The FMS Manual:
A developer's guide to the GFDL Flexible Modeling System

V. Balaji (for the FMS Development Team)

17 December 2002

Abstract

The GFDL Flexible Modeling System (FMS) is a publicly available software infrastructure for the construction of atmosphere, ocean, and coupled climate models. The code is portable across a wide variety of scalar and parallel computing architectures, and is designed for flexibility, modularity and extensibility.

The FMS manual is aimed at all users and developers of models using the FMS infrastructure. It describes the scope of the FMS infrastructure; the standards and conventions employed by FMS; the principles of design of individual FMS components and elements; the policies regarding its use both inside and outside GFDL, including the proposal and implementation of modifications and extensions; as well as providing links to the actual source and documentation. It is intended that the manual will permit a free and open manner for the user/developer to interact with FMS, including configuring models and running them, designing and building extensions, porting to new platforms, and replacing any FMS component with new code that will be usable across the FMS community.

An online version of this document is available at:
`http://www.gfdl.gov/~vb/FMSManual`

Contents

Chapter 1

Overview

1.1 Introduction

In climate research, with the increased emphasis on detailed representation of individual physical processes governing the climate, the construction of a model has come to require large teams working in concert, with individual sub-groups each specializing in a different component of the climate system, such as the ocean circulation, the biosphere, land hydrology, radiative transfer and chemistry, and so on. The development of model code now requires teams to be able to contribute components to an overall coupled system, with no single kernel of researchers mastering the whole. This may be called the *distributed development model*, in contrast with the monolithic small-team model of earlier decades.

A simultaneous trend is the increase in hardware and software complexity in high-performance computing, as we shift toward the use of scalable computing architectures. Scalable architectures come in several varieties, including shared-memory parallel vector systems, distributed memory massively-parallel systems, and distributed shared-memory NUMA systems. The individual computing elements themselves can embody complex memory hierarchies. To facilitate sharing of code and development costs across multiple institutions, it is necessary to abstract away the details of the underlying architecture and provide a uniform programming model across different scalable and uniprocessor architectures.

These developments entail a change in the programming paradigm used in the construction of complex earth systems models. The approach is to build code out of independent modular components, which can be assembled by either choosing a configuration of components suitable to the scientific task at hand, or else easily extended to such a configuration. The code must thus embody the principles of *modularity, flexibility and extensibility*.

The current trend in model development is along these lines, with systematic efforts under way in Europe and the U.S to develop shared infrastructure for earth systems models. It is envisaged that the models developed on this shared infrastructure will go to meet a variety of needs: they will work on different available computer architectures at different levels of complexity, with the same model code using one set of components on a university researcher's desktop, and with a different choice of subsystems, running comprehensive assessments of climate evolution at large supercomputing

sites using the best assembly of climate component models available at the moment.

The shared infrastructure currently in development concentrates on the underlying "plumbing" for coupled earth systems models, building the layers necessary for efficient parallel computation and data transfer between model components on independent grids.

1.2 The GFDL Flexible Modeling System

The Geophysical Fluid Dynamics Laboratory (NOAA/GFDL) undertook a technology modernization program beginning in the late 1990s. The principal aim was to prepare an orderly transition from vector to parallel computing. Simultaneously, the opportunity presented itself for a software modernization effort, the result of which is the GFDL Flexible Modeling System (FMS). FMS is an attempt to address the need to develop high-performance kernels for the numerical algorithms underlying non-linear flow and physical processes in complex fluids, while maintaining the high-level structure needed to harness component models of climate subsystems developed by independent groups of researchers. It constitutes a specification of standards, and a shared software infrastructure implementing those standards, for the construction of climate models and model components for vector and parallel computers. It forms the basis of current and future coupled modeling at GFDL. In 2000, it was benchmarked on a wide variety of high-end computing systems, and runs in production on three very different architectures: parallel vector (PVP), distributed massively-parallel (MPP) and distributed shared-memory (DSM)[1], as well as on scalar microprocessors. Models in production within FMS include a hydrostatic spectral atmosphere, a hydrostatic grid-point atmosphere, an ocean model (MOM), and land and sea ice models. In development, or scheduled for inclusion, are a non-hydrostatic atmospheric model, an isopycnal coordinate ocean model, and an ocean data assimilation system.

The shared software for FMS includes at the lowest level a parallel framework for handling distribution of work among multiple processors, described in Chapter 5. Upon this are built the *exchange grid* software layer for conservative data exchange between independent model grids, and a layer for parallel I/O. Further layers of software include a *diagnostics manager* for creating runtime diagnostic datasets in a variety of file formats, a *time manager*, general utilities for file-handling and error-handling, and a uniform interface to scientific software libraries providing methods such as FFTs. Interchangeable components are designed to present a uniform interface, so that for instance, behind an "ocean model" interface in FMS may lie a full-fledged ocean model, a few lines of code representing a mixed layer, or merely a routine that reads in an appropriate dataset, without requiring other component models to be aware which of these has been chosen in a particular model configuration. Coupled climate models in FMS are built as a *single executable* calling subroutines for component models for the atmosphere, ocean and so on. Component models may be on independent logically rectangular (though possibly physically curvilinear) grids, linked by the exchange grid, and making maximal use of the shared software layers.

