking a MAPL_AddChild for each one.

The Cap for a MAPL application has only two children: a single instance of the Root Component of a MAPL hierarchy and a single instance of MAPL's own History Component. The History Component services the computational components' diagnostic output.

As might be expected from this simple set of rules, all MAPL Caps are very similar. We have therefore gathered the basic MAPL Cap functionality in a single Fortran subroutine (MAPL_Cap) that is included in the MAPL library.

For basic MAPL Caps, a call to this subroutine is the only required executable statement of the main program. As an example, the following code is the entire main program of the Held-Suarez example:

```
#define I_AM_MAIN
#include "MAPL_Generic.h"

Program Main

  use MAPL_Mod
  use GEOS_AgcmSimpleGridCompMod, only:  ROOT_SetServices => SetServices

  implicit none
  integer           :: STATUS
  character(len=18) :: Iam="Main"

  call MAPL_CAP(ROOT_SetServices, rc=STATUS)
  VERIFY_(STATUS)

  call exit(0)

end Program Main
```

Notice that, besides calling the MAPL_Cap subroutine, the only purpose of this program is to identify the root component of the MAPL hierarchy by accessing its SetServices through use association. The rest of the code is a bit of MAPL boilerplate. In fact, doing away with MAPL and Fortran niceties, the code can be reduced to:

```
Program Main

  use GEOS_AgcmSimpleGridCompMod, only:  ROOT_SetServices => SetServices
  call MAPL_CAP(ROOT_SetServices)

end Program Main
```

In either case, only the name GEOS_AgcmSimpleGridCompMod needs to be modified to use these codes in another application.

In using MAPL, it is important to know exactly what boilerplate routines, such as MAPL_Cap, are doing for you, so that you can supplement or replace them with custom code, if necessary. MAPL_Cap is simple enough that it is probably easier to look at its full code than to try to describe its functioning in detail. Studying this code should also be useful if one decides to write a more specialized custom version to replace it.

A.1.1 MAPL_Cap – Implements generic Cap functionality

INTERFACE:

```
subroutine MAPL_CAP(ROOT_SetServices, Name, AmIRoot, FinalFile, RC)
```

ARGUMENTS:

```
external                              :: ROOT_SetServices
character*(*), optional, intent(IN ) :: Name
logical,       optional, intent(OUT) :: AmIRoot
character*(*), optional, intent(IN ) :: FinalFile
integer,       optional, intent(OUT) :: rc
```

RESOURCES:

Name	Description	Units	Default
"ROOT_CF:"	Name of ROOT's config file	string	"ROOT.rc"

Name	Description	Units	Default
"ROOT_NAME:"	Name to assign to the ROOT component	string	"ROOT"
"HIST_CF:"	Name of HISTORY's config file	string	"HIST.rc"
"EXTDATA_CF:"	Name of ExtData's config file	string	'ExtData.rc'
"MAPL_ENABLE_TIMERS:"	Control Timers	string	'NO'
"MAPL_ENABLE_MEMUTILS:"	Memory Diagnostic Utility	string	'NO'

RESOURCES:

Name	Description	Units	Default
'BEG_DATE:'			
'BEG_YY:'	Beginning year (integer)	year	
'BEG_MM:'	Beginning month (integer 1-12)	month	1
'BEG_DD:'	Beginning day of month (integer 1-31)	day	1
'BEG_H:'	Beginning hour of day (integer 0-23)	hour	0
'BEG_M:'	Beginning minute (integer 0-59)	minute	0
'BEG_S:'	Beginning second (integer 0-59)	second	0
'END_DATE:'			
'END_YY:'	Ending year (integer)	year	
'END_MM:'	Ending month (integer 1-12)	month	1
'END_DD:'	Ending day of month (integer 1-31)	day	1
'END_H:'	Ending hour of day (integer 0-23)	hour	0
'END_M:'	Ending minute (integer 0-59)	minute	0
'END_S:'	Ending second (integer 0-59)	second	0
'JOB_SGMT:'			
'JOB_DURATION:'			
'DUR_YY:'	Ending year (integer)	year	
'DUR_MM:'	Ending month (integer 1-12)	month	0
'DUR_DD:'	Ending day of month (integer 1-31)	day	1
'DUR_H:'	Ending hour of day (integer 0-23)	hour	0
'DUR_M:'	Ending minute (integer 0-59)	minute	0
'DUR_S:'			0
'HEARTBEAT_DT:'	Interval of the application clock (the Heartbeat)	seconds	
'NUM_DT:'	numerator of decimal fraction of time step	1	0
'DEN_DT:'	denominator of decimal fraction of time step	1	1
'CALENDAR:'	Calendar type	string	"GREGORIAN"

A.2 Module MAPL_GenericMod

DESCRIPTION:

MAPL_Generic allows the user to easily build ESMF gridded components. It has its own SetServices, Initialize, Run, and Finalize (IRF) methods, and thus is itself a valid gridded component, although somewhat non-standard since it makes its IRF methods public. An instance of MAPL_Generic does no useful work, but can be used as a null MAPL_Generic component.

The standard way to use MAPL_Generic is as an aid in building ESMF gridded components. A MAPL/ESMF gridded component built in this way will always have its own SetServices, which will call the subroutine MAPL_GenericSetServices. When MAPL_GenericSetServices is called it sets the component's IRF methods to the generic versions, MAPL_GenericInitialize, MAPL_GenericFinalize, and MAPL_GenericRun. Any (or all) of these may be used as default methods by a gridded component. (As we will see below, using all three default IRF methods in this way need not be equivalent to instanciating a null component.) If for any of the three IRF methods the default version is inadequate, it can simply be overrided by having the component register its own method after the call to MAPL_GenericSetServices.

The generic IRF methods perform a number of useful functions, including creating, allocating, and initializing the components Import, Export, and Internal states. It would be a shame to waste this capability when a component needs to write its own version of an IRF method. A common situation is that the component wants support in performing these functions, but needs to do some (usually small) additional specialized work; for example, it may need to do some special initializations. In this case, one would write a light version of the IRF method that does the specialized work and *calls directly* the corresponding MAPL_Generic method to do the boilerplate. This is why MAPL_Generic, unlike a standard ESMF gridded component, makes its IRF methods public and why we added the "Generic" modifier (i.e., MAPL_GenericInitialize, rather than MAPL_Initialize), to emphasize that they are directly callable IRF methods.

MAPL_Generic may also be viewed as a fairly standard Fortran 90 "class," which defines and makes public an opaque object that we refer to as a "MAPL_Generic State." This object can be created only in association with a standard ESMF Gridded Component (GC), by making a MAPL_GenericSetServices call. This object can be obtained through an ESMF GC method which is currently provided with MAPL. The MAPL_Generic State is, therefore, just another thing that lives in the ESMF GC, like the grid and the configuration. The MAPL_Generic State is private, but user components can access its contents through public MAPL_Generic methods (Get, Set, etc). The bulk of MAPL_Generic consists of methods that act on this object.

MAPL_GenericSetServices and MAPL_Generic IRF methods cannot create their own ESMF grid. The grid must be inherited from the parent or created by the component either in its own SetServices or in its Initialize, if it is writing one. In any case, an important assumption of MAPL is that the grid must already be *present in the component and initialized* when MAPL_GenericSetServices is invoked. The same is true of the configuration.

In MAPL_Generic, we distinguish between *simple (leaf)* gridded compnents and *composite* gridded components, which contain other (*child*) gridded components. We also define three types of services, which can be registered by the component's SetServices routine.

- **Functional services**: These are the standard EMSF callable IRF methods for the component.
- **Data services:** These are descriptions of the component's import, export, and internal states, which can be manipulated by MAPL_Generic.
- **Child services:** These are the services of the component's children and their connectivity.
- **Profiling Services:** These are profiling counters (clocks) that can be used by the component and are automatically reported by generic finalize.

MAPL_GenericSetServices provides generic versions of all these, as described below.

USES:

```
use ESMF
use ESMFL_Mod
use MAPL_BaseMod
use MAPL_IOMod
use MAPL_CFIOMod
use MAPL_ProfMod
use MAPL_MemUtilsMod
use MAPL_CommsMod
use MAPL_ConstantsMod
use MAPL_SunMod
use MAPL_VarSpecMod
use MAPL_GenericCplCompMod
use MAPL_LocStreamMod
use m_chars
use, intrinsic :: ISO_C_BINDING
```

PUBLIC MEMBER FUNCTIONS:

```
public MAPL_GenericSetServices
public MAPL_GenericInitialize
public MAPL_GenericRun
public MAPL_GenericFinalize

public MAPL_AddInternalSpec
public MAPL_AddImportSpec
public MAPL_AddExportSpec

public MAPL_DoNotDeferExport

public MAPL_GridCompSetEntryPoint
public MAPL_GetObjectFromGC
public MAPL_Get
public MAPL_Set
public MAPL_GenericRunCouplers

!public MAPL_StateGetSpecAttrib
!public MAPL_StateSetSpecAttrib
!public MAPL_StateGetVarSpecs
!public MAPL_StatePrintSpec
public MAPL_StatePrintSpecCSV

! MAPL_Connect
public MAPL_AddChild
public MAPL_AddConnectivity
public MAPL_TerminateImport

! MAPL_Util
!public MAPL_GenericStateClockOn
!public MAPL_GenericStateClockOff
!public MAPL_GenericStateClockAdd
public MAPL_TimerOn
public MAPL_TimerOff
public MAPL_TimerAdd
public MAPL_GetResource
public MAPL_ReadForcing
```

PUBLIC TYPES:

```
public MAPL_MetaComp
```

This defines the MAPL_Generic class. It is an opaque object that can be queried using `MAPL_GenericStateGet`. An instance of this type is placed in the default internal state location of the ESMF gridded component by `MAPL_GenericSetServices`. This instance can be retreived by using `MAPL_InternalStateGet`.

CODE:

```
type  MAPL_MetaComp
   private
   type (ESMF_GridComp          ), pointer :: GCS(:)            = > null()
   type (ESMF_State             ), pointer :: GIM(:)            = > null()
   type (ESMF_State             ), pointer :: GEX(:)            = > null()
   type (ESMF_CplComp           ), pointer :: CCS(:,:)          = > null()
   type (ESMF_State             ), pointer :: CIM(:,:)          = > null()
   type (ESMF_State             ), pointer :: CEX(:,:)          = > null()
   logical,                        pointer :: CCcreated(:,:)    = > null()
   type (MAPL_GenericGrid       )          :: GRID
   type (ESMF_Alarm             )          :: ALARM(0:LAST_ALARM)
   integer                                 :: ALARMLAST = 0
   type (ESMF_Clock             )          :: CLOCK
   type (ESMF_Config            )          :: CF
   type (ESMF_State             )          :: INTERNAL
   type (MAPL_SunOrbit          )          :: ORBIT
   type (MAPL_VarSpec           ), pointer :: IMPORT_SPEC(:)    = > null()
   type (MAPL_VarSpec           ), pointer :: EXPORT_SPEC(:)    = > null()
   type (MAPL_VarSpec           ), pointer :: INTERNAL_SPEC(:)  = > null()
   type (MAPL_VarSpec           ), pointer :: FRCSPEC(:)   = > null()
   type (MAPL_Prof              ), pointer :: TIMES(:)          = > null()
   character(len = ESMF_MAXSTR)     , pointer :: GCNameList(:)     = > null()
   type(ESMF_GridComp)                     :: RootGC
   type(ESMF_GridComp)           , pointer :: parentGC          = > null()
   logical                                 :: ChildInit  =  .true.
   type (MAPL_Link)              , pointer :: LINK(:)           = > null()
   type (MAPL_LocStream)                   :: ExchangeGrid
   type (MAPL_LocStream)                   :: LOCSTREAM
   character(len = ESMF_MAXSTR)              :: COMPNAME
   type (MAPL_GenericRecordType)  , pointer :: RECORD           = > null()
```

```
   type (ESMF_State)                          :: FORCING
   integer                          , pointer :: phase_init (:)     = > null()
   integer                          , pointer :: phase_run  (:)     = > null()
   integer                          , pointer :: phase_final(:)     = > null()
   integer                          , pointer :: phase_record(:)    = > null()
   integer                          , pointer :: phase_coldstart(:) = > null()
   real                                       :: HEARTBEAT
   type (MAPL_Communicators)                  :: comm
end type MAPL_MetaComp
```

A.2.1 MAPL_GenericSetServices

INTERFACE:

```
recursive subroutine MAPL_GenericSetServices ( GC, RC )
```

ARGUMENTS:

```
   type(ESMF_GridComp),                  intent(INOUT) :: GC  ! Gridded component
   integer,                              intent(  OUT) :: RC  ! Return code
```

DESCRIPTION:

`MAPL_GenericSetServices` performs the following tasks:

- Allocate an instance of MAPL_GenericState, wrap it, and set it as the GC's internal state.
- Exract the grid and configuration from the GC and save them in the generic state.
- Set GC's IRF methods to the generic versions
- If there are children
  - Allocate a gridded comoponent and an import and export state for each child
  - Create each child's GC using the natural grid and the inherited configuration.
  - Create each child's Import and Export states. These are named `GCNames(I)//"_IMPORT"` and `GCNames(I)//"_EXPORT"`

- Invoke each child's set services.
- Add each item in each child's export state to GC's export state.
- Add each item in each child's import state to GC's import, eliminating duplicates.

Since `MAPL_GenericSetServices` calls SetServices for the children, which may be generic themselves, the routine must be recursive.

The optional arguments describe the component's children. There can be any number of children but they must be of one of the types specified by the five SetServices entry points passed. If SSptr is not specified there can only be five children, one for each `SSn`, and the names must be in `SSn` order.

A.2.2 MAPL_GenericInitialize – Initializes the component and its children

INTERFACE:

```
recursive subroutine MAPL_GenericInitialize ( GC, IMPORT, EXPORT, CLOCK, RC )
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(INOUT) :: GC     ! Gridded component
  type(ESMF_State),    intent(INOUT) :: IMPORT ! Import state
  type(ESMF_State),    intent(INOUT) :: EXPORT ! Export state
  type(ESMF_Clock),    intent(INOUT) :: CLOCK  ! The clock
  integer, optional,   intent(  OUT) :: RC     ! Error code:
```

A.2.3 MAPL_GenericRun

INTERFACE:

```
recursive subroutine MAPL_GenericRun ( GC, IMPORT, EXPORT, CLOCK, RC)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(INOUT) :: GC     ! Gridded component
type(ESMF_State),    intent(INOUT) :: IMPORT ! Import state
type(ESMF_State),    intent(INOUT) :: EXPORT ! Export state
type(ESMF_Clock),    intent(INOUT) :: CLOCK  ! The clock
integer, optional,   intent(  OUT) :: RC     ! Error code:
```

A.2.4 MAPL_GenericFinalize – Finalizes the component and its children

INTERFACE:

```
recursive subroutine MAPL_GenericFinalize ( GC, IMPORT, EXPORT, CLOCK, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(inout) :: GC     ! composite gridded component
type(ESMF_State),    intent(inout) :: IMPORT ! import state
type(ESMF_State),    intent(inout) :: EXPORT ! export state
type(ESMF_Clock),    intent(inout) :: CLOCK  ! the clock
integer, optional,   intent(  out) :: RC     ! Error code:
                                             !  =  0 all is well
                                             ! otherwise, error
```

A.2.5 MAPL_StateAddImportSpec — Sets the specifications for an item in the IMPORT state.