This document provides a description of the overall design of FMS, with a specification of the coding constructs required of developers building elements of the FMS. We first lay out the overall

[1] Also known as cache-coherent non-uniform memory access (ccNUMA) architecture.

structure of FMS, followed by a section on general coding standards, and finally standards specific to different FMS elements.

1.3 Purpose of the Manual

The documentation for FMS is divided in three categories: a developer's guide; a user's guide; and a technical description where appropriate. This document serves as the developer's guide. We visualize several categories of developer: consider

- a university researcher with an existing atmospheric model, who wishes to conduct a coupled climate experiment with a slab ocean;
- a scientific programmer porting the FMS infrastructure to a novel scalable architecture;
- a developer of physics parameterizations;
- a climate modeler constructing an FMS model configuration.

The user's guide and technical description are distributed as modular documentation along with the code. The user's guide described the use and call syntax associated with an FMS module, and the technical description provides more details on the algorithms and their implementation. As the examples listed above show, the category of user we call "developer" is broad – every user is a potential developer – and needs a broader document linking the whole. While remaining closely linked to the user guide and technical documents, this developer's guide will provide an understanding of design principles on which the module interfaces and data structures in FMS are constructed, and how it is intended to be used. The underpinning for this is provided by the *standards* describing the design specification for FMS. It is hoped that the manual will permit a free and open manner for the user/developer to interact with FMS, including the design and building of extensions, porting to new platforms, and replacing any FMS component with new code that will be usable across the FMS community.

Chapter 2

General design specification

This chapter describes a design specification for FMS. It first lays out the general design principles which have informed our choices in code design, in Section 2.1. The different elements that constitute FMS models are laid out in Section 2.2. The standards followed by FMS are described in subsequent sections. These include a section on the conventions we employ, a section on programming practice, a section on the representation of physical information and a section on programming style. General documentation standards are then described, also with sections on conventions, practice and style. Finally, in Section 2.3 we describe the organization of the FMS, and how it is delivered as a product to the user/developer. This includes details on the organization of the source, version control, compliance verification and review, compilation and data processing requirements, test scripts, and guidelines for interaction with the FMS Development Team.

2.1 FMS design principles

The construction of FMS is guided by several principles of design, outlined here.

- We follow a *distributed development model*, with large teams working in concert and contributing pieces to FMS from their areas of scientific and technical expertise. The harnessing of the work of disparate teams requires *modular design*, with well-defined standards for the interfaces between modules. These are *open* standards, defined by consultation across a network of developers. Programming practices that embody the modularity principle include:
  - Data-hiding: modules only publish data and interfaces that are needed to interact with other modules, and keep the rest private;
  - Encapsulation: modules encapsulate information with data structures that embody the objects for which it is responsible;
  - Self-sufficiency: Modules are responsible for initializing and configuring themselves, saving their state on termination, and publishing diagnostic information about themselves.

- We intend FMS eventually to be usable across a wide variety of high-end scalable computing architectures, as well as desktop uniprocessor systems. This implies close attention to issues of *portability of code*. We achieve this by close adherence to official language standards, the use of community-standard software packages, and compliance with the standards we have chosen to define for FMS. When departing from the standard (which could happen for instance, if a non-standard proprietary programming interface offered major performance enhancements for a particular platform), a standard-conformant alternative is required.

- We intend FMS to serve a variety of needs within the modeling community, spanning a range of resolutions between paleoclimate modeling, and severe storm simulations, with choices of simple and and comprehensive representations for model components, physics, and so on. This *flexibility* is achieved by designing FMS to permit researchers to assemble and configure particular experiments out of a wide choice of available components and modules.

- We anticipate that users and developers of FMS will extend the scope of FMS beyond what is available at any time. This could include porting to novel architectures, adding dynamical cores, and new physics options to various component models, implementing new grids and numerical stencils and so on. We therefore view *extensibility* as a design principle always to be kept in view. This requires some attempt to anticipate future needs, and design interfaces that can accommodate them. In general, choices of model components for the same function (e.g ocean model, convection parameterization, advection scheme) should have identical interfaces, or be capable of being wrapped under an identical interface. This requires some care and forward thinking.

  Another consideration related to extensibility is in the use of data structures. If it is anticipated that the data structures might evolve, it would be prudent to make them opaque objects (using the `private` keyword in `f90` derived types) whose elements can only be accessed through specified interfaces.

- The design of FMS calls for rigorous *documentation* using standard documentation formats. The documentation itself is modular, and distributed along with the source. At a minimum, the documentation is required to explain how to interact with the module: its syntax, function and public interfaces. In addition, as an aid to developers, it is expected that authors will provide more detailed technical documentation explaining the inner workings of their code, and a rationale behind their design choices.

- As described above, we do expect users of FMS to also be developers. The *building of community* is also viewed as a fundamental design principle vital to the success of a shared software infrastructure, and the distributed development model. We achieve this by situating the distribution of FMS within standard, open, and freely available tools for version control, documentation, source and data distribution, sharing of information, and feedback.

The principles outlined here – **modularity**, **portability**, **flexibility**, **extensibility** and **community** – are in our view the vital elements of a successful distributed development model. The conventions and standards described in the following sections all embody these principles.

2.2 Elements of FMS

The design of FMS is principally aimed at the construction of coupled climate models running as a single executable on vector and parallel high-performance computers. The FMS source tree is divided into three principal sections:

2.2.1 Component models

Component models are models for the atmosphere, ice (or ocean surface), land surface, and ocean subsystems. Any component model conforming to FMS practice can be used as a component model. FMS permits each component model to be run either *solo* or *coupled*.

Each component model comes with:

1. a *solo driver*, a program to run the model standalone, if appropriate;
2. a *coupled model driver*, a routine for communicating with the coupler layer described below;
3. a directory of physical parameterizations of unresolved phenomena. Different choices of parameterization of the *same* process (e.g moist convection in the atmosphere) must use an identical interface;
4. an unspecified number of *component model cores*, each in its own subdirectory. These are choices of a core representation of the component model's role in the coupled system. *One* of these cores in conjunction with the items above provide a comprehensive representation of that climate model component in a coupled system for a particular experiment. These could be dynamical cores (e.g the B-grid or spectral atmosphere) or simplified representations of a model component (e.g a mixed-layer or slab ocean) or even routines that merely read in a dataset (e.g an AMIP dataset for sea ice). Different choices of model for the *same* component must use an identical interface.

2.2.2 Coupler

The *coupler* consists of the main program driving a coupled model, as well as the *exchange grid* software for communicating data between component models, which can be on independent grids. Component models communicate only with the coupler, which mediates all interactions between them.

2.2.3 Shared utilities

These consists of fairly general purpose utility routines that are common to the component models and the coupler layer. These include:

- parallel communication and memory management kernels;

- parallel I/O;
- a diagnostics manager for registering, processing, and writing out model fields;
- standard interfaces to scientific software libraries (e.g for FFTs);
- a model time and calendar manager;
- performance evaluation;
- error-handling.

2.3 Organization of FMS

2.3.1 The FMS source

CVS is the software used for FMS version control. The FMS source is available under CVS as described in the FMS distribution webpage. The directory tree reflects the modeling system structure described in Section 2.2.

2.3.2 The FMS executable

An FMS *experiment* is an instantiation of a source code subset used to run a climate model simulation, solo or coupled. In either case, an experiment consists of a *single executable*. This executable will call component models as routines. On a parallel system, component models may run serially on the same processor set, or concurrently on independent sets of processors.

The executable must be configurable at runtime as much as possible. Features may be frozen at compile time if a clear performance advantage is demonstrable. In particular, the choice of model size, grid, and domain decomposition must be runtime-configurable.

2.3.3 Compilation

FMS is designed to be written in a high-level abstract language, with the current choice being `f90` (see Section 2.4.1). The source is split up among many files, and will contain many inter-procedural dependencies. Compilation is potentially slow, and in the case of `f90`, must be performed in a certain order, following a hierarchy of `use` statements. The use of Makefiles is thus strongly recommended.

The `mkmf` utility supplied with FMS performs source file dependency analysis, with particular attention to `f90`, and will generate Makefiles for the task at hand.

2.4 FMS coding conventions

2.4.1 Language

The FMS code is currently written in Fortran 90Metcalf and Reid (1999). `f90` has many of the high-level features required to satisfy the design principles, while retaining adequate numerical performance.

The programming standards below in Section 2.5 all are written specifically for `f90`. Should code in other languages become part of FMS, these standards will be appropriately extended.

1. All elements of the ANSI `f90` standard are permitted, with a few listed exceptions whose use is discouraged or prohibited. These are enumerated below in Section 2.5.8.

2. Language extensions are severely restricted. They may be used in limited fashion, provided a pressing reason exists (e.g major performance enhancement using a particular proprietary software system), *and* an alternate formulation is provided for compiling environments that do not permit the extension.

3. The language of FMS may change in the future, to Fortran 95 or Fortran 2000, or any other, after review.

2.4.2 Preprocessing

FMS uses preprocessor directives based on `cpp`. The use is intended for language extensions, and in some circumstances, it is used to generate module procedures under a generic interface for variables of different type, kind and rank (thus circumventing `f90`'s strict typing), while maintaining a single copy of the source.

The use of preprocessor directives in FMS is permitted under the following conditions:

1. Where language extensions are used (see Section 2.4.1), `cpp` `#ifdef` statements must be used to shield lines from compilers that may not recognize them.

2. Use is restricted to the builtin preprocessor of the `f90` compiler (based on `cpp`), and cannot be based on external preprocessors such as `m4`. This condition may be relaxed on platforms where the builtin preprocessor proves to be inadequate.

3. Use is restricted to short code sections (a useful rule of thumb is that an `#ifdef` and the matching `#endif` must both be visible on a single 80×24 editor window).

4. Owing to restrictions in certain compilers, preprocessor variable names may not exceed 31 characters.

2.4.3 Code units

FMS source is divided into *software modules*. A software module consists of one of the following:

1. An `f90` module;
2. An `f90` program unit;
3. A group of `f90` modules constituting a *package*. A package is a software unit that has been separated into multiple `f90` modules and source files for convenience, but intended to be used through a single interface, with unified documentation, a single set of public interfaces, a single I/O point, and so on. Examples of packages in FMS include the spectral transform package (which includes a separate source file for the Legendre transform code) and the diagnostics manager, which has been divided for convenience into multiple files. Direct use of the `f90` modules within a package is discouraged, as the individual modules may not adhere to the standards specified here. Packages are identified by the `f90` module at the head of the package tree.

   `f90` modules and source files which are part of a package will be explicitly identified as such, both in the source and in the documentation.

Subsequent discussion only refers to software modules. The manual will explicitly identify items that specifically refer to `f90` modules.