A.2.5.1

INTERFACE:

```
subroutine MAPL_StateAddImportSpec (GC, SHORT_NAME, LONG_NAME,              &
```

```
                                    UNITS,  Dims, VLocation,                &
                                    DATATYPE,NUM_SUBTILES, REFRESH_INTERVAL, &
                                    AVERAGING_INTERVAL, HALOWIDTH, DEFAULT,  &
                                    RESTART, UNGRIDDED_DIMS, FIELD_TYPE, RC)
```

ARGUMENTS:

```
type (ESMF_GridComp)            , intent(INOUT)   :: GC
character (len = *)               , intent(IN)      :: SHORT_NAME
character (len = *)  , optional   , intent(IN)      :: LONG_NAME
character (len = *)  , optional   , intent(IN)      :: UNITS
integer            , optional   , intent(IN)      :: DIMS
integer            , optional   , intent(IN)      :: DATATYPE
integer            , optional   , intent(IN)      :: NUM_SUBTILES
integer            , optional   , intent(IN)      :: VLOCATION
integer            , optional   , intent(IN)      :: REFRESH_INTERVAL
integer            , optional   , intent(IN)      :: AVERAGING_INTERVAL
integer            , optional   , intent(IN)      :: HALOWIDTH
real               , optional   , intent(IN)      :: DEFAULT
logical            , optional   , intent(IN)      :: RESTART
integer            , optional   , intent(IN)      :: UNGRIDDED_DIMS(:)
integer            , optional   , intent(IN)      :: FIELD_TYPE
integer            , optional   , intent(OUT)     :: RC
```

A.2.5.2 Add IMPORT spec from child

INTERFACE:

```
subroutine MAPL_StateAddImportSpec        ( STATE, SHORT_NAME, CHILD_ID, RC )
```

ARGUMENTS:

```
type (MAPL_MetaComp)            , intent(INOUT)   :: STATE
character (len = *)               , intent(IN)      :: SHORT_NAME
integer                         , intent(IN)      :: CHILD_ID
integer            , optional   , intent(OUT)     :: RC
```

A.2.6 MAPL_StateAddExportSpec — sets the specifications for an item in the EXPORT state

A.2.6.1

INTERFACE:

```
subroutine MAPL_StateAddExportSpec (GC, SHORT_NAME, LONG_NAME,           &
                                    UNITS, Dims, VLocation,              &
                                    DATATYPE,NUM_SUBTILES,               &
                                    REFRESH_INTERVAL, AVERAGING_INTERVAL, &
                                    HALOWIDTH, DEFAULT, UNGRIDDED_DIMS,  &
                                    FIELD_TYPE, RC )
```

ARGUMENTS:

```
type (ESMF_GridComp)            , intent(INOUT)   :: GC
character (len = *)               , intent(IN)      :: SHORT_NAME
character (len = *)  , optional   , intent(IN)      :: LONG_NAME
character (len = *)  , optional   , intent(IN)      :: UNITS
integer             , optional   , intent(IN)      :: DIMS
integer             , optional   , intent(IN)      :: DATATYPE
integer             , optional   , intent(IN)      :: VLOCATION
integer             , optional   , intent(IN)      :: NUM_SUBTILES
integer             , optional   , intent(IN)      :: REFRESH_INTERVAL
integer             , optional   , intent(IN)      :: AVERAGING_INTERVAL
integer             , optional   , intent(IN)      :: HALOWIDTH
real                , optional   , intent(IN)      :: DEFAULT
integer             , optional   , intent(IN)      :: UNGRIDDED_DIMS(:)
integer             , optional   , intent(IN)      :: FIELD_TYPE
integer             , optional   , intent(OUT)     :: RC
```

A.2.6.2 Add EXPORT spec from child

INTERFACE:

```
subroutine MAPL_StateAddExportSpec        ( GC, SHORT_NAME, CHILD_ID, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp),              intent(INOUT)   :: GC
character (len = *)               , intent(IN)      :: SHORT_NAME
integer                         , intent(IN)      :: CHILD_ID
integer            , optional   , intent(OUT)     :: RC
```

A.2.6.3 Add EXPORT spec from all

INTERFACE:

```
subroutine MAPL_StateAddExportSpec       ( STATE, RC )
```

ARGUMENTS:

```
type (MAPL_MetaComp)            , intent(INOUT)   :: STATE
integer            , optional   , intent(OUT)     :: RC
```

A.2.7 MAPL_AddInternalSpec

A.2.7.1 Sets specifications for an item in the INTERNAL state

INTERFACE:

```
subroutine MAPL_AddInternalSpec     (GC,                &
                                     SHORT_NAME,        &
                                     LONG_NAME,         &
                                     UNITS,             &
                                     DIMS,              &
                                     VLOCATION,         &
                                     DATATYPE,          &
                                     NUM_SUBTILES,      &
                                     REFRESH_INTERVAL,  &
```

```
                                    AVERAGING_INTERVAL, &
                                    DEFAULT,            &
                                    RESTART,            &
                                    HALOWIDTH,          &
                                    PRECISION,          &
                                    FRIENDLYTO,         &
                                    ADD2EXPORT,         &
                                    ATTR_RNAMES,        &
                                    ATTR_INAMES,        &
                                    ATTR_RVALUES,       &
                                    ATTR_IVALUES,       &
                                    UNGRIDDED_DIMS,     &
                                    FIELD_TYPE,         &
                                    RC)
```

ARGUMENTS:

```
type (ESMF_GridComp)            , intent(INOUT)   :: GC
character (len = *)               , intent(IN)      :: SHORT_NAME
character (len = *)  , optional   , intent(IN)      :: LONG_NAME
character (len = *)  , optional   , intent(IN)      :: UNITS
integer             , optional   , intent(IN)      :: DIMS
integer             , optional   , intent(IN)      :: DATATYPE
integer             , optional   , intent(IN)      :: VLOCATION
integer             , optional   , intent(IN)      :: NUM_SUBTILES
integer             , optional   , intent(IN)      :: REFRESH_INTERVAL
integer             , optional   , intent(IN)      :: AVERAGING_INTERVAL
integer             , optional   , intent(IN)      :: PRECISION
real                , optional   , intent(IN)      :: DEFAULT
logical             , optional   , intent(IN)      :: RESTART
character (len = *)  , optional   , intent(IN)      :: HALOWIDTH
character (len = *)  , optional   , intent(IN)      :: FRIENDLYTO
logical             , optional   , intent(IN)      :: ADD2EXPORT
character (len = *)  , optional   , intent(IN)      :: ATTR_INAMES(:)
character (len = *)  , optional   , intent(IN)      :: ATTR_RNAMES(:)
integer             , optional   , intent(IN)      :: ATTR_IVALUES(:)
real                , optional   , intent(IN)      :: ATTR_RVALUES(:)
integer             , optional   , intent(IN)      :: UNGRIDDED_DIMS(:)
integer             , optional   , intent(IN)      :: FIELD_TYPE
integer             , optional   , intent(OUT)     :: RC
```

DESCRIPTION:

Sets the specifications for an item in the INTERNAL state.

A.2.8 MAPL_DoNotDeferExport

INTERFACE:

```
subroutine MAPL_DoNotDeferExport(GC, NAMES, RC)
```

ARGUMENTS:

```
type (ESMF_GridComp)            , intent(INOUT)   :: GC
character (len = *)               , intent(IN)      :: NAMES(:)
integer            , optional   , intent(OUT)     :: RC
```

DESCRIPTION:

For each entry in NAMES marks the export spec to not be deferred during MAPL_GenericInitialize.

A.2.9 MAPL_GridCompSetEntryPoint

INTERFACE:

```
subroutine MAPL_GridCompSetEntryPoint(GC, subroutineType, subroutineName, RC)
```

ARGUMENTS:

```
type(ESMF_GridComp),                  intent(INOUT) :: GC         ! Gridded component
type(ESMF_Method_Flag),               intent(IN   ) :: subroutineType
external                                            :: subroutineName
integer,                    optional, intent(  OUT) :: RC         ! Return code
```

A.2.10 MAPL_GetObjectFromGC

A.2.10.1

INTERFACE:

```
subroutine MAPL_GetObjectFromGC  ( GC, MAPLOBJ, RC)
```

ARGUMENTS:

```
type(ESMF_GridComp),            intent(INout) :: GC ! Gridded component
type (MAPL_MetaComp),                 pointer :: MAPLOBJ
integer,             optional, intent(  OUT) :: RC ! Return code
```

DESCRIPTION:

This is the recommended way of getting the opaque MAPL Generic state object from the gridded component (GC). It can be called at any time *after* `MAPL_GenericSetServices` has been called on GC. Note that you get a pointer to the object.

A.2.11 MAPL_Get

A.2.11.1

INTERFACE:

```
subroutine MAPL_Get             (STATE, IM, JM, LM, VERTDIM,               &
                                 NX, NY, NXO, NYO, LAYOUT,                 &
                                 GCNames,                                  &
                                 LONS, LATS, ORBIT, RUNALARM,              &
                                 IMPORTspec, EXPORTspec, INTERNALspec,     &
                                 INTERNAL_ESMF_STATE,                      &
                                 TILETYPES, TILEKIND,                      &
                                 TILELATS,TILELONS,TILEAREA,LOCSTREAM,     &
                                 EXCHANGEGRID,                             &
                                 CLOCK,                                    &
```

```
                                    NumInitPhases,                              &
                                    GCS, CCS, GIM, GEX, CF, HEARTBEAT, maplComm, RC )
```

ARGUMENTS:

```
type (MAPL_MetaComp),           intent(INOUT) :: STATE
type (ESMF_Alarm),    optional, intent(  OUT) :: RUNALARM
type (MAPL_SunOrbit), optional, intent(  OUT) :: ORBIT
integer,              optional, intent(  OUT) :: IM, JM, LM
integer,              optional, intent(  OUT) :: VERTDIM
integer,              optional, intent(  OUT) :: NX, NY, NXO, NYO
type (ESMF_DELayout), optional, intent(  OUT) :: LAYOUT
real, pointer,        optional                :: LONS(:,:)
real, pointer,        optional                :: LATS(:,:)
integer,              optional, intent(  OUT) :: RC     ! Error code:
character(len = ESMF_MAXSTR),optional, pointer  :: GCNames(:)
type (MAPL_VarSpec),  optional, pointer       :: IMPORTspec(:)
type (MAPL_VarSpec),  optional, pointer       :: EXPORTspec(:)
type (MAPL_VarSpec),  optional, pointer       :: INTERNALspec(:)
type (ESMF_State),    optional, intent(  OUT) :: INTERNAL_ESMF_STATE
integer,              optional, pointer       :: TILETYPES(:)
integer,              optional, pointer       :: TILEKIND(:)
real, pointer,        optional                :: TILELONS(:)
real, pointer,        optional                :: TILELATS(:)
real, pointer,        optional                :: TILEAREA(:)
type (MAPL_LocStream),optional, intent(  OUT) :: LOCSTREAM
type (MAPL_LocStream),optional, intent(  OUT) :: EXCHANGEGRID
type (ESMF_CLOCK)    ,optional, intent(  OUT) :: CLOCK
type (ESMF_GridComp), optional, pointer       :: GCS(:)
type (ESMF_CplComp),  optional, pointer       :: CCS(:,:)
type (ESMF_State),    optional, pointer       :: GIM(:)
type (ESMF_State),    optional, pointer       :: GEX(:)
real                 ,optional, intent(  OUT) :: HEARTBEAT
integer,              optional, intent(  OUT) :: NumInitPhases
type (ESMF_Config),   optional, intent(  OUT) :: CF
type (MAPL_Communicators), optional, intent(OUT) :: maplComm
```

DESCRIPTION:

This is the way of querying the opaque *MAPL_Generic* state object. The arguments are:

STATE The MAPL object to be queried.

IM Size of the first horizontal dimension (X) of local arrays.

JM Size of the second horizontal dimension (Y) of local arrays.

LM Size of the vertical dimension.

VERTDIM Position of the vertical dimension of 2 or higher dimensional arrays.

NX Size of the DE array dimension aligned with the first horizontal dimension of arrays

NY Size of the DE array dimension aligned with the second horizontal dimension of arrays

NX0, NY0 Coordinates of current DE.

LONS X coordinates of array locations. Currently longitude in radians.

LATS Y coordinates of array locations. Currently latitude in radians.

INTERNAL_ESMF_STATE The gridded component's INTERNAL state.

GCNames Names of the children.

GCS The child gridded components.

GIM The childrens' IMPORT states.

GEX The childrens' EXPORT states.

CCS Array of child-to-child couplers.