1. A module is responsible for its own I/O, including diagnostics, restarts, and input namelists.
2. A module has a well-defined set of public interfaces, including its own procedural interfaces, file I/O interfaces, and public derived types.

Each source code file defines a single `program` or `f90` `module`.
The general coding standard for a software module is described below in Section 2.5.

2.4.4 Filenames

The *basename* for the `f90` module `module_name` is `module_name`. (Note that the module name extension `_mod` is omitted from the basename). All filenames associated with this module use this basename. The basename for a package is the name at the head of the package tree.

1. The source file for `module_name` is `module_name.f90` (or `.F90` if it contains preprocessor directives).
2. Compilers produce object code for each source file, usually with a `.o` extension (`module_name.o`). During linking, it is required that each object file have a unique name. The `module_name` must be carefully chosen to prevent name collisions. Extremely generic names must be avoided. The recommended practice is to use suitable prefixes identifying the package to which a file belongs (such as `mpp_` or `diag_`).

3. The namelist file, if any, associated with `module_name` is `module_name.nml`. The namelist itself is called `module_name_nml`. Any namelist may also appear as an entry in the file `input.nml`, the general namelist file.

4. The restart file, if any, associated with `module_name` is `module_name.res`. If more than one restart file is present, a unqiue number is appended, thus `module_name#.res`. If the restart is written in netCDF, The extension is `.res.nc`.

5. An ASCII text output file has the extension `.out`.

6. An raw binary output file has the extension `.data`.

7. The documentation associated with instructions for use of `module_name` is preferably formatted for web access, and is named `module_name.html`. Additional (detailed) technical documentation may also be present in other formats, with the basename `module_name.technotes` and a *standard* file extension (e.g `module_name.ps` for a PostScript file, `module_name.pdf` for PDF). PDF is recommended since PDF files are now indexed by many web search engines.

8. If the documentation was generated from LaTeX source, the file `module_name.tex` may also be distributed. The use of non-standard LaTeX packages is discouraged.

9. *Distributed datasets* are datasets where each processor has written its portion of some global data to a separate file, intended for later assembly offline. These should be identified by the 4-digit processor ID *following* the standard extensions described here (e.g `module_name`.nc.####) so it is evident that this is an incomplete file.

Files are sorted in subdirectories below the working directory. The convention calls for input restart files to be read from the `INPUT/` directory, the output restarts to be written to the `RESTART/` directory, and input datasets and namelists to be in the `DATA/` directory. Documentation for a module will reside in the same directory as its source code.

2.4.5 Binary data formats

The preferred format for binary data in FMS is `netCDF`, a self-describing dataset format widely used in the climate modeling community. netCDF follows the IEEE standards for binary data representation. We currently follow the COARDS convention[1] for netCDF metadata. We anticipate that very soon we will adopt the CF convention[2] currently under final review. The CF convention is fully backward-compatible with COARDS.

The conventional extension for netCDF files from FMS is `.nc`.

[1] http://ferret.wrc.noaa.gov/noaa_coop/coop_cdf_profile.html
[2] http://www.cgd.ucar.edu/cms/eaton/netcdf/CF-current.htm

2.4.6 Indices

By convention, spatial indices `(x,y,z)` should use indices `(i,j,k)`.

2.5 Programming Standards

2.5.1 Scope

Each module in FMS must have private scope by default. Each public interface therefore needs to be explicitly published.

2.5.2 Typing

The use of implicit typing is forbidden. Every module must contain the line:

```
implicit none
```

in the module header, and every variable explicitly declared.

Variables are generally assumed to be of default `KIND`. There may sometimes be reason to specify the `KIND` of a variable:

1. If `KIND` must be specified for reasons of precision, the `f90` intrinsics `SELECTED_REAL_KIND` and `SELECTED_INT_KIND` must be used.

2. If `KIND` must be specified in order to control the storage size (bytelength) of a variable (typically in communication and I/O code) it must be done using the parameters `r8_kind`, `r4_kind`, `i8_kind`, `i4_kind` and `i2_kind`, supplied by the module `platform_mod`, which sets various values that are specific to the computing platform. The `platform` module sets these values to the appropriate `KIND` values for FP and integer variables of the required bytelength.

2.5.3 Character variables

There are a few restrictions on the length of a character variable:

1. Character variables that are *arguments* to routines must be declared with `(len=*)`. It has been observed that compilers are inconsistent in their “padding” practices, and the standard is silent on the subject.

2. It is recommended that other character variables be declared with length a multiple of 4, or preferably 8. This is a *requirement* for variables that are components of derived types, since it has been observed that without these restriction, there are occasional word alignment fault errors generated.

2.5.4 Arguments

The `intent` of arguments to subroutines and functions must be explictly specified.

2.5.5 Arithmetic

FMS requires the use of a default real variable kind that is equivalent to IEEE 64-bit floating-point arithmetic.

2.5.6 Constants

Constants shared across packages must never be hardcoded: instead mnemonically useful names are required. This applies to physical constants such as the universal gas constant, gravity, and so on, but also for flags used to select code options. In particular, this coding construct:

```
subroutine advection(flag)
integer, intent(in) :: flag
...
if( flag.EQ.1 )then
    call upwind_advection( ... )
else if( flag.EQ.2 )then
    call smolar_advection( ... )
...
endif
end subroutine advection
...
call advection(1)
```

is forbidden. This should instead be written as:

```
integer, parameter :: UPWIND=1, SMOLAR=2
...
subroutine advection(flag)
integer, intent(in) :: flag
...
if( flag.EQ.UPWIND )then
    call upwind_advection( ... )
else if( flag.EQ.SMOLAR )then
    call smolar_advection( ... )
...
endif
end subroutine advection
...
call advection(UPWIND)
```

2.5.7 Intrinsics

The `f90` language provides a number of intrinsic functions for performing common operations. The use of the standard intrinsics is generally encouraged. The following conditions apply:

1. The generic form of the intrinsic (e.g `max()`) must be used rather than the specific one (e.g `dmax0()`). This permits flexibility to later changes of type.
2. Many of the intrinsic array operations have been found to be poorly optimized for performance (e.g `reshape()`, `matmul()`) since they have to be perfectly general. These must be used with care in code regions that are critical for performance. **(Restate this in performance chapter).**
3. Several older standard intrinsic names have been declared obsolescent, and the current names are preferred (e.g `modulo()` instead of `mod()`, `real()` instead of `float()`).

2.5.8 Deprecated language elements

Deprecated language elements include:

1. `common` blocks. Use module global variables instead.
2. implicit typing: every code unit must be `implicit none` (see Section 2.5.2).
3. `STOP` statements (see Section 2.5.9).

2.5.9 Error exit

Error exit in a parallel environment requires additional care for a graceful exit on all processors. The FMS standard requires that:

1. the `STOP` statement not be used anywhere, including for the scheduled exit, since this may cause one processor to exit, and the others to hang.
2. the exit print an adequate account of its reasons, ID number of the processor where the error occurred, and a call stack traceback if one is accessible, to `stderr`.
3. the error exit return a non-zero status to the operating system, so that job scripts are made aware of a failure.

It is strongly recommended that error exits be made through the `mpp_error` interface, which satisfies all of these conditions.

2.5.10 The `module` statement

1. The module name must be an unambiguous description of the module's function, with a `_mod` extension.

2. The `module` statement must appear on the same line as the module name, i.e, do not use:

   ```
   module &
     module_name
   ```

   This is to be consistent with the dependency analysis performed by `mkmf` outlined in Section 2.3.3.

2.5.11 `use` statements

1. The `use` statement must appear on the same line as the module name, i.e, do not use:

   ```
   use &
     module_name
   ```

   This is to be consistent with the dependency analysis performed by `mkmf` outlined in Section 2.3.3.

2. The `use, only:` clause is *required* so that all imported elements are explicitly declared.

3. Variables imported by a `use` statement must not be modified by the importing module.

4. Modules cannot publish variables and interfaces imported from another module. Thus, each public element of a module is only available through that module. This does not apply to modules in a package, where the package interface may provide all the required interfaces.

2.5.12 Version identifier

Since the FMS is expected to be in constant evolution, each revision being used must have a unique identifier. We use CVS keywords for this purpose. Each module must contain the following lines:

```
character(len=128) :: version = '$Id$'
character(len=128) :: tagname = '$Name$'
```

The first entry returns a unique identifier to a particular revision of the source file. The second entry returns the tag that was used to checkout the code from the CVS repository. The author is expected to make these entries exactly as shown prior to the first import of the code into the repository. Subsequently, CVS will expand the keywords and keep the names current.

Additional information can be included in the `version` and `tagname` strings if desired. In particular, if your are compiling using a file that has been modified from the repository version, this fact should be signalled by adding a string such as "`modified`" to the `version` string.

2.5.13 The logfile

FMS maintains a logfile `logfile.out` that can be used for an exact reconstruction of the source and inputs used in a run. Each module must make an entry in the logfile on initialization. The entry includes the revision information from Section 2.5.12, the contents of namelists, and identifiers for input files used. See Section 2.5.19.

2.5.14 Model fields

1. A *field* is a function of space representing the instantaneous state of a model field.
2. A field is represented by a floating-point array, either declared as such or as a component of a type.
3. Arrays containing fields may be of higher rank, with the extra dimensions representing, say, tracer number, or timestep. These dimensions must *follow* the spatial dimensions.

2.5.15 Memory management

1. The FMS is runtime-configurable, in that all the work arrays are dynamically allocated. Also, the processor count and domain decomposition must be specifiable at runtime.
2. All fields must be allocated on the data domain of the associated decomposition. In particular, the allocation of 3D fields on the global domain is prohibited. This standardization permits all or most of the allocation to be done at initialization, and reduces the use of assumed-size arrays.
3. The considerations related to domain decomposition do not apply to modules that entirely column-oriented and have no horizontal dependencies.

2.5.16 Parallelism

The parallelism discussed here refers to the message-passing between nodes on a cluster. If in addition the user has access to on-node shared memory parallelism, this can easily be applied on top of the existing parallelism interfaces in thread-safe regions of code (Section 3.2).

1. All parallel processing is done through the `mpp_mod`, `mpp_domains_mod`, and `mpp_io_mod` interfaces.
2. Domain decomposition is generally in the horizontal only. For the logically rectilinear grids (see Section 4.1.1) under consideration, most inter-processor communication can be formulated as halo updates and data transposes, both of which can be handled `mpp_update_domains` procedure. It is anticipated that direct use of `mpp_transmit` should rarely be necessary.

The parallel processing modules are documented in Chapter 5.
The MPP layers are designed to work on single processors with negligible overhead.