A.2.12 MAPL_Set

A.2.12.1

INTERFACE:

```
   subroutine MAPL_Set              (STATE, ORBIT, LM, RUNALARM, CHILDINIT, &
                                     LOCSTREAM, EXCHANGEGRID, CLOCK, NAME,  &
                                     CF, ConfigFile, maplComm, RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),            intent(INOUT) :: STATE
type (ESMF_Alarm),     optional, intent(IN   ) :: RUNALARM
type (MAPL_SunOrbit),  optional, intent(IN   ) :: ORBIT
integer,               optional, intent(IN   ) :: LM
logical,               optional, intent(IN   ) :: CHILDINIT
type (MAPL_LocStream), optional, intent(IN   ) :: LOCSTREAM
type (MAPL_LocStream), optional, intent(IN   ) :: EXCHANGEGRID
type (ESMF_Clock)    , optional, intent(IN   ) :: CLOCK
type (ESMF_Config)   , optional, intent(IN   ) :: CF
character(len = *)     , optional, intent(IN   ) :: NAME
character(len = *)     , optional, intent(IN   ) :: ConfigFile
type(MAPL_Communicators), optional, intent(IN) :: maplComm
integer,               optional, intent(  OUT) :: RC
```

A.2.12.2

INTERFACE:

```
subroutine MAPL_Set                    (GC, ORBIT, LM, RUNALARM, CHILDINIT, &
                                        LOCSTREAM, EXCHANGEGRID, CLOCK, RC)
```

ARGUMENTS:

```
type (ESMF_GridComp),            intent(INout) :: GC
type (ESMF_Alarm),     optional, intent(IN   ) :: RUNALARM
type (MAPL_SunOrbit),  optional, intent(IN   ) :: ORBIT
integer,               optional, intent(IN   ) :: LM
logical,               optional, intent(IN   ) :: CHILDINIT
type (MAPL_LocStream), optional, intent(IN   ) :: LOCSTREAM
type (MAPL_LocStream), optional, intent(IN   ) :: EXCHANGEGRID
type (ESMF_Clock)    , optional, intent(IN   ) :: CLOCK
integer,               optional, intent(  OUT) :: RC
```

A.2.13 MAPL_GenericRunCouplers

INTERFACE:

```
subroutine MAPL_GenericRunCouplers( STATE, CHILD, CLOCK, RC )
```

ARGUMENTS:

```
type (MAPL_MetaComp),    intent(INOUT) :: STATE
integer,                 intent(IN   ) :: CHILD  ! Child Id
type(ESMF_Clock),        intent(INOUT) :: CLOCK  ! The clock
integer, optional,       intent(  OUT) :: RC     ! Error code
```

A.2.14 MAPL_StatePrintSpecCSV

INTERFACE:

```
recursive subroutine MAPL_StatePrintSpecCSV(GC, printSpec, RC)
```

ARGUMENTS:

```
type(ESMF_GridComp),           intent(INOUT)  :: GC
integer,                       intent(IN   )  :: printSpec
integer,             optional, intent(  OUT)  :: RC
```

A.2.15 MAPL_AddChild

A.2.15.1 From Meta

INTERFACE:

```
recursive integer function MAPL_AddChild        (META, NAME, GRID, &
                                                 CONFIGFILE, SS, PARENTGC, &
                                                 petList, RC)
```

ARGUMENTS:

```
type(MAPL_MetaComp),           intent(INOUT) :: META
character(len = *),              intent(IN   ) :: NAME
type(ESMF_Grid),  optional,    intent(INout) :: GRID
character(len = *), optional,    intent(IN   ) :: CONFIGFILE
external                                       :: SS
type(ESMF_GridComp), optional, intent(IN   ) :: parentGC
integer,           optional  , intent(IN   ) :: petList(:)
integer,           optional  , intent(  OUT) :: rc
```

A.2.15.2 From gc

INTERFACE:

```
recursive integer function MAPL_AddChild      (GC, NAME, SS, petList, RC)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(INOUT) :: GC
character(len = *),    intent(IN   ) :: NAME
external                             :: SS
integer, optional  , intent(IN   ) :: petList(:)
integer, optional  , intent(  OUT) :: rc
```

A.2.16 MAPL_AddConnectivity

A.2.16.1 Rename

INTERFACE:

```
subroutine MAPL_AddConnectivity       ( GC, SRC_NAME, SRC_ID, &
                                        DST_NAME, DST_ID, RC  )
```

ARGUMENTS:

```
     type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
     character (len = *),              intent(IN   ) :: SRC_NAME !FROM_NAME =  = SHORT_NAME
     character (len = *),              intent(IN   ) :: DST_NAME !TO_NAME
     integer,                        intent(IN   ) :: SRC_ID !FROM_EXPORT
     integer,                        intent(IN   ) :: DST_ID !TO_IMPORT
     integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.16.2 Rename many

INTERFACE:

```
   subroutine MAPL_AddConnectivity           ( GC, SRC_NAME, SRC_ID, &
                                               DST_NAME, DST_ID, RC  )
```

ARGUMENTS:

```
     type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
     character (len = *),              intent(IN   ) :: SRC_NAME(:)
     character (len = *),              intent(IN   ) :: DST_NAME(:)
     integer,                        intent(IN   ) :: SRC_ID !FROM_EXPORT
     integer,                        intent(IN   ) :: DST_ID !TO_IMPORT
     integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.16.3 Many

INTERFACE:

```
   subroutine MAPL_AddConnectivity     ( GC, SHORT_NAME, SRC_ID, DST_ID, RC  )
```

ARGUMENTS:

```
type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
character (len = *)             , intent(IN   ) :: SHORT_NAME(:)
integer,                        intent(IN   ) :: SRC_ID
integer,                        intent(IN   ) :: DST_ID
integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.17 MAPL_TerminateImport

A.2.17.1 Do not connect

INTERFACE:

```
subroutine MAPL_TerminateImport ( GC, SHORT_NAME, CHILD, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
character (len = *)             , intent(IN   ) :: SHORT_NAME
integer,                        intent(IN   ) :: CHILD
integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.17.2 Do not connect many

INTERFACE:

```
subroutine MAPL_TerminateImport  ( GC, SHORT_NAME, CHILD, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
character (len = *)             , intent(IN   ) :: SHORT_NAME(:)
integer,                        intent(IN   ) :: CHILD
integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.17.3 Do not connect any import

INTERFACE:

```
subroutine MAPL_TerminateImport       ( GC, CHILD, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
integer,                        intent(IN   ) :: CHILD
integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.17.4 Terminate import all

INTERFACE:

```
subroutine MAPL_TerminateImport    ( GC, ALL, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp),            intent(INOUT) :: GC ! Gridded component
logical,                        intent(IN   ) :: ALL
integer,              optional, intent(  OUT) :: RC     ! Error code:
```

A.2.18 MAPL_TimerOn

A.2.18.1

INTERFACE:

```
subroutine MAPL_TimerOn            (STATE,NAME,RC)
```

ARGUMENTS:

```
  type (MAPL_MetaComp),        intent(INOUT) :: STATE
  character(len = *),            intent(IN   ) :: NAME
  integer, optional,           intent(  OUT) :: RC     ! Error code:
```

A.2.19 MAPL_TimerOff

A.2.19.1

INTERFACE:

```
subroutine MAPL_TimerOff           (STATE,NAME,RC)
```

ARGUMENTS:

```
  type (MAPL_MetaComp),        intent(INOUT) :: STATE
  character(len = *),            intent(IN   ) :: NAME
  integer, optional,           intent(  OUT) :: RC     ! Error code:
```

A.2.20 MAPL_TimerAdd

A.2.20.1

INTERFACE:

```
subroutine MAPL_TimerAdd           (GC, NAME, RC)
```

ARGUMENTS:

```
type (ESMF_GridComp),        intent(INOUT) :: GC
character(len = *),            intent(IN   ) :: NAME
integer, optional,           intent(  OUT) :: RC     ! Error code:
```

A.2.21 MAPL_GetResource

A.2.21.1 I41

INTERFACE:

```
subroutine MAPL_GetResource   (STATE,VALUE,LABEL,DEFAULT,RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),       intent(INOUT)    :: STATE
character(len = *),           intent(IN   )    :: LABEL
integer*4,                  intent(INOUT)    :: VALUE(:)
integer*4, optional,        intent(IN   )    :: DEFAULT(:)
integer  , optional,        intent(  OUT)    :: RC
```

A.2.21.2 I4

INTERFACE:

```
subroutine MAPL_GetResource  (STATE,VALUE,LABEL,DEFAULT,RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),       intent(INOUT)    :: STATE
character(len = *),           intent(IN   )    :: LABEL
integer*4,                  intent(INOUT)    :: VALUE
integer  , optional,        intent(IN   )    :: DEFAULT
integer  , optional,        intent(  OUT)    :: RC
```

A.2.21.3 I8

INTERFACE:

```
subroutine MAPL_GetResource  (STATE,VALUE,LABEL,DEFAULT,RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),            intent(INOUT)    :: STATE
character(len = *),                intent(IN   )    :: LABEL
integer(ESMF_KIND_I8),           intent(  OUT)    :: VALUE
integer(ESMF_KIND_I8), optional, intent(IN   )    :: DEFAULT
integer  , optional,             intent(  OUT)    :: RC
```

A.2.21.4 R4

INTERFACE:

```
subroutine MAPL_GetResource  (STATE,VALUE,LABEL,DEFAULT,RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),       intent(INOUT)    :: STATE
character(len = *),           intent(IN   )    :: LABEL
real*4,                     intent(INOUT)    :: VALUE
real     , optional,        intent(IN   )    :: DEFAULT
integer  , optional,        intent(  OUT)    :: RC
```

A.2.21.5 R8

INTERFACE:

```
subroutine MAPL_GetResource  (STATE,VALUE,LABEL,DEFAULT,RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),         intent(INOUT)    :: STATE
character(len = *),             intent(IN   )    :: LABEL
real(ESMF_KIND_R8),           intent(  OUT)    :: VALUE
real(ESMF_KIND_R8), optional, intent(IN   )    :: DEFAULT
integer  , optional,          intent(  OUT)    :: RC
```

A.2.21.6 C

INTERFACE:

```
subroutine MAPL_GetResource (STATE,VALUE,LABEL,DEFAULT,RC)
```

ARGUMENTS:

```
type (MAPL_MetaComp),       intent(INOUT)    :: STATE
character(len = *),           intent(IN   )    :: LABEL
character(len = *),           intent(INOUT)    :: VALUE
character(len = *), optional, intent(IN   )    :: DEFAULT
integer  , optional,        intent(  OUT)    :: RC
```

INTERFACE:

```
logical function MAPL_IsFieldAllocated(FIELD, RC)
```

ARGUMENTS:

```
type(ESMF_Field),    intent(INout) :: FIELD  ! Field
integer, optional,   intent(  OUT) :: RC     ! Error code:
```

DESCRIPTION:

Shortcut for checking that field is allocated

A.2.22 MAPL_ReadForcing

A.2.22.1 1

INTERFACE:

```
subroutine MAPL_ReadForcing (STATE,NAME,DATAFILE,CURRENTTIME,    &
                             FORCING,INIT_ONLY,ON_TILES,RC )
```

ARGUMENTS:

```
type (MAPL_MetaComp),     intent(INOUT)   :: STATE
character(len = *),         intent(IN   )   :: NAME
character(len = *),         intent(IN   )   :: DATAFILE
type (ESMF_Time),         intent(INout)   :: CURRENTTIME
real,                     intent(  OUT)   :: FORCING(:)
logical, optional,        intent(IN   )   :: INIT_ONLY
logical, optional,        intent(IN   )   :: ON_TILES
integer, optional,        intent(  OUT)   :: RC
```

A.2.22.2 2

INTERFACE:

```
subroutine MAPL_ReadForcing (STATE,NAME,DATAFILE,CURRENTTIME,    &
                             FORCING,INIT_ONLY,RC )
```

ARGUMENTS:

```
type (MAPL_MetaComp),     intent(INOUT)   :: STATE
character(len = *),         intent(IN   )   :: NAME
character(len = *),         intent(IN   )   :: DATAFILE
type (ESMF_Time),         intent(INout)   :: CURRENTTIME
real,                     intent(  OUT)   :: FORCING(:,:)
logical, optional,        intent(IN   )   :: INIT_ONLY
integer, optional,        intent(  OUT)   :: RC
```

A.3 Module MAPL_CFIO — CF Compliant I/O for ESMF

DESCRIPTION:

`MAPL_CFIO` provides *Climate and Forecast* (CF) compliant I/O methods for high level ESMF data types by using the CFIO Library. It currently includes read-write support for ESMF Bundles and States, and read-only support for ESMF Fields and Fortran arrays. The API consists of 4 basic methods:

- MAPL_CFIORead
- MAPL_CFIOCreate
- MAPL_CFIOWrite
- MAPL_CFIODestroy

Reading a file. When reading data from a CFIO compliant file, in the very least the user needs to specify the file name, the time, and an ESMF object to receive the data. There is no need to explicitly involve the `MAPL_CFIO` object in this case. For example, assuming that one has already defined an ESMF grid and clock, here is how to read a single time instance of all variables from a file into an ESMF Bundle:

```
bundle = ESMF_BundleCreate ( name='Mary', grid=grid )
call MAPL_CFIORead ( 'forecast_data.nc', clock, bundle )
```

This method will read all variables on file, doing any necessary (horizontal) regriding to the ESMF `grid` used to create the bundle, and allocating memory for each variable, as necessary. Currently, the file is open, read from and subsequently closed. `MAPL_CFIO` also provides methods to read single variables into a simple fortran arrays:

```
real :: ps(im,jm)
call MAPL_CFIORead ( 'surfp', 'forecast_data.nc', clock, grid, ps )
```

This method will read the variable named 'surfp' into the 2D Fortran array `ps`, performing any necessary (horizontal) interpolation to the destination `grid`. Consult the API reference below for several optional parameters to the MAPL_CFIORead() method, including the ability to select the variables to read and transparently perform time interpolation.

Writing to a file. For writing, a new file is created, written to, and explicitly closed. Assuming one has already defined an ESMF `bundle` and `clock` here is how to save a bundle to a new file:

```
type(MAPL_CFIO) :: mcfio
call MAPL_CFIOcreate  ( mcfio, 'climate_data', clock, bundle )
call MAPL_CFIOwrite   ( mcfio, clock )
call MAPL_CFIOdestroy ( mcfio )
```

Consult the API reference below for many optional parameters controling the behavior of these methods. As of this writing, a `MAPL_CFIOopen()` function to write to an already existing file has not been implemented.

File formats. `MAPL_CFIO` is designed to work with a variety of file formats, provided these files can be annotaded with the necessary CF metadata. The particularities of the specific file format is handled by the backend CFIO library. As of this writing the backend CFIO library supports self-describing formats such as NetCDF and HDF, and support for GrADS compatible binary files is in alpha testing. There are also plans to add support for GRIB versions 1, 2 or both.

Self-desbring (SDF) formats. The support for SDF formats is implemented in the backend CFIO library using the NetCDF Version 2 API. This API is currently supported by NetCDF versions 2 through 4 and HDF version 4. By selecting one of these libraries at build time it is possible to read/write several versions of NetCDF/HDF as summarized in the following table:

Library	Reads	Writes
HDF-4	NetCDF, HDF-4	HDF-4
NetCDF-2	NetCDF	NetCDF

```
NetCDF-3     NetCDF          NetCDF
NetCDF-4     NetCDF, HDF-5   NetCDF, HDF-5
```

NetCDF versions 2 and 3 can only read/write its own native `NetCDF` format. HDF version 4 offers some form of interoperability with NetCDF, but it can only write HDF-4 files. The new NetCDF version 4 is written on top of the HDF-5 library. This version of NetCDF still retains the ability of reading and writing its legacy `NetCDF` format, but advanced features such as compression is only available when writing in HDF-5 format. Beware that NetCDF-4 can only read the particular kind of HDF-5 files it writes; it cannot read generic HDF-5 files such has HDF-5 EOS. Because the standard HDF-5 library (without NetCDF-4) no longer supports the NetCDF 2 API (or the native HDF-4 API for that matter) it cannot be used with the SDF backend of the CFIO library. It is important to notice that because of conflicts in the API one cannot load more than one NetCDF or HDF library in one application.

API Design Issues. The `MAPL_CFIO` package is still under active development. The current state of the API was dictated by the features needed to build the GEOS-5 model at NASA/GSFC, and some asymmetry still remains in the API. In particular, the *read methods* utilize file names to specify the file object, while the *write methods* uses the `MAPL_CFIO` object much like a file handle. Both methods of access are valid and useful under different circunstances, and ought to be supported for both read and write operations. When using *file name* access mode the following should be possible:

```
call MAPL_CFIORead  ( 'forecast_data.nc', clock, bundle )
call MAPL_CFIOWrite ( 'new_file.nc', clock, bundle )
call MAPL_CFIOWrite ( 'existing_file.nc', clock, bundle, append=.true. )
```

In this case, the file is opened, read from/written to and closed on exit. For users desiring to keep files open in between operations a *file handle* mode should be provided for both read and write. Here is a typical use case for reading:

```
mcfio = MAPL_CFIOopen ( 'forecast_data.nc' )
call MAPL_CFIORead    ( mcfio, clock_now,   bundle )
...
call MAPL_CFIORead    ( mcfio, clock_later, bundle )
call MAPL_CFIOdestroy ( mcfio )
```

Future versions of `MAPL_CFIO` may support these features.

USES:

```
use ESMF
```

```
use MAPL_BaseMod
use MAPL_CommsMod
use MAPL_ConstantsMod
use ESMF_CFIOMod
use ESMF_CFIOUtilMod
use ESMF_CFIOFileMod
use MAPL_IOMod
use MAPL_HorzTransformMod
use ESMFL_Mod
use MAPL_ShmemMod
use MAPL_CFIOServerMod
```

PUBLIC MEMBER FUNCTIONS:

```
! MAPL-style names
! ----------------
public MAPL_CFIOCreate
public MAPL_CFIOSet
public MAPL_CFIOOpenWrite
public MAPL_CFIOCreateWrite
public MAPL_CFIOClose
public MAPL_CFIOWrite
public MAPL_CFIOWriteBundlePost
public MAPL_CFIOWriteBundleWait
public MAPL_CFIOWriteBundleWrite
public MAPL_CFIORead
public MAPL_CFIODestroy
public MAPL_GetCurrentFile
public MAPL_CFIOIsCreated
public MAPL_CFIOGetFilename
public MAPL_CFIOGetTimeString
public MAPL_CFIOStartAsyncColl
public MAPL_CFIOBcastIONode

! ESMF-style names
! ----------------
public ESMF_ioRead     ! another name for MAPL_CFIORead
public ESMF_ioCreate   ! another name for MAPL_CFIOCreate
public ESMF_ioWrite    ! another name for MAPL_CFIOWrite
public ESMF_ioDestroy  ! another name for MAPL_CFIODestroy
```

PUBLIC TYPES:

```
   public MAPL_CFIO
```

CODE:

```
  type MAPL_CFIO
     private
     logical                     :: CREATED = .false.
     character(len = ESMF_MAXSTR) :: NAME
     character(len = ESMF_MAXSTR) :: fNAME
     character(len = ESMF_MAXSTR) :: format
     character(len = ESMF_MAXSTR) :: expid
     type(ESMF_CFIO)             :: CFIO
     integer                     :: XYOFFSET
     real                        :: VSCALE
     type(ESMF_TIMEINTERVAL)     :: OFFSET
     type(ESMF_CLOCK)            :: CLOCK
     type(ESMF_FIELDBUNDLE)      :: BUNDLE
     type(ESMF_GridComp)         :: GC
     type(ESMF_Grid)             :: Grid
     integer                     :: Root = MAPL_Root
     integer                     :: PartSize = 1
     integer                     :: myPE
     integer                     :: numcores
     integer                     :: comm
     integer                     :: Order = -1
     integer                     :: Nbits = 1000
     integer                     :: IM, JM, LM
     integer, pointer            :: SUBSET(:)  = > null()
     integer, pointer            :: VarDims(:) = >null()
     integer, pointer            :: VarType(:) = >null()
     integer, pointer            :: needVar(:) = >null()
     integer, pointer            :: pairList(:) = >null()
     character(len = ESMF_MAXSTR), &
                         pointer :: vectorList(:,:) = >null()
     logical                     :: Vinterp
     real                        :: pow = 0.0
     character(len = ESMF_MAXSTR) :: Vvar
     character(len = 3           ) :: Func
     character(len = ESMF_MAXSTR), &
```

```
                    pointer :: VarName(:) = >null()
    integer, pointer           :: Krank(:) = >null()
    real,    pointer           :: levs(:)
    type(MAPL_CommRequest), &
                    pointer :: reqs(:) = >null()
    type(MAPL_HorzTransform)   :: Trans
    logical                    :: async
    integer                    :: AsyncWorkRank
    integer                    :: globalComm
  end type MAPL_CFIO
```

A.3.1 MAPL_CFIOCreate — Creates a MAPL CFIO Object

A.3.1.1 Creates MAPL CFIO Object from a Bundle

INTERFACE:

```
   subroutine MAPL_CFIOCreate           ( MCFIO, NAME, CLOCK, BUNDLE, OFFSET,      &
                                          RESOLUTION, SUBSET, CHUNKSIZE, FREQUENCY, LEVELS, DES
                                          XYOFFSET, VCOORD, VUNIT, VSCALE,         &
                                          SOURCE, INSTITUTION, COMMENT, CONTACT,   &
                                          FORMAT, EXPID, DEFLATE, GC,  ORDER, &
                                          NumCores, nbits, TM, Conservative,  &
                                          Async, VectorList, RC )
```

ARGUMENTS:

```
    type(MAPL_CFIO),             intent(OUT) :: MCFIO
    character(LEN = *),            intent(IN)  :: NAME
    type(ESMF_FIELDBUNDLE),      intent(INout)  :: BUNDLE
    type(ESMF_CLOCK),            intent(INout):: CLOCK
    type(ESMF_TIMEINTERVAL), &
                     optional,   intent(INout):: OFFSET
    integer,         optional,   pointer     :: RESOLUTION(:)
    real,            optional,   pointer     :: SUBSET(:)
    integer,         optional,   pointer     :: CHUNKSIZE(:)
    integer,         optional,   intent(IN)  :: FREQUENCY
    real,            optional,   pointer     :: LEVELS(:)
    character(LEN = *),optional,   intent(IN)  :: DESCR
```

```
integer,         optional,   intent(IN)  :: XYOFFSET
real,            optional,   intent(IN)  :: VSCALE
integer,         optional,   intent(IN)  :: DEFLATE
character(len = *),optional,   intent(IN)  :: VUNIT
character(len = *),optional,   intent(IN)  :: VCOORD
character(len = *),optional,   intent(IN)  :: source
character(len = *),optional,   intent(IN)  :: institution
character(len = *),optional,   intent(IN)  :: comment
character(len = *),optional,   intent(IN)  :: contact
character(len = *),optional,   intent(IN)  :: format
character(len = *),optional,   intent(IN)  :: EXPID
integer,         optional,   intent(IN)  :: Conservative
type(ESMF_GridComp),optional,intent(IN)  :: GC
integer,         optional,   intent(IN)  :: Order
integer,         optional,   intent(IN)  :: Nbits
integer,         optional,   intent(IN)  :: NumCores
integer,         optional,   intent(IN)  :: TM
logical,         optional,   intent(IN)  :: Async
character(len = *),  pointer,&
                 optional,   intent(IN)  :: vectorList(:,:)
integer,         optional,   intent(OUT) :: RC
```

DESCRIPTION:

Creates a MAPL_CFIO object from a Bundle. The MAPL_CFIO objects is opaque and its properties can only be set by this method at creation. Currently, its properties cannot be queried. The object is used only as a handle in write operations. It is not needed for reading.

Its non-optional arguments associate a `NAME`, an ESMF `BUNDLE`, and a `CLOCK` with the object. An ESMF TimeInterval `OFFSET` is an optional argument that sets an offset between the time on the clock when eriting and the time stamp used for the data (defaults to no offset).

The `format` optional argument determines whether the write will use the linked self-describing format (SDF) library (HDF or netcdf) or write GrADS readable flat files. Currently only the SDF library option is supported.

The remaining (optional) arguments are especialized and used primarily to support MAPL_History, or to provide documentation in the form of character strings that will be placed in corresponding attributes in the SDF file. REVISION HISTORY:

```
19Apr2007 Todling  - Added ability to write out ak/bk
                   - Added experiment ID as optional argument
```

A.3.1.2 Creates MAPL CFIO Object from a State

INTERFACE:

```
subroutine MAPL_CFIOCreate          ( MCFIO, NAME, CLOCK, STATE, OFFSET,  &
                                      RESOLUTION, SUBSET, CHUNKSIZE, FREQUENCY, &
                                      LEVELS, DESCR, BUNDLE, &
                                      XYOFFSET, VCOORD, VUNIT, VSCALE,   &
                                      SOURCE, INSTITUTION, COMMENT, CONTACT, &
                                      FORMAT, EXPID, DEFLATE, GC,  ORDER, &
                                      NumCores, nbits, TM, Conservative,  RC )
```

ARGUMENTS:

```
type(MAPL_CFIO),             intent(OUT) :: MCFIO
character(LEN = *),            intent(IN)  :: NAME
type(ESMF_State),            intent(INout)  :: STATE
type(ESMF_Clock),            intent(INOUT)  :: CLOCK
type(ESMF_FieldBundle), optional,  pointer    :: BUNDLE
type(ESMF_TimeInterval), &
                optional,   intent(INOUT):: OFFSET
integer,         optional,   pointer     :: RESOLUTION(:)
real,            optional,   pointer     :: SUBSET(:)
integer,         optional,   pointer     :: CHUNKSIZE(:)
integer,         optional,   intent(IN)  :: FREQUENCY
real,            optional,   pointer     :: LEVELS(:)
character(LEN = *),optional,   intent(IN)  :: DESCR
real,            optional,   intent(IN)  :: VSCALE
character(len = *),optional,   intent(IN)  :: VUNIT
character(len = *),optional,   intent(IN)  :: VCOORD
integer,         optional,   intent(IN)  :: XYOFFSET
character(len = *),optional,   intent(IN)  :: source
character(len = *),optional,   intent(IN)  :: institution
character(len = *),optional,   intent(IN)  :: comment
character(len = *),optional,   intent(IN)  :: contact
character(len = *),optional,   intent(IN)  :: format
character(len = *),optional,   intent(IN)  :: EXPID
integer,         optional,   intent(IN)  :: DEFLATE
type(ESMF_GridComp),optional,intent(IN)  :: GC
```

```
integer,         optional,   intent(IN)  :: Order
integer,         optional,   intent(IN)  :: Nbits
integer,         optional,   intent(IN)  :: NumCores
integer,         optional,   intent(IN)  :: TM
integer,         optional,   intent(IN)  :: CONSERVATIVE
integer, optional,           intent(OUT) :: RC
```

DESCRIPTION:

Creates a MAPL_CFIO object from a State. States are written by "serializing" all Fields in them, whether they are directly in the State or are contained within a hierarchy of embedded Bundles and States, into a single Bundle.

The Method optionally returns a pointer to the serialized ESMF Bundle, but this is not needed for MAPL_Write operations. Otherwise arguments are the same as for CreateFromBundle.

Its non-optional arguments associate a `NAME`, an ESMF `BUNDLE`, and a `CLOCK` with the object. An ESMF TimeInterval `OFFSET` is an optional argument that sets an offset between the time on the clock when eriting and the time stamp used for the data (defaults to no offset).

The `format` optional argument determines whether the write will use the linked self-describing format (SDF) library (HDF or netcdf) or write GrADS readable flat files. Currently only the SDF library option is supported.

The remaining (optional) arguments are especialized and used primarily to support MAPL_History, or to provide documentation in the form of character strings that will be placed in corresponding attributes in the SDF file. REVISION HISTORY:

```
12Jun2007 Todling  Added EXPID as opt argument
```

A.3.2 MAPL_CFIOWrite — Writing Methods

A.3.2.1 Writes an ESMF Bundle

INTERFACE:

```
subroutine MAPL_CFIOWrite      Post( MCFIO, RC )
```

ARGUMENTS:

```
type(MAPL_CFIO  ),               intent(INOUT) :: MCFIO
integer,               optional, intent(  OUT) :: RC
```

DESCRIPTION:

Writes an ESMF Bundle to a File. Only the MAPL_CFIO object is a required argument as pointers to the actual data to be written is recorded in it during creation.

`CLOCK, BUNDLE` can be used to override the choice made at creation, but this is of dubious value, particularly for `BUNDLE` since it must be excatly conformant with the creation `BUNDLE`. `NBITS` if the number of bits of the mantissa to retain. This is used to write files with degraded precision, which can then be compressed with standard utilities. The default is no degradation of precision.

A note about compression. NetCDF-4, HDF-4 and HDF-5 all support transparent internal GZIP compression of the data being written. However, very little is gained by compressing float point fields from earth system models. Compression yields can be greatly increased by setting to zero bits in the mantissa of float numbers. On average 50% compression can be achieved, while preserving a meaningful accuracy in the fields. Unlike classical CF compression by means of `scale_factor` and `add_offset` attributes, internal GZIP compression requires no special handling by the users of the data. In fact, they do not even need to know that the data is compressed! At this point, MAPL_CFIO does not activate this GZIP compression feature in the files being written, but the resulting precision degredaded files can be compressed offline with the HDF-4 `hrepack` utility.

A.3.3 MAPL_CFIOWrite — Writing Methods

A.3.3.1 Writes an ESMF Bundle

INTERFACE:

```
subroutine MAPL_CFIOWrite      ( MCFIO, CLOCK, Bundle, &
                                 VERBOSE, NBITS, RC    )
```

ARGUMENTS:

```
type(MAPL_CFIO  ),                intent(INOUT) :: MCFIO
```

```
type(ESMF_CLOCK),       optional,  intent(INOUT) :: CLOCK
type(ESMF_FIELDBUNDLE), optional,  intent(INout) :: BUNDLE
logical,                optional,  intent(IN   ) :: VERBOSE
integer,                optional,  intent(IN   ) :: NBITS
integer,                optional,  intent(  OUT) :: RC
```

DESCRIPTION:

Writes an ESMF Bundle to a File. Only the MAPL_CFIO object is a required argument as pointers to the actual data to be written is recorded in it during creation.

`CLOCK, BUNDLE` can be used to override the choice made at creation, but this is of dubious value, particularly for `BUNDLE` since it must be excatly conformant with the creation `BUNDLE`. `NBITS` if the number of bits of the mantissa to retain. This is used to write files with degraded precision, which can then be compressed with standard utilities. The default is no degradation of precision.

A note about compression. NetCDF-4, HDF-4 and HDF-5 all support transparent internal GZIP compression of the data being written. However, very little is gained by compressing float point fields from earth system models. Compression yields can be greatly increased by setting to zero bits in the mantissa of float numbers. On average 50% compression can be achieved, while preserving a meaningful accuracy in the fields. Unlike classical CF compression by means of `scale_factor` and `add_offset` attributes, internal GZIP compression requires no special handling by the users of the data. In fact, they do not even need to know that the data is compressed! At this point, MAPL_CFIO does not activate this GZIP compression feature in the files being written, but the resulting precision degredaded files can be compressed offline with the HDF-4 `hrepack` utility.