2.5.17 I/O

I/O must flow through the standard I/O packages provided by FMS (the diagnostics manager, the FMS I/O manager, or the MPP I/O layer underlying these). In particular, the direct use of Fortran I/O or other I/O APIs for opening and closing files is forbidden.

A particularly dangerous practice is using Fortran I/O units without checking if they are already in use. The use of the FMS I/O standard interfaces prevents this.

2.5.18 Procedural interfaces

Procedural interfaces are the public interfaces to subroutines and functions provided by a module.

1. Procedures that perform the same function on different datatypes (e.g of differing type, kind or rank) must have a single generic interface. When the generic public interface exists, all the module procedures that constitute it must be private.

2. Optional arguments, if any, should *follow* the required arguments, so that the procedure may be called without explicit argument keywords.

3. Argument lists should be short. If necessary related elements of an argument list should be encapsulated in a public derived type.

2.5.19 Module constructor

Each module or package must have an initialization procedure called a *constructor*. It is generally called once in the course of the run.

1. The constructor is conventionally a *subroutine* named `module_name_init`.

2. The constructor may be responsible for allocating global storage.

3. The constructor reads the input data, if any is required. This includes namelists, restart files and any other data files. In every case, the constructor must be capable of generating internal defaults if the input file is not present. It must terminate gracefully if it is neither capable of proceeding without an input file, nor of generating internal defaults.

4. The constructor writes entries to the logfile `logfile.out` so that the model output contains a permanent record of the exact state of the code that was used to generate it (see Section 2.5.13). The FMS I/O package returns the unit number `stdlog` for this. Entries include the version identifier (see Section 2.5.12), the namelist contents, and identifiers for any input files.

5. There must be a private logical variable in the module generally called `module_name_initialized` that is initialized at runtime to `.FALSE.` and is set to `.TRUE.` by the constructor. All module procedures can subsequently check if the module had been properly initialized. If the constructor is called and this variable is `.TRUE.`, it must exit cleanly and silently.

6. The constructor must attempt to call the constructor of each module that it `use`s.

The constructor may be omitted from a module if none of the initialization functions described here (accessing input data, allocating storage, logging) are required.

2.5.20 Module destructor

It is recommended that each module have a termination procedure called a *destructor*. It terminates use of the module, not of the run. It is generally called once in the course of the run.

1. The destructor is by convention a *subroutine* named `module_name_end`.

2. The destructor is responsible for deallocating module global storage.

3. The destructor closes any open files associated with the module.

4. The variable `module_name_initialized` (see Section 2.5.19) is set to `.FALSE.` by the destructor.

5. Restart files save the state of a module upon exit. The destructor is responsible for the writing of restarts. Restart files are written in full 64-bit precision to preserve the bitwise exact model state. These are currently being written in fortran unformatted I/O in IEEE64 format, and will eventually also be written in netCDF.

2.6 Coding style recommendations

Style is somewhat personal, and it would be needlessly restrictive to attempt to impose style requirements. These are recommendations which we believe will lead pleasant interactions for developers with clear, legible and understandable code. The only style requirement we place is that of `consistency`: a single code unit is required to be rigorous in using the author's preferred set of stylistic attributes. It is not onerous to follow a style: modern editors have many language-aware features designed to produce a consistent, customizable style.

Style recommendations include the following:

1. The use of free format;

2. The use of do...end do constructs (as opposed to numbered loops as in Fortran-IV);

3. The use of proper indentation of loops and blocks;

4. The liberal use of blank lines to delimit code blocks;
5. The use of comment lines of dashes or dots to delimit procedures;
6. The use of useful descriptive names for physically meaningful variables; short conventional names for iterators (e.g `(i,j,k)` for spatial grid indices);
7. The use of uppercase for constants (parameters), lowercase for variables;
8. The use of verbose syntax on `end` statements (e.g `subroutine sub...end subroutine sub` rather than `subroutine sub...end`);
9. The use of short comments on the same line to identify variables; longer comments in well-delineated blocks to describe what a portion of code is doing;
10. Compact code units: long procedures should be split up if possible. 200 lines is a rule-of-thumb procedure length limit.

2.7 Module documentation standard

There are three categories of documentation:

Internal documentation consists of comments in the code, expected to be reasonably descriptive but terse. These include:

1. Descriptions of module and interface functionality, including brief descriptions of interface arguments.
2. Descriptions of important internal variables.
3. Frequent comments before sections of code.

User guide This is external documentation distributed alongside the code. This section of the Manual describes in more detail the standards and conventions to be followed by a user guide.

Technical and scientific documentation This contains a technical and/or mathematical description of the process or algorithm being solved and should be referenced by the user guide. It may take the form of a scientific paper. As described in Section 2.4.4, these may be in PDF or PostScript, with PDF preferred.

Each FMS module is required to have a user guide, with the exception of modules that are always invoked as part of a package (Section 2.4.3).

2.7.1 Language

The user guide documentation is written in HTML. A standard format is required as it is automatically processed by scripts.

2.7.2 Sections

Sections MUST be delimited by the following HTML comments:

```
<!-- BEGIN section_name -->
... section text ...
<!-- END section_name -->
```

The `section_name` must be in uppercase.

The section names are given below (items marked with an asterisk are *required*). An HTML anchor should be placed before the section title. The form of this anchor should be:

```
<A NAME="SECTION NAME">
```

where the section name should be all capitals (see the list of section names below). The section titles should not appear between the delimiters.

HEADER * module name, contact person, tags link (see template)

OVERVIEW * A brief description of what the module does.

DESCRIPTION * A more detailed description of what the module does, including links to technical/scientific documentation.

OTHER MODULES USED A list of other modules used. It is recommended that the list also include a version number that the module was tested with.

PUBLIC INTERFACE * A brief description of the entire public interface. This includes all public data and routines. Should also mention whether a namelist interface exists, if data sets are needed, and how any restart data might be used. One line summaries should suffice here. (This section could include much more information???)

PUBLIC DATA A detailed description of all public data and data types (includes units, variable types, and dimensions).

PUBLIC ROUTINES A detailed description of public routines and operators (all arguments must be described including their units, type, and dimensions).

NAMELIST A detailed descriptions of all namelist variables (includes units, type, and default value).

DIAGNOSTIC FIELDS A list of possible netcdf diagnostic output fields (includes short name, units, and description).

DATA SETS Data sets used.

CHANGE HISTORY * Link to the CVS log history for this module.

ERROR MESSAGES A list of all error messages in this module with a brief description and solution of the error.

REFERENCES A list of references and/or link to technical documentation.

COMPILER SPECIFICS A list of compiler recommendations (might include recommended compiler version or optimization options for a particular system).

PRECOMPILER OPTIONS A list of precompiler options.

LOADER OPTIONS A list of loader options (e.g., libraries) and/or recommendations (note that this may be machine dependent).

KNOWN BUGS A list of known bugs.

NOTES Developer notes.

FUTURE PLANS Future plans.

2.7.3 Hyperlinks

Hyperlinks within a document or across documents follow these rules:

- Links to documents that are not in the current directory must have a fully-qualified URL.
- Links to documents that are in the current directory must include the filename.
- Links to the current document that are between section delimiters must also include the current filename.

2.7.4 Embedded scripts

The use of embedded scripts is forbidden. This includes:

1. dynamic HTML;
2. Java and Javascript;
3. server-side scripts, with the exception of webCVS.

2.7.5 Style

As for the source, we do not place stringent style requirements for documentation, except to require consistency. Issues specific to HTML files:

1. Browsers vary widely in their adherence to standards, so the HTML standard itself is not much use. Testing on different browsers is recommended;
2. Leave as much as possible of the choice of fonts and sizes to the reader;
3. Use cascading stylesheets to provide a uniform look and feel across multiple HTML files. FMS stylesheets are stored in a separate directory and must be invoked in the HTML header using a fully-qualified URL.

2.7.6 Template

A template[3] is provided for simple generation of conformant user guide HTM documentation. Steps to be followed using the template are:

1. Enter an appropriate name for contact person (line 25).
2. Change the string `sample` to the name of your module (lines 3, 21).
3. Complete the URL to the WebCVS Log link (line 28) and Change History link (line 147) by appending the path in the CVS repository to the source file. For example, for `mpp.F90`, you would change the default:

   ```
   http://www.gfdl.gov/fms-cgi-bin/cvsweb.cgi/FMS/
   ```

   to:

   ```
   http://www.gfdl.gov/fms-cgi-bin/cvsweb.cgi/FMS/shared/mpp/mpp.F90
   ```

4. Remove sections that you do not expect to use if they are not applicable to your source file.

[3] http://www.gfdl.gov/~fms/gfdl/sample.html

Chapter 3

Coding for performance

As high performance is a vital consideration, we provide guidelines for coding that may assist in writing efficient code. These need not be followed in routines that are not critical to the overall performance of the code.

3.1 Memory management

The bulk of the memory is taken up with 3D fields. Care must be taken for efficient use of memory for module fields. This section deals with module internal fields and work arrays.

There are several performance considerations to keep in mind in designing a memory management strategy.

- Calls to allocate and deallocate memory from the heap can be expensive as they often require system calls.[1] Calls to allocate and deallocate from the stack (such as automatic arrays) can be fast, but stack overflows generally overflow to the heap if the request is larger than the available stack.

- Putting all arrays in static memory may inflate memory usage beyond practical limit. Besides, it contradicts a requirement of runtime-configurability.

In light of these, two strategies suggest themselves. One is for modules to allocate all or most of the required memory at initialization. Workspace can be managed through the use of simple *user stacks*, which are initialized by the module constructor (Section 2.5.19) and reused. Examples of user stacks are available in the modules in the MPP package, as described in Chapter 5. In particular, we use very simple stack management, where there is no pervasive storage in the stack: each call can use all of the stack, and all of the stack is considered to be released on exit from the call.

[1] If a process has once allocated memory to itself and then deallocated, that portion of memory can generally be reused without a system call to assign another memory arena. This optimization is however not guaranteed on all platforms; and besides is only useful if subsequent allocations fit within the present one.

3.2 Thread Safety

Scalable architectures can be divided into two broad classes: *distributed memory*, where processors each have independent memory hardware, and *shared memory*, where many processors have read and write access to the same physical memory. We can think of MPI and OpenMP as the canonical programming paradigms for these two architectural types. Increasingly, a hybrid architecture is becoming a basic form, usually called the *"cluster of SMPs"*, where a group of shared memory nodes operates as distributed memory cluster.

A basic distributed memory programming model usually can target all these types of architectures, and that is the approach followed in FMS. We have not chosen to recast our models in a hybrid programming paradigm nesting the MPI and OpenMP approaches, choosing instead to limit software complexity.