A.3.3.2 Writes an ESMF State

INTERFACE:

```
subroutine MAPL_CFIOWrite      ( MCFIO, CLOCK, State, &
                                 VERBOSE, NBITS, RC   )
```

ARGUMENTS:

```
type(MAPL_CFIO),            intent(INOUT) :: MCFIO
type(ESMF_State),           intent(INout) :: STATE
```

```
type(ESMF_CLOCK),            intent(INOUT) :: CLOCK
integer, optional,           intent(  OUT) :: RC
logical, optional,           intent(  IN)  :: VERBOSE
integer, optional,           intent(  IN)  :: NBITS
```

DESCRIPTION:

Serializes an ESMF state into a Bundle and writes it to a file. Only the MAPL_CFIO object is a required argument as pointers to the actual data to be written is recorded in it during creation.

`CLOCK, BUNDLE` can be used to override the choice made at creation, but this is of dubious value, particularly for `BUNDLE` since it must be excatly conformant with the creation `BUNDLE`. `NBITS` if the number of bits of the mantissa to retain. This is used to write files with degraded precision, which can then be compressed with standard utilities. The default is no degradation of precision.

A note about compression. NetCDF-4, HDF-4 and HDF-5 all support transparent internal GZIP compression of the data being written. However, very little is gained by compressing float point fields from earth system models. Compression yields can be greatly increased by setting to zero bits in the mantissa of float numbers. On average 50% compression can be achieved, while preserving a meaningful accuracy in the fields. Unlike classical CF compression by means of `scale_factor` and `add_offset` attributes, internal GZIP compression requires no special handling by the users of the data. In fact, they do not even need to know that the data is compressed! At this point, MAPL_CFIO does not activate this GZIP compression feature in the files being written, but the resulting precision degredaded files can be compressed offline with the HDF-4 `hrepack` utility.

A.3.4 MAPL_CFIORead — Reading Methods

A.3.4.1 Reads an ESMF Bundle

INTERFACE:

```
subroutine MAPL_CFIORead       ( FILETMPL, TIME, BUNDLE, NOREAD, RC, &
                                 VERBOSE, FORCE_REGRID, ONLY_VARS,   &
                                 TIME_IS_CYCLIC, TIME_INTERP, EXPID )
```

ARGUMENTS:

```
character(len = *),            intent(IN   ) :: FILETMPL
type(ESMF_TIME),             intent(INout) :: TIME
type(ESMF_FIELDBUNDLE),           intent(INOUT) :: BUNDLE
logical, optional,           intent(IN   ) :: NOREAD
integer, optional,           intent(  OUT) :: RC
logical, optional,           intent(IN)    :: VERBOSE
logical, optional,           intent(IN)    :: FORCE_REGRID
logical, optional,           intent(IN)    :: TIME_IS_CYCLIC
logical, optional,           intent(IN)    :: TIME_INTERP
character(len = *), optional,  intent(IN)    :: ONLY_VARS
character(len = *), optional,  intent(IN)    :: EXPID
```

DESCRIPTION:

Reads an ESMF Bundle from a file on a given time. The file is open, read from, and closed on exit. The arguments are:

FILETMPL A GrADS-style file name template. In its simplest form is the full path name for the file to be read. However, it can contain the following tokens which will be expanded from the current time in *TIME*:

%y4 4 digits for year

%m2 2 digits for month, to expand to 01, 02, .., 12

%m3 3 digits for month, to expand to jan, feb, mar, ..., dec

%d2 2 digits for day

%h2 2 digits for hour

%n2 2 digits for minutes

Example: if FILETMPL = "forecast.%y4-%m2-%d2_says it is 18Z on 05 February 2007, the template will expand in the following file name: "forecast.2007-02-05_18Z.nc4"

TIME The ESMF time to read from the file

BUNDLE An ESMF Bundle to read the data in. When the Bundle is empty one field is added for each variable present in the input file, and the necessary memory allocated according to the ESMF grid present in the Bundle.

[**NOREAD**] If .TRUE., no data is actually read into the Bundle. This is useful to define a Bundle with the same variables as presented in the file, which in turn can be used to created a MAPL_CFIO object for writing.

[**RC**] Error return code; set to ESMF_SUCCESS if all is well.

[**VERBOSE**] If .TRUE., prints progress messages to STDOUT; useful for debugging.

[**FORCE_REGRID**] Obsolete; kept for backward compatibility but has no effect.

[**TIME_IS_CYCLIC**] If .TRUE. it says that the input file is periodic in time. Useful for reading climatological files. For example, if the input file has 12 monthly means from January to December of 2001, setting this option to .TRUE. allows one to read this data for any other year. See note below regarding issues with reading monthly mean data.

[**TIME_INTERP**] If .TRUE., the input file does not have to coincide with the actual times on file. In such cases, the data for the bracketing times are read and the data is properly interpolated in time. The input time, though, need to be within the range of times present on file (unless `TIME_IS_CYCLIC` is specified).

[**ONLY_VARS**] A list of comma separated vafriables to be read from the file. By default, all variables are read from the file. This option allows one to read a subset of vafriables. Example: ONLY_VARS = "u,v,ps".

A note about storing monthly climatological data. As per the CF conventions, month is not a well defined unit of time, as the time step is not constant throughout the year. When storing 12 months of climatological data one way around it is to use an average number of hours: use 732 or 730 hours depending on whether the year recorded in the file is a leap-year or not.

DESIGN ISSUES:

The input argument `TIME` should be replaced with `CLOCK` for consistency with the rest of the API. One should also provide an interface involving the MAPL CFIO object.

A.3.4.2 Reads an ESMF State

INTERFACE:

```
subroutine MAPL_CFIORead      ( FILETMPL, TIME, STATE, NOREAD, RC, &
                                VERBOSE, FORCE_REGRID, ONLY_VARS,   &
                                TIME_IS_CYCLIC, TIME_INTERP )
```

ARGUMENTS:

```
character(len = *),            intent(IN   ) :: FILETMPL
type(ESMF_TIME),              intent(INout) :: TIME
type(ESMF_STATE),             intent(INOUT) :: STATE
```

```
logical, optional,           intent(IN   ) :: NOREAD
integer, optional,           intent(  OUT) :: RC
logical, optional,           intent(  IN)  :: VERBOSE
logical, optional,           intent(IN)    :: FORCE_REGRID ! obsolete
logical, optional,           intent(IN)    :: TIME_IS_CYCLIC
logical, optional,           intent(IN)    :: TIME_INTERP
character(len = *), optional,  intent(IN   ) :: ONLY_VARS ! comma separated,
                                                          ! no spaces
```

DESCRIPTION:

Serializes an ESMF state into a Bundle and reads its content from a file. The file is open, read from, and closed on exit. The arguments are:

FILETMPL A GrADS-style file name template. In its simplest form is the full path name for the file to be read. However, it can contain the following tokens which will be expanded from the current time in *TIME*:

%y4 4 digits for year

%m2 2 digits for month, to expand to 01, 02, .., 12

%m3 3 digits for month, to expand to jan, feb, mar, ..., dec

%d2 2 digits for day

%h2 2 digits for hour

%n2 2 digits for minutes

Example: if FILETMPL = "forecast.%y4-%m2-%d2_says it is 18Z on 05 February 2007, the template will expand in the following file name: "forecast.2007-02-05_18Z.nc4"

TIME The ESMF time to read from the file

STATE An ESMF State to read the data in. Usually used in conjubction with ONLY_VARS.

[**NOREAD**] If .TRUE., no data is actually read into the Bundle. This is useful to define a Bundle with the same variables as presented in the file, which in turn can be used to created a MAPL_CFIO object for writing.

[**RC**] Error return code; set to ESMF_SUCCESS if all is well.

[**VERBOSE**] If .TRUE., prints progress messages to STDOUT; useful for debugging.

[**FORCE_REGRID**] Obsolete; kept for backward compatibility but has no effect.

[**TIME_IS_CYCLIC**] If .TRUE. it says that the input file is periodic in time. Useful for reading climatological files. For example, if the input file has 12 monthly means from January to December of 2001, setting this option to .TRUE. allows one to read this data for any other year. See note below regarding issues with reading monthly mean data.

[**TIME_INTERP**] If .TRUE., the input file does not have to coincide with the actual times on file. In such cases, the data for the bracketing times are read and the data is properly interpolated in time. The input time, though, need to be within the range of times present on file (unless `TIME_IS_CYCLIC` is specified).

[**ONLY_VARS**] A list of comma separated vafriables to be read from the file. By default, all variables are read from the file. This option allows one to read a subset of vafriables. Example: ONLY_VARS = "u,v,ps".

DESIGN ISSUES:

The input argument `TIME` should be replaced with `CLOCK` for consistency with the rest of the API. One should also provide an interface involving the MAPL CFIO object.

A.3.4.3 Reads an ESMF Field

INTERFACE:

```
subroutine MAPL_CFIORead          ( VARN, FILETMPL, TIME,       FIELD, RC, &
                                    VERBOSE, FORCE_REGRID, TIME_IS_CYCLIC, &
                                    TIME_INTERP )
```

ARGUMENTS:

```
character(len = *),            intent(IN   ) :: VARN       ! Variable name
character(len = *),            intent(IN   ) :: FILETMPL   ! File name
type(ESMF_TIME),             intent(INout) :: TIME
type(ESMF_FIELD),            intent(INout) :: FIELD
integer, optional,           intent(  OUT) :: RC
logical, optional,           intent(  IN)  :: VERBOSE
logical, optional,           intent(IN)    :: FORCE_REGRID
logical, optional,           intent(IN)    :: TIME_IS_CYCLIC
logical, optional,           intent(IN)    :: TIME_INTERP
```

DESCRIPTION:

Reads a variable from a file and stores it on an ESMF Field. The file is open, read from, and closed on exit. The arguments are:

VARN The variable name.

FILETMPL A GrADS-style file name template. In its simplest form is the full path name for the file to be read. However, it can contain the following tokens which will be expanded from the current time in *TIME*:

%y4 4 digits for year

%m2 2 digits for month, to expand to 01, 02, .., 12

%m3 3 digits for month, to expand to jan, feb, mar, ..., dec

%d2 2 digits for day

%h2 2 digits for hour

%n2 2 digits for minutes

Example: if FILETMPL = "forecast.%y4-%m2-%d2_says it is 18Z on 05 February 2007, the template will expand in the following file name: "forecast.2007-02-05_18Z.nc4"

TIME The ESMF time to read from the file

[**RC**] Error return code; set to ESMF_SUCCESS if all is well.

[**VERBOSE**] If .TRUE., prints progress messages to STDOUT; useful for debugging.

[**FORCE_REGRID**] Obsolete; kept for backward compatibility but has no effect.

[**TIME_IS_CYCLIC**] If .TRUE. it says that the input file is periodic in time. Useful for reading climatological files. For example, if the input file has 12 monthly means from January to December of 2001, setting this option to .TRUE. allows one to read this data for any other year. See note below regarding issues with reading monthly mean data.

[**TIME_INTERP**] If .TRUE., the input file does not have to coincide with the actual times on file. In such cases, the data for the bracketing times are read and the data is properly interpolated in time. The input time, though, need to be within the range of times present on file (unless `TIME_IS_CYCLIC` is specified).

[**ONLY_VARS**] A list of comma separated vafriables to be read from the file. By default, all variables are read from the file. This option allows one to read a subset of vafriables. Example: ONLY_VARS = "u,v,ps".

DESIGN ISSUES:

The input argument `TIME` should be replaced with `CLOCK` for consistency with the rest of the API. The input `GRID` is not necessary as it can be found inside the field. One should also provide an interface involving the MAPL CFIO object.

A.3.4.4 Reads a 3D Fortran Array

INTERFACE:

```
subroutine MAPL_CFIORead        ( VARN, FILETMPL, TIME, GRID, farrayPtr, RC, &
                                  VERBOSE, FORCE_REGRID, TIME_IS_CYCLIC,  &
                                  TIME_INTERP )
```

ARGUMENTS:

```
character(len = *),            intent(IN   ) :: VARN       ! Variable name
character(len = *),            intent(IN   ) :: FILETMPL   ! File name
type(ESMF_TIME),             intent(INout) :: TIME
type(ESMF_GRID),             intent(IN   ) :: GRID
real, pointer                              :: farrayPtr(:,:,:)
integer, optional,           intent(  OUT) :: RC
logical, optional,           intent(  IN)  :: VERBOSE
logical, optional,           intent(IN)    :: FORCE_REGRID
logical, optional,           intent(IN)    :: TIME_IS_CYCLIC
logical, optional,           intent(IN)    :: TIME_INTERP
```

DESCRIPTION:

Reads a variable from a file and stores it on an 3D Fortrran array. The file is open, read from, and closed on exit. The arguments are:

VARN The variable name.

FILETMPL A GrADS-style file name template. In its simplest form is the full path name for the file to be read. However, it can contain the following tokens which will be expanded from the current time in *TIME*:

%y4 4 digits for year

%m2 2 digits for month, to expand to 01, 02, .., 12

%m3 3 digits for month, to expand to jan, feb, mar, ..., dec

%d2 2 digits for day

%h2 2 digits for hour

%n2 2 digits for minutes

Example: if FILETMPL = "forecast.%y4-%m2-%d2_says it is 18Z on 05 February 2007, the template will expand in the following file name: "forecast.2007-02-05_18Z.nc4"

TIME The ESMF time to read from the file

GRID The ESMF grid associated with the Field. The data will be (horizontally) interpolated to this grid if necessary.

[**RC**] Error return code; set to ESMF_SUCCESS if all is well.

[**VERBOSE**] If .TRUE., prints progress messages to STDOUT; useful for debugging.

[**FORCE_REGRID**] Obsolete; kept for backward compatibility but has no effect.

[**TIME_IS_CYCLIC**] If .TRUE. it says that the input file is periodic in time. Useful for reading climatological files. For example, if the input file has 12 monthly means from January to December of 2001, setting this option to .TRUE. allows one to read this data for any other year. See note below regarding issues with reading monthly mean data.

[**TIME_INTERP**] If .TRUE., the input file does not have to coincide with the actual times on file. In such cases, the data for the bracketing times are read and the data is properly interpolated in time. The input time, though, need to be within the range of times present on file (unless `TIME_IS_CYCLIC` is specified).

[**ONLY_VARS**] A list of comma separated vafriables to be read from the file. By default, all variables are read from the file. This option allows one to read a subset of vafriables. Example: ONLY_VARS = "u,v,ps".

DESIGN ISSUES:

The input argument `TIME` should be replaced with `CLOCK` for consistency with the rest of the API. One should also provide an interface involving the MAPL CFIO object.

A.3.4.5 Reads a 2D Fortran Array

INTERFACE:

```
subroutine MAPL_CFIORead        ( VARN, FILETMPL, TIME, GRID, farrayPtr, RC, &
                                  VERBOSE, FORCE_REGRID, TIME_IS_CYCLIC,  &
                                  TIME_INTERP )
```

ARGUMENTS:

```
character(len = *),            intent(IN)  :: VARN       ! Variable name
```

```
character(len = *),           intent(IN)  :: FILETMPL   ! File name
type(ESMF_TIME),            intent(INout)  :: TIME
type(ESMF_GRID),            intent(IN)  :: GRID
real, pointer                           :: farrayPtr(:,:)
integer, optional,          intent(OUT) :: RC
logical, optional,          intent(IN)  :: VERBOSE
logical, optional,          intent(IN)  :: FORCE_REGRID
logical, optional,          intent(IN)  :: TIME_IS_CYCLIC
logical, optional,          intent(IN)  :: TIME_INTERP
```

DESCRIPTION:

Reads a variable from a file and stores it on an 3D Fortrran array. The file is open, read from, and closed on exit. The arguments are:

VARN The variable name.

FILETMPL A GrADS-style file name template. In its simplest form is the full path name for the file to be read. However, it can contain the following tokens which will be expanded from the current time in *TIME*:

%y4 4 digits for year

%m2 2 digits for month, to expand to 01, 02, .., 12

%m3 3 digits for month, to expand to jan, feb, mar, ..., dec

%d2 2 digits for day

%h2 2 digits for hour

%n2 2 digits for minutes

Example: if FILETMPL = "forecast.%y4-%m2-%d2_says it is 18Z on 05 February 2007, the template will expand in the following file name: "forecast.2007-02-05_18Z.nc4"

TIME The ESMF time to read from the file

GRID The ESMF grid associated with the Field. The data will be (horizontally) interpolated to this grid if necessary.

[**RC**] Error return code; set to ESMF_SUCCESS if all is well.

[**VERBOSE**] If .TRUE., prints progress messages to STDOUT; useful for debugging.

[**FORCE_REGRID**] Obsolete; kept for backward compatibility but has no effect.

[**TIME_IS_CYCLIC**] If .TRUE. it says that the input file is periodic in time. Useful for reading climatological files. For example, if the input file has 12 monthly means from January to December of 2001, setting this option to .TRUE. allows one to read this data for any other year. See note below regarding issues with reading monthly mean data.

[**TIME_INTERP**] If .TRUE., the input file does not have to coincide with the actual times on file. In such cases, the data for the bracketing times are read and the data is properly interpolated in time. The input time, though, need to be within the range of times present on file (unless `TIME_IS_CYCLIC` is specified).

[**ONLY_VARS**] A list of comma separated vafriables to be read from the file. By default, all variables are read from the file. This option allows one to read a subset of vafriables. Example: ONLY_VARS = "u,v,ps".

DESIGN ISSUES:

The input argument `TIME` should be replaced with `CLOCK` for consistency with the rest of the API. One should also provide an interface involving the MAPL CFIO object.

A.3.5 MAPL_CFIODestroy — Destroys MAPL CFIO Object

INTERFACE:

```
subroutine MAPL_CFIODestroy( MCFIO, RC )
```

ARGUMENTS:

```
type(MAPL_CFIO),             intent(INOUT) :: MCFIO
integer, optional,           intent(  OUT) :: RC
```

DESCRIPTION:

Destroys a MAPL CFIO object. It closes any file associated with it and deallocates memory.

A.3.6 MAPL_CFIOClose — Close file in MAPL CFIO Object

INTERFACE:

```
subroutine MAPL_CFIOClose( MCFIO, filename, RC )
```

ARGUMENTS:

```
type(MAPL_CFIO),              intent(INOUT) :: MCFIO
character(len = *), optional,  intent(IN   ) :: filename
integer, optional,            intent(  OUT) :: RC
```

DESCRIPTION:

Not a full destroy; only closes the file.

A.4 Module MAPL_LocStreamMod – Manipulate location streams

USES:

```
use ESMF
use ESMFL_Mod
use MAPL_BaseMod
use MAPL_ConstantsMod
use MAPL_IOMod
use MAPL_CommsMod
use MAPL_HashMod
use MAPL_ShmemMod
```

PUBLIC MEMBER FUNCTIONS:

```
public MAPL_LocStreamCreate
public MAPL_LocStreamAdjustNsubtiles
public MAPL_LocStreamTransform
public MAPL_LocStreamIsAssociated
public MAPL_LocStreamXformIsAssociated
public MAPL_LocStreamGet
public MAPL_LocStreamCreateXform
```

```
public MAPL_LocStreamFracArea
public MAPL_GridCoordAdjust
public MAPL_LocStreamTileWeight


INCLUDE 'mpif.h'
```

PUBLIC TYPES:

```
type, public :: MAPL_LocStream
  type(MAPL_LocStreamType), pointer :: Ptr = >null()
end type MAPL_LocStream

type, public :: MAPL_LocStreamXform
  type(MAPL_LocStreamXformType), pointer :: Ptr = >null()
end type MAPL_LocStreamXform
```

A.4.1 MAPL_LocStreamCreate

A.4.1.1 Create from file

INTERFACE:

```
  subroutine MAPL_LocStreamCreate        (LocStream, LAYOUT, FILENAME, NAME, MASK, GRID, NewG
```

ARGUMENTS:

```
    type(MAPL_LocStream),                intent(  OUT) :: LocStream
    type(ESMF_DELayout),                 intent(IN   ) :: LAYOUT
    character(len = *),                    intent(IN   ) :: FILENAME
    character(len = *),                    intent(IN   ) :: NAME
    integer,                   optional, intent(IN   ) :: MASK(:)
    type(ESMF_Grid), optional,           intent(INout) :: GRID
    logical,                   optional, intent(IN   ) :: NewGridNames
    integer,                   optional, intent(  OUT) :: RC
```

DESCRIPTION:

Creates a location stream from a file. This does not decompose the location stream; so the global stream is described in each processor. The stream can be decomposed later in various ways. Currently we only decompose it by "attaching" it to a decomposed grid.

A.4.1.2 Create from stream

INTERFACE:

```
subroutine MAPL_LocStreamCreate          (LocStreamOut, LocStreamIn, NAME, MASK, RC)
```

ARGUMENTS:

```
type(MAPL_LocStream),                intent(  OUT) :: LocStreamOut
type(MAPL_LocStream),                intent(IN   ) :: LocStreamIn
character(len = *),                    intent(IN   ) :: NAME
integer,                   optional, intent(IN   ) :: MASK(:)
integer,                   optional, intent(  OUT) :: RC
```

DESCRIPTION:

Creates a location stream as a subset of another according to mask.

A.4.2 MAPL_LocStreamTransform

A.4.2.1 Transform field

INTERFACE:

```
subroutine MAPL_LocStreamTransform      (LocStream, OUTPUT, INPUT, MASK, &
                                         GRID_ID, GLOBAL, ISMINE, INTERP, RC )
```

ARGUMENTS:

```
type(ESMF_Field),          intent(OUT) :: OUTPUT
type(ESMF_Field),          intent(INout) :: INPUT
type(MAPL_LocStream),      intent(IN ) :: LocStream
integer, optional,         intent(IN ) :: MASK(:)
logical, optional,         intent(IN ) :: ISMINE(:), INTERP
logical, optional,         intent(IN ) :: GLOBAL
integer, optional,         intent(IN ) :: GRID_ID
integer, optional,         intent(OUT) :: RC
```

A.4.2.2 T2G

INTERFACE:

```
subroutine MAPL_LocStreamTransform    (LocStream, OUTPUT, INPUT, MASK, SAMPLE, TRANSPOSE, RC )
```

ARGUMENTS:

```
type(MAPL_LocStream),      intent(IN ) :: LocStream
real,                      intent(INOUT) :: OUTPUT(:,:)
real,                      intent(INOUT) :: INPUT(:)
logical, optional,         intent(IN ) :: MASK(:)
logical, optional,         intent(IN ) :: SAMPLE
logical, optional,         intent(IN ) :: TRANSPOSE
integer, optional,         intent(OUT) :: RC
```

A.4.2.3 G2T

INTERFACE:

```
subroutine MAPL_LocStreamTransform    ( LocStream, OUTPUT, INPUT,      &
                                        MASK, GRID_ID, GLOBAL, ISMINE, &
                                        INTERP, TRANSPOSE, RC )
```

ARGUMENTS:

```
type(MAPL_LocStream),      intent(IN ) :: LocStream
real,                      intent(INOUT) :: OUTPUT(:)
real,                      intent(INOUT) :: INPUT(:,:)
logical, optional,         intent(IN ) :: MASK(:), ISMINE(:), INTERP
logical, optional,         intent(IN ) :: GLOBAL
integer, optional,         intent(IN ) :: GRID_ID
logical, optional,         intent(IN ) :: TRANSPOSE
integer, optional,         intent(OUT) :: RC
```

A.4.2.4 T2T

INTERFACE:

```
subroutine MAPL_LocStreamTransform    ( OUTPUT, XFORM, INPUT, RC )
```

ARGUMENTS:

```
real,                       intent(OUT) :: OUTPUT(:)
type(MAPL_LocStreamXform), intent(IN ) :: XFORM
real,                       intent(IN ) :: INPUT(:)
integer, optional,          intent(OUT) :: RC
```

A.4.2.5 T2TR4R8

INTERFACE:

```
subroutine MAPL_LocStreamTransform        ( OUTPUT, XFORM, INPUT, RC )
```

ARGUMENTS:

```
  real(kind = ESMF_KIND_R8),   intent(OUT) :: OUTPUT(:)
  type(MAPL_LocStreamXform), intent(IN ) :: XFORM
  real,                      intent(IN ) :: INPUT(:)
  integer, optional,         intent(OUT) :: RC
```

A.4.2.6 T2TR8R4

INTERFACE:

```
subroutine MAPL_LocStreamTransform        ( OUTPUT, XFORM, INPUT, RC )
```

ARGUMENTS:

```
  real,                      intent(OUT) :: OUTPUT(:)
  type(MAPL_LocStreamXform), intent(IN ) :: XFORM
  real(kind = ESMF_KIND_R8),   intent(IN ) :: INPUT(:)
  integer, optional,         intent(OUT) :: RC
```

A.5 Module MAPL_BaseMod — A Collection of Assorted MAPL Utilities

USES:

```
use ESMF
use MAPL_ConstantsMod, only: MAPL_PI, MAPL_PI_R8
```

PUBLIC MEMBER FUNCTIONS:

```
public MAPL_AllocateCoupling    ! Atanas: please provide 1-line for each
public MAPL_FieldAllocCommit
```

```
 public MAPL_FieldF90Deallocate
public MAPL_Asrt
public MAPL_ClimInterpFac
 public MAPL_ConnectCoupling
public MAPL_DecomposeDim
public MAPL_FieldCreate
public MAPL_FieldCreateEmpty
public MAPL_FieldGetTime
public MAPL_FieldSetTime
public MAPL_GridGet
public MAPL_IncYMD
public MAPL_Interp_Fac
public MAPL_LatLonGridCreate   ! Creates regular Lat/Lon ESMF Grids
public MAPL_Nhmsf
public MAPL_Nsecf2
public MAPL_PackTime
public MAPL_RemapBounds
public MAPL_Rtrn
public MAPL_Tick
public MAPL_TimeStringGet
public MAPL_UnpackTime
public MAPL_Vrfy
public MAPL_RmQualifier
public MAPL_GetImsJms
public MAPL_AttributeSet
public MAPL_SetPointer
public MAPL_FieldCopyAttributes
public MAPL_StateAdd
public MAPL_FieldBundleAdd
public MAPL_FieldBundleGet
public MAPL_FieldDestroy
public MAPL_FieldBundleDestroy
public MAPL_GetHorzIJIndex
public MAPL_GenGridName
public MAPL_GeosNameNew
public MAPL_Communicators

  !PUBLIC PARAMETERS
integer, public, parameter :: MAPL_CplUNKNOWN         =  0
integer, public, parameter :: MAPL_CplSATISFIED       =  1
integer, public, parameter :: MAPL_CplNEEDED          =  2
integer, public, parameter :: MAPL_CplNOTNEEDED       =  4
integer, public, parameter :: MAPL_FriendlyVariable   =  8
integer, public, parameter :: MAPL_FieldItem          =  8
integer, public, parameter :: MAPL_BundleItem         =  16
```

```
integer, public, parameter :: MAPL_NoRestart          =  32

integer, public, parameter :: MAPL_Write2Disk         =  0
integer, public, parameter :: MAPL_Write2RAM          =  1

integer, public, parameter :: MAPL_VLocationNone    =  0
integer, public, parameter :: MAPL_VLocationEdge    =  1
integer, public, parameter :: MAPL_VLocationCenter  =  2

integer, public, parameter :: MAPL_DimsUnknown      =  0
integer, public, parameter :: MAPL_DimsVertOnly     =  1
integer, public, parameter :: MAPL_DimsHorzOnly     =  2
integer, public, parameter :: MAPL_DimsHorzVert     =  3
integer, public, parameter :: MAPL_DimsTileOnly     =  4
integer, public, parameter :: MAPL_DimsTileTile     =  5

integer, public, parameter :: MAPL_ScalarField      =  1
integer, public, parameter :: MAPL_VectorField      =  2


integer, public, parameter :: MAPL_CplAverage       =  0
integer, public, parameter :: MAPL_CplMin           =  1
integer, public, parameter :: MAPL_CplMax           =  2
integer, public, parameter :: MAPL_MinMaxUnknown    =  MAPL_CplAverage

integer, public, parameter :: MAPL_AttrGrid         =  1
integer, public, parameter :: MAPL_AttrTile         =  2

integer, public, parameter :: MAPL_UnInitialized   =  0
integer, public, parameter :: MAPL_InitialDefault   =  1
integer, public, parameter :: MAPL_InitialRestart   =  2

integer, public, parameter :: MAPL_DuplicateEntry   =  -99
integer, public, parameter :: MAPL_Self  =  0
integer, public, parameter :: MAPL_Import  =  1
integer, public, parameter :: MAPL_Export  =  2
integer, public, parameter :: MAPL_ConnUnknown  =  -1
integer, public, parameter :: MAPL_FirstPhase    =  1
integer, public, parameter :: MAPL_SecondPhase   =  MAPL_FirstPhase+1
integer, public, parameter :: MAPL_ThirdPhase    =  MAPL_FirstPhase+2
integer, public, parameter :: MAPL_FourthPhase   =  MAPL_FirstPhase+3
integer, public, parameter :: MAPL_FifthPhase    =  MAPL_FirstPhase+4

real,    public, parameter :: MAPL_UNDEF              =  1.0e15
```

```
integer, public, parameter :: MAPL_Ocean               =  0
integer, public, parameter :: MAPL_Lake                =  19
integer, public, parameter :: MAPL_LandIce             =  20
integer, public, parameter :: MAPL_Land                =  100
integer, public, parameter :: MAPL_Vegetated           =  101

integer, public, parameter :: MAPL_NumVegTypes         =  6


integer, public, parameter :: MAPL_HorzTransOrderBinning   =  0
integer, public, parameter :: MAPL_HorzTransOrderBilinear  =  1
integer, public, parameter :: MAPL_HorzTransOrderSample    =  99

character(len = ESMF_MAXSTR), public, parameter :: MAPL_StateItemOrderList  =  'MAPL_StateIte
character(len = ESMF_MAXSTR), public, parameter :: MAPL_BundleItemOrderList  =  'MAPL_BundleI

type MAPL_Communicators
   integer :: maplComm
   integer :: esmfComm
   integer :: ioComm
   integer :: maplCommSize
   integer :: esmfCommSize
   integer :: ioCommSize
   integer :: ioCommRoot
   integer :: myGlobalRank
   integer :: myIoRank
   integer :: CoresPerNode
   integer :: maxMem ! maximum memory per node in megabytes
end type MAPL_Communicators
```

DESCRIPTION:

The module `MAPL_Base` provides a collection assorted utilities and constants used throughout the MAPL Library.