Instead, the design choice in FMS is to have clearly demarcated regions of code where distributed memory parallelism and shared memory parallelism are evoked. The basic parallel construct is the horizontal domain decomposition outlined above in Section 2.5.16. Inside each of these parallel regions, we may invoke shared memory parallelism in regions of code which are known to contain no horizontal dependencies, if the underlying architecture is known to deliver significant increases in performance or scalability with a shared memory programming model.

An example where this approach might be followed is in a spectral atmospheric model. The spectral transform method for distributed memory typically reaches its limit of scalable efficiency in a 1D decomposition (longitudinal for the FFTs, latitudinal for the Legende transforms). The decomposition is longitudinal when data is in grid space. The bulk of the computation is in grid space and is generally taken up in column physics routines (Section 4.2). Since these routines have no horizontal data dependencies, it is possible to parallelize further in this region of code using shared memory parallelism.

Which brings us to the issue of *"thread safety"*. This is the somewhat imprecise term used to describe the organization of memory to allow multiple execution threads to use a shared region of memory, typically through OpenMP or equivalent directives. The key issue is to distinguish memory addresses as being private to a thread, or shared across threads (which users of earlier Cray parallel vector syntax may remember as `task common` and `global common`). *Thread safety* is the careful sorting of variables into *thread-private* and *thread-shared*, and careful control of how shared variables are updated. It is best in general to avoid updating of shared variables, and if it must be done, the code must be done in *critical regions* where multiple threads cannot create a race condition.

The thread safety considerations proposed for FMS include:

- Only routines with no horizontal dependencies (e.g column physics) are permitted to have shared memory threads. Typically, use is restricted to the column physics routines described in Section 4.2.

- Global storage in these routines is never updated by a shared memory thread: any variable that must be updated by one of these routines must be passed through an argument list.

- No I/O is performed from shared memory threads (beyond simple notes to `stdout` and error messages).

More detailed thread-safety guidelines are provided in Section 4.2.

3.3 Pointers

The use of pointers in `f90` is a subject of much debate. In general, the use of `f90` pointers may be detrimental to performance, as it inhibits optimization. However, the standard itself requires dynamic arrays encapsulated within derived types to have the `pointer` attribute. This is now widely recognized within the community to have been an error in the standard: and future revisions of the standard will permit such arrays to have an `allocatable` attribute. This in fact is already available as an extension in many compilers, but not widely enough to be usable here.

FMS is also required to be parsed by automatic source-to-source differentiation tools to generate adjoint and tangent linear models for data assimilation. It has been found that the use of `f90` pointers places insuperable demands on automatic differentiation. However, many of these can be overcome by placing a restriction that `f90` poonters be static, i.e, once assigned, they will never be redirected. FMS adopts this restriction for code segments subject to automatic differentiation. For code segments violating this restriction, the developer is required to provide adjoint code so that automatic differentiation for that code segment may be avoided. The adjoint requirement is not currently in force, but will be shortly.

Another style of pointer of much utility is the Cray pointer. This is outside the `f90` standard, but is universally available on compilers on designed for high-performance, including all the major scalable and vector system native compilers, as well as several compilers designed for HPC on Linux clusters. Its utility is in avoiding memory-to-memory copies, and in writing interfaces interoperable with C. Cray pointers are used in FMS in performance-critical low-level utilities such as the communication kernels (Chapter 5).

Chapter 4

Interfaces for component models

Component models are defined as the model code for each climate subsystem, specifically: atmosphere, ocean, land surface, sea ice, etc. They are divided into a core which treats resolved-scale dynamics (referred to as the *dynamical core*) and a set of parameterized representations of unresolved phenomena (referred to as the *physics routines*).

This chapter specifies standards for component models. The standards are divided into a section specifying general standards for representing physical information, a section specifying standards for writing column physics (mainly relevant for atmospheric models); the interface specification for how component models will call column physics; and finally sections specifying the interface between specific component models and the coupled model driver.

4.1 General representation of physical information

4.1.1 Horizontal grid

The state of a component model is represented at any instant on a *logically rectilinear* grid. The horizontal coordinates are permitted to be physically curvilinear. For coupled model experiments, the coupler places restrictions on the *span*[1] of coupled model elements:

1. The span of the `atmos` component is defined as the model's global domain.
2. The span of the planetary surface (the union of `land` and `ice`[2] components) must equal the atmospheric span. In addition there must be no holes and no overlaps between the components of the planetary surface.
3. The `ocean` component must exactly underlie the `ice` component.
4. If any span is incomplete, it must patch in data for the uncovered region. The coupler assumes that each horizontal grid it sees supplies all the data required for coupling.

[1] The **span** is defined as the physical area of the planetary surface covered by a component.

[2] The component model covering the ocean surface is conventionally referred to as the `ice` component.

There are no restrictions on the span of solo component models.

4.1.2 Vertical grid

The vertical grid specification of a component model is generally internal to that model. Its specification will be described separately in the specification of component model cores.

4.1.3 Boundary state vector

Each component model has a *boundary state vector* that contains all information about its instantaneous internal state that may be required by the coupler. The requirements of the interface state vector that each component model must provide are specified in Section 4.3.

4.1.4 Timestepping

1. Explicit or implicit timestepping may be used by any component model. This places certain restrictions on the treatment of fluxes between component models, described below in Section