A.5.1 MAPL_LatLonGridCreate — Create regular Lat/Lon Grid

INTERFACE:

```
function MAPL_LatLonGridCreate (Name, vm,                &
                                Config, ConfigFile,      &
                                Nx, Ny,                  &
                                IM_World, BegLon, DelLon, &
                                JM_World, BegLat, DelLat, &
                                LM_World,                &
                                rc)                      &
result(Grid)
```

INPUT PARAMETERS:

```
  character(len = *),            intent(in)  :: Name
  type (ESMF_VM),    OPTIONAL, target,     &
                               intent(in)  :: VM


  There are 3 possibilities to provide the coordinate information:

                                           ! 1) Thru Config object:
  type(ESMF_Config), OPTIONAL, target,     &
                               intent(in)  :: Config

                                           ! 2) Thru a resource file:
  character(len = *),  OPTIONAL, intent(in)  :: ConfigFile


                                           ! 3) Thru argument list:
  integer,           OPTIONAL, intent(in)  :: Nx, Ny          ! Layout
  integer,           OPTIONAL, intent(in)  :: IM_World        ! Zonal
  real,              OPTIONAL, intent(in)  :: BegLon, DelLon  ! in degrees

  integer,           OPTIONAL, intent(in)  :: JM_World        ! Meridional
  real,              OPTIONAL, intent(in)  :: BegLat, DelLat  ! in degrees

  integer,           OPTIONAL, intent(in)  :: LM_World        ! Vertical
```

OUTPUT PARAMETERS:

```
  type (ESMF_Grid)                         :: Grid  ! Distributed grid
  integer,           OPTIONAL, intent(out) :: rc    ! return code
```

DESCRIPTION:

This routine creates a distributed ESMF grid where the horizontal coordinates are regular longitudes and latitudes. The grid is created on the user specified **VM**, or on the current VM if the user does not specify one. The layout and the coordinate information can be provided with a `ESMF_Config attribute`, a resource file name or specified through the argument list.

Using resource files

The **resource file** `ConfigFile` has a syntax similar to a GrADS control file. Here is an example defining a typical GEOS-5 1x1.25 grid with 72 layers:

```
GDEF: LatLon
IDEF: 32
JDEF: 16
LDEF:  1
XDEF: 288 LINEAR -180. 1.25
YDEF: 181 LINEAR -90. 1.
ZDEF:  72 LINEAR 1 1
```

More generally,

```
GDEF: LatLon
IDEF: Nx
JDEF: Ny
LDEF: Nz
XDEF: IM_World XCoordType BegLon, DelLon
YDEF: JM_World YCoordType BegLat, DelLat
ZDEF: LM_World ZCoordType 1        1
```

The attribute **GDEF** must always be `LatLon` for Lat/Lon grids. The remaining parameters are:

Nx is the number of processors used to decompose the X dimension

Ny is the number of processors used to decompose the Y dimension

Nz is the number of processors used to decompose the Z dimension; must be 1 for now.

IM_World is the number of longitudinal grid points; if `IM_World = 0` then the grid has no zonal dimension.

XCoordType must be set to LINEAR

BegLon is the longitude (in degrees) of the *center* of the first gridbox

DelLon is the constant mesh size (in degrees); if `DelLon<1` then a global grid is assumed.

JM_World is the number of meridional grid points if `JM_World = 0` then the grid has no meridional dimension.

YCoordType must be set to LINEAR

BegLat is the latitude (in degrees) of the *center* of the first gridbox

DelLat is the constant mesh size (in degrees); if `DelLat<1` then a global grid is assumed.

LM_World is the number of vertical grid points; if `LM_World = 0` then the grid has no vertical dimension.

As of this writing, only the size of the vertical grid (`LM_World`) needs to be specified.

Passing an ESMF Config

The **ESMF_Config** object `Config`, when specified, must contain the same information as the resource file above.

subsubsection*Providing parameters explicitly through the argument list

Alternatively, one can specify coordinate information in the argument list; their units and meaning is as in the resource file above. In this case you must specify at least `Nx, Ny, IM_World, JM_World,` and `LM_World`. The other parameters have default values

BegLon defaults to -180. (the date line)

DelLon defaults to -1. (meaning a global grid)

BegLat defaults to -90. (the south pole)

DelLat deaults to -1. (meaning a global grid)

Restrictions

The current implementation imposes the following restrictions:

1. Only uniform longitude/latitude grids are supported (no Gaussian grids).
2. Only 2D Lon-Lat or 3D Lon-Lat-Lev grids are currently supported (no Lat-Lev or Lon-Lev grids supprted yet).
3. No vertical decomposition yet (`Nz = 1`).

Future enhancements

The `IDEF/JDEF/LDEF` records in the resource file should be extended as to allow specification of a more general distribution. For consistency with the `XDEF/YDEF/ZDEF` records a similar syntax could be adopted. For example,

```
IDEF 4   LEVELS  22 50 50 22
XDEF 144 LINEAR -180 2.5
```

would indicate that longitudes would be decomposed in 4 PETs, with the first PET having 22 grid points, the second 50 gridpoints, and so on.

A.5.2 MAPL_GetHorzIJIndex – Get indexes on destributed ESMF grid for an arbitary lat and lon

INTERFACE:

```
subroutine MAPL_GetHorzIJIndex(lon,lat,npts,Grid,II,JJ,rc)
```

ARGUMENTS:

```
real,              intent(in   ) :: lon(:) ! array of longitudes in radians
real,              intent(in   ) :: lat(:) ! array of latitudes in radians
integer,           intent(in   ) :: npts ! number of points in lat and lon arrays
type(ESMF_Grid),   intent(inout) :: Grid ! ESMF grid
integer,           intent(inout) :: II(:) ! array of the first index for each lat and lo
integer,           intent(inout) :: JJ(:) ! array of the second index for each lat and l
integer, optional, intent(out  ) :: rc  ! return code

!DESCRIPTION
For a set of longitudes and latitudes in radians this routine will return the indexes fo
```

```
If the Lat/Lon pair is not in the domain -1 is returned.
The routine works for both the gmao cube and lat/lon grids.
Currently the lat/lon grid is asumed to go from -180 to 180
```

A.6 Module ESMFL_MOD

USES:

```
use ESMF
use MAPL_ConstantsMod
use MAPL_BaseMod
use MAPL_CommsMod
```

```
ALT These need to be changed!!! values here are just to compile
```

DEFINED PARAMETERS:

```
integer, parameter, public :: ESMFL_UnitsRadians  =  99
```

PUBLIC MEMBER FUNCTIONS:

```
public ESMFL_StateGetField
public ESMFL_StateGetFieldArray
public ESMFL_StateGetPointerToData
public ESMFL_BundleGetPointerToData
public ESMFL_BundleCpyField
public ESMFL_GridCoordGet
! public ESMFL_Connect2STATE
public ESMFL_FCOLLECT
public ESMF_GRID_INTERIOR
public ESMFL_StateFreePointers
public ESMFL_StateSetFieldNeeded
public ESMFL_StateFieldIsNeeded
public ESMFL_FieldGetDims
public ESMFL_GridDistBlockSet
```

```
public ESMFL_FieldRegrid ! alt: this should be MAPL_FieldRegrid
                         !      (topo_bin may need to be here)

public ESMFL_RegridStore ! only used for regridding using ESMF_FieldRegrid
public ESMFL_Regrid
public ESMFL_Diff
public ESMFL_State2Bundle
public ESMFL_Bundle2State
public ESMFL_Bundles2Bundle
public ESMFL_Add2Bundle
public ESMFL_HALO
public ESMFL_BundleAddState
public MAPL_AreaMean
```

A.6.1 ESMFL_GridCoordGet - retrieves the coordinates of a particular axis

INTERFACE:

```
subroutine ESMFL_GridCoordGet(GRID, coord, name, Location, Units, rc)
```

ARGUMENTS:

```
type(ESMF_Grid),   intent(INout ) :: GRID
real, dimension(:,:), pointer  :: coord
character (len = *) , intent(IN) :: name
type(ESMF_StaggerLoc)          :: location
integer                        :: units
integer, optional              :: rc
```

A.6.2 ESMFL_RegridStore

INTERFACE:

```
subroutine ESMFL_RegridStore (srcFLD, SRCgrid2D, dstFLD, DSTgrid2D, &
                              vm, rh, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_Field), intent(inout)       :: srcFLD
type(ESMF_Field), intent(inout)       :: dstFLD
type(ESMF_Grid), intent(out)          :: SRCgrid2D
type(ESMF_Grid), intent(out)          :: DSTgrid2D
type(ESMF_RouteHandle), intent(inout) :: rh
type(ESMF_VM), intent(in)             :: vm  ! should be intent IN
integer, optional, intent(OUT)        :: rc
```

DESCRIPTION:

Given a srcFLD and its associated 3dGrid and a dstFLD and its associated 3DGrid create their corresponding 2DGrids and a 2D routehandle.

A.6.3 FieldRegrid1

INTERFACE:

```
subroutine FieldRegrid1 (srcFLD, Sgrid2D, dstFLD, Dgrid2D, &
                                 vm, rh, fname, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_Field), intent(in)          :: srcFLD
type(ESMF_Field), intent(inout)       :: dstFLD
type(ESMF_Grid), intent(in)           :: Sgrid2D
type(ESMF_Grid), intent(in)           :: Dgrid2D
type(ESMF_RouteHandle), intent(inout) :: rh
type(ESMF_VM), intent(inout)          :: vm
! assumes name of src and dst are the same!!
character(len = *), intent(in)          :: fname
integer, optional, intent(out)        :: rc
```

DESCRIPTION:

Regrid 3D fields using ESMF_FieldRegrid

A.6.4 BundleRegrid1

INTERFACE:

```
subroutine BundleRegrid1 (srcBUN, Sgrid2D, dstBUN, Dgrid2D, &
                          vm, rh, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inOUT)      :: srcBUN
type(ESMF_FieldBundle), intent(inout)      :: dstBUN
type(ESMF_Grid), intent(in)           :: Sgrid2D
type(ESMF_Grid), intent(in)           :: Dgrid2D
type(ESMF_RouteHandle), intent(inout) :: rh
type(ESMF_VM), intent(inout)          :: vm
integer, optional, intent(out)        :: rc
```

DESCRIPTION:

A.6.5 BundleRegrid

INTERFACE:

```
subroutine BundleRegrid (srcBUN, dstBUN, rc)
```

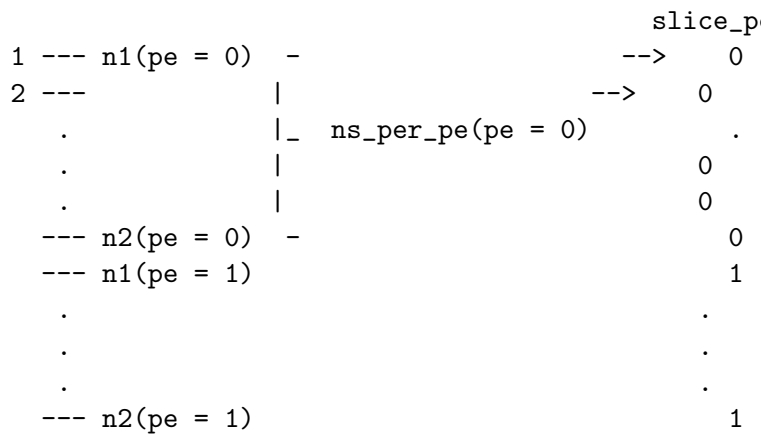
USES:

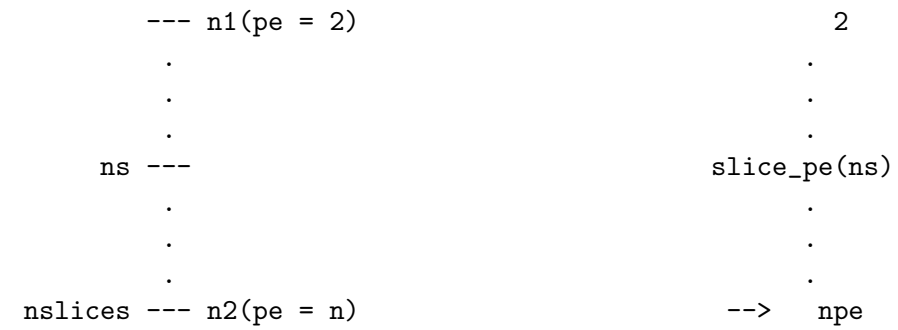
ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inout)      :: srcBUN ! source bundle
type(ESMF_FieldBundle), intent(inout)      :: dstBUN ! destination bundle
integer, optional, intent(out)        :: rc     ! return code
```

DESCRIPTION:

Regrid a source bundle (srcBUN) into a destination bundle (dstBUN) using hinterp. A bundle is thought of as being comprised of n 2D slices (nslices) distributed among the n PEs (ns_per_pe). The limits among each ns_per_pe region are given by n1 and n2 which are functions of mype (the local PE):





Each slice is gathered, regridded (hinterp), and scattered on a PE determined by a slice-to-PE map (slice_pe) to "load balance" the work of the serial hinterp function.

A.6.6 Bundle_Prep_

INTERFACE:

```
subroutine Bundle_Prep_ (srcBUN, dstBUN, only_vars)
```

USES:

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inout)      :: srcBUN !ALT: intent(in)
type(ESMF_FieldBundle), intent(inout)      :: dstBUN
character(len = *), optional, intent(in):: only_vars ! comma separated,
                                                     ! no spaces
```

DESCRIPTION:

Prepare for regridding

A.6.7 assign_slices_

INTERFACE:

```
subroutine assign_slices_ (nslices, mype, npe, slice_pe, nfirst, nlast)
```

USES:

ARGUMENTS:

```
integer, intent(in)    :: nslices       ! number of slices
integer, intent(in)    :: mype          ! local PE
integer, intent(in)    :: npe           ! number of PEs
integer, intent(inout) :: slice_pe(:)   ! slice-to-pe map
integer, intent(out)   :: nfirst
integer, intent(out)   :: nlast
```

DESCRIPTION:

Determine number of bundle slices per PE and "load balanced" map of slices-to-pes (slice_pe)

A.6.8 Do_Gathers_

INTERFACE:

```
subroutine Do_Gathers_ (BUN, BUF)
```

USES:

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inout)         :: BUN
real(4), intent(inout), dimension(:,:,:) :: BUF
```

DESCRIPTION:

gather FLDs in a BUNdle on all PEs into a BUFfer Note: local adressing is used

---

A.6.9 Do_Regrid_

INTERFACE:

```
subroutine Do_Regrid_ (n, inBuf, outBuf)
```

USES:

ARGUMENTS:

```
integer, intent(in)                    :: n      ! slice index
real(4), intent(inout), dimension(:,:) :: inBuf  ! source buffer
real(4), intent(inout), dimension(:,:) :: outBuf ! destination buffer
```

DESCRIPTION:

Call hinterp on local PE

A.6.10 Do_Scatters_

INTERFACE:

```
subroutine Do_Scatters_ (BUN, BUF)
```

USES:

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inout)          :: BUN
real(4), intent(inout), dimension(:,:,:) :: BUF
```

DESCRIPTION:

scatter from BUffer onto FLDs in a BUNdle Note: local adressing is used

---

A.6.11 StateRegrid

INTERFACE:

```
subroutine StateRegrid (srcSTA, dstSTA, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_State), intent(inout)  :: srcSTA
type(ESMF_State), intent(inout)  :: dstSTA
integer, optional, intent(out)   :: rc     ! return code
```

DESCRIPTION:

Regrid a state

A.6.12 ESMFL_FieldGetDims

INTERFACE:

```
subroutine ESMFL_FieldGetDims(FLD, gCPD, lCPD, lm, ar)
```

USES:

ARGUMENTS:

```
type(ESMF_Field), intent(inout)         :: FLD !ALT: intent(in)
integer, optional, intent(out)       :: gCPD(3)
integer, optional, intent(out)       :: lCPD(3)
integer, optional, intent(out)       :: lm
integer, optional, intent(out)       :: ar
```

DESCRIPTION:

Return some grid information associated from an ESMF field

A.6.13 BundleDiff

INTERFACE:

```
subroutine BundleDiff (srcBUN, dstBUN, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inout) :: srcBUN
type(ESMF_FieldBundle), intent(inout) :: dstBUN
integer, optional, intent(out)   :: rc     ! return code
```

DESCRIPTION:

diff two bundles

A.6.14 StateDiff

INTERFACE:

```
subroutine StateDiff (srcSTA, dstSTA, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_State), intent(inout)  :: srcSTA
type(ESMF_State), intent(inout)  :: dstSTA
integer, optional, intent(out)   :: rc     ! return code
```

DESCRIPTION:

Regrid a state

A.6.15 ESMFL_GridDistBlockSet

INTERFACE:

```
subroutine ESMFL_GridDistBlockSet (Egrid, ist, jst, il, jl, &
                                   rlons, rlats, rc)
```

USES:

ARGUMENTS:

```
type(ESMF_Grid), intent(inout)      :: Egrid
integer, intent(in), dimension(:):: ist, jst, il, jl
real(8), optional, dimension(:)     :: rlats
real(8), optional, dimension(:)     :: rlons
integer, optional, intent(out)      :: rc    ! return code
```

DESCRIPTION:

A.7 Module MAPL_HistoryGridCompMod

USES:

```
use ESMF
use ESMFL_Mod
use MAPL_BaseMod
use MAPL_VarSpecMod
use MAPL_ConstantsMod
use MAPL_IOMod
use MAPL_CommsMod
use MAPL_GenericMod
use MAPL_LocStreamMod
use MAPL_CFIOMod
use MAPL_GenericCplCompMod
use MAPL_NewArthParserMod
use MAPL_SortMod
use ESMF_CFIOMOD, only:  StrTemplate  = > ESMF_CFIOstrTemplate
use m_chars,  only: uppercase
```

```
use MAPL_CFIOServerMod
!use ESMF_CFIOMOD
```

PUBLIC MEMBER FUNCTIONS:

```
public SetServices
```

DESCRIPTION:

`MAPL_HistoryGridCompMod` is an internal MAPL gridded component used to manange output streams from a MAPL hierarchy. It write Fields in the Export states of all MAPL components in a hierarchy to file collections during the course of a run. It also has the some limited capability to interpolate the fields horizontally and/or vertically beofore outputing them.

It is usually one of the two gridded components in the "cap" or main program of a MAPL application, the other being the root of the MAPL hierarchy it is servicing. It is instanciated and all its registered methods are run automatically by `MAPL_Cap,` if that is used. If writing a custom cap, `MAPL_HistoryGridCompMod`'s SetServices can be called anytime after ESMF is initialized. Its Initialize method should be executed before entering the time loop, and its Run method at the bottom of each time loop, after advancing the Clock. 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 MAPL component, but it generally makes no sense to checkpoint and restart these.

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

`MAPL_HistoryGridCompMod` uses `MAPL_CFIO` for creating and writing its files; it thus obeys all `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 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 `EXDID:` attribute value. A filename extension may also be added to identify the type of file (e.g., .nc4).

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

The extension is not added automatically, it is up to the user to add the appropriate one. If the format is CFIO or CFIOasync and the extension is absent or .nc a NETCDF4 classic file will be produced. Is the extentions is .nc4 a NETCDF4 file will be produced. If it is "flat", the data files have whatever extension you provide 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 ("CFIO", "CFIOasync", "flat"). "CFIO" uses MAPL_CFIO and produces netcdf output. "CFIOasync" uses MAPL_CFIO but delegates the actual I/O to the MAPL_CFIOServer (see MAPL_CFIOServer documenation for details). Default = "flat".

`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 *frquency*; 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

`end_time` Integer (HHMMSS) ending time 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 ouput stream. Transforms betwee 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`($\cdot$), `pow`($\cdot$,*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.

`conservative` Set to a non-zero integer to turn on conservative regridding when going from a native cube-sphere grid to lat-lon output. Default: 0

`deflate` Set deflate level (0-9) of NETCDF output when format is CFIO or CFIOasync. Default: 0

`subset` Optional subset (lonMin lonMax latMin latMax) for the output when performing non-conservative cube-sphere to lat-lon regridding of the output.

`chunksize` Optional user specified chunking of NETCDF output when format is CFIO or CFIOasync, (Lon chunksize, Lat chunksize, Lev chunksize, Time chunksize)

The following is a sample HISORY.rc take from the FV_HeldSuarez test.

```
EXPID:  fvhs_example
EXPDSC: 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:     'CFIO',
dynamics_vars_eta.frequency:  240000,
dynamics_vars_eta.duration:   240000,
dynamics_vars_eta.fields:     'T_EQ'     , 'HSPHYSICS'           ,
                              'U'        , 'FVDYNAMICS'          ,
                              'V'        , 'FVDYNAMICS'          ,
                              'T'        , 'FVDYNAMICS'          ,
                              'PLE'      , 'FVDYNAMICS'          ,
                       ::
```

```
dynamics_vars_p.template:   '%y4%m2%d2_%h2%n2z',
dynamics_vars_p.format:     'flat',
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 1,
dynamics_vars_p.fields:     'T_EQ'     , 'HSPHYSICS'           ,
                            'U'        , 'FVDYNAMICS'          ,
                            'V'        , 'FVDYNAMICS'          ,
                            'T'        , 'FVDYNAMICS'          ,
                            'PLE'      , 'FVDYNAMICS'          ,
                       ::
```

BUGS:

1. It may not be well behaved if more than one instance exists in an application.
2. Its use for servicing a non-MAPL gridded components is not documented.
3. GrADS output is currently done through specialized calls, rather than through the CFIO library.
4. Horizontal and vertical interpolation correctly rely on CFIO, and so are not yet available for GrADS files.
5. If resolution attribute is used to INCREASE resolution, code may break.
6. Grid offsetting is very limited and does not allow for arbitrary rotations in longitude
7. Currently, this component only handles MAPL supported grids, which currently are the regular lat-lon and the cubed-sphere ESMF grids that tile the entire sphere.

A.8 Module MAPL_GenericCplCompMod

DESCRIPTION:

This is a generic coupler component used by MAPL to instantiate the automatic couplers it needs.

INTERFACE:

```
module MAPL_GenericCplCompMod
```

USES:

```
  use ESMF
  use ESMFL_Mod
  use MAPL_BaseMod
  use MAPL_ConstantsMod
  use MAPL_IOMod
  use MAPL_ProfMod
  use MAPL_SunMod
  use MAPL_VarSpecMod
```

PUBLIC MEMBER FUNCTIONS:

```
  public GenericCplSetServices
  public MAPL_CplCompSetVarSpecs
```

A.8.1 GenericCplSetServices

DESCRIPTION:

`SetServices` for generic couplers. INTERFACE:

```
  subroutine GenericCplSetServices ( CC, RC )
```

ARGUMENTS:

```
    type (ESMF_CplComp  ),          intent(INOUT) :: CC
    integer,                        intent(  OUT) :: RC
```

A.8.2 INITIALIZE

DESCRIPTION:

Initialize method for generic couplers. INTERFACE:

```
subroutine Initialize(CC, SRC, DST, CLOCK, RC)
```

ARGUMENTS:

```
type (ESMF_CplComp)      :: CC
type (ESMF_State)        :: SRC
type (ESMF_State)        :: DST
type (ESMF_Clock)        :: CLOCK
integer, intent(  OUT)   :: RC
```

A.8.3 RUN

DESCRIPTION:

Run method for the generic coupler. INTERFACE:

```
subroutine Run(CC, SRC, DST, CLOCK, RC)
```

ARGUMENTS:

```
type (ESMF_CplComp)      :: CC
type (ESMF_State)        :: SRC
type (ESMF_State)        :: DST
type (ESMF_Clock)        :: CLOCK
integer, intent(  OUT)   :: RC
```

A.8.4 FINALIZE

DESCRIPTION:

Finalize method for the generic coupler. INTERFACE:

```
subroutine Finalize(CC, SRC, DST, CLOCK, RC)
```

ARGUMENTS:

```
type (ESMF_CplComp)      :: CC
type (ESMF_State)        :: SRC
type (ESMF_State)        :: DST
type (ESMF_Clock)        :: CLOCK
integer, intent(  OUT)   :: RC
```

A.9 Module MAPL_ExtDataGridCompMod - Implements Interface to External Data

DESCRIPTION:

`MAPL_ExtDataGridComp` is an ESMF gridded component implementing an interface to boundary conditions and other types of external data files.

Developed for GEOS-5 release Fortuna 2.0 and later. *USES:*

```
USE ESMF
use MAPL_BaseMod
use MAPL_CommsMod
use ESMFL_Mod
use MAPL_GenericMod
use MAPL_VarSpecMod
use ESMF_CFIOFileMod
use ESMF_CFIOMod
use ESMF_CFIOUtilMod
use MAPL_CFIOMod
```

```
use MAPL_NewArthParserMod
use MAPL_ConstantsMod, only: MAPL_PI

IMPLICIT NONE
PRIVATE
```

PUBLIC MEMBER FUNCTIONS:

```
PUBLIC SetServices
```

A.9.1 SetServices — Sets IRF services for the MAPL_ExtData

INTERFACE:

```
SUBROUTINE SetServices ( GC, RC )
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(INOUT) :: GC  ! gridded component
integer, optional                  :: RC  ! return code
```

DESCRIPTION:

Sets Initialize, Run and Finalize services. REVISION HISTORY:

```
12Dec2009  da Silva  Design and first implementation.
```

A.9.2 Initialize_ — Initialize MAPL_ExtData

INTERFACE:

```
SUBROUTINE Initialize_ ( GC, IMPORT, EXPORT, CLOCK, rc )
```

USES:

INPUT PARAMETERS:

```
type(ESMF_Clock),  intent(inout)   :: CLOCK   ! The clock
```

OUTPUT PARAMETERS:

```
type(ESMF_GridComp), intent(inout) :: GC      ! Grid Component
type(ESMF_State), intent(inout)    :: IMPORT  ! Import State
type(ESMF_State), intent(inout)    :: EXPORT  ! Export State
integer, intent(out)               :: rc      ! Error return code:
                                              !  0 - all is well
                                              !  1 -
```

DESCRIPTION:

This is a simple ESMF wrapper. REVISION HISTORY:

```
12Dec2009  da Silva  Design and first implementation.
```

A.9.3 Run_ — Runs MAPL_ExtData

INTERFACE:

```
SUBROUTINE Run_ ( GC, IMPORT, EXPORT, CLOCK, rc )
```

USES:

INPUT PARAMETERS:

```
type(ESMF_Clock),  intent(inout) :: CLOCK     ! The clock
```

OUTPUT PARAMETERS:

```
type(ESMF_GridComp), intent(inout)  :: GC     ! Grid Component
type(ESMF_State), intent(inout) :: IMPORT     ! Import State
type(ESMF_State), intent(inout) :: EXPORT     ! Export State
integer, intent(out) ::  rc                   ! Error return code:
                                              !  0 - all is well
                                              !  1 -
```

DESCRIPTION:

This is a simple ESMF wrapper. REVISION HISTORY:

```
12Dec2009  da Silva  Design and first implementation.
```

A.9.4 Finalize_ — Finalize MAPL_ExtData

INTERFACE:

```
SUBROUTINE Finalize_ ( GC, IMPORT, EXPORT, CLOCK, rc )
```

USES:

INPUT PARAMETERS:

```
type(ESMF_Clock),  intent(inout) :: CLOCK      ! The clock
```

OUTPUT PARAMETERS:

```
type(ESMF_GridComp), intent(inout)  :: GC     ! Grid Component
type(ESMF_State), intent(inout) :: IMPORT     ! Import State
type(ESMF_State), intent(inout) :: EXPORT     ! Export State
integer, intent(out) ::  rc                   ! Error return code:
                                              !  0 - all is well
                                              !  1 -
```

DESCRIPTION:

This is a simple ESMF wrapper. REVISION HISTORY:

```
12Dec2009  da Silva  Design and first implementation.
```

Bibliography

[1] DeLuca, C. and the ESMF Joint Specification Team. *Earth System Modeling Framework Project Plan 2005-2010*, `http://www.esmf.ucar.edu` (is the website correct!?!?!)

[2] ESMF Joint Specification Team. *ESMF Reference Manual for Fortran, Version 5.3* http://www.earthsystemmodeling.org

[3] Hill, C., C. DeLuca, V. Balaji, M. Suarez, A. da Silva. *The Architecture of the Earth System Modeling Framework*. Computing in Science and Engineering, Vol. 11, No. 6, January/February 2004, pp. 18-28.

[4] Held, I.M. and M.J. Suarez. *A proposal for the intercomparison of the dynamical cores of atmospheric general circulation models*. Bulletin of the American Meteorological Society, 75(10), 1825-1830, 1994.

[5] Williamson, D.L., J.G. Olson, B.A. Boville. *A comparison of semi-Lagrangian and Eulerian tropical climate simulations*. Mon. Wea. Rev., 126, 1001-1012, 1998.

Index

This page is intentionally left blank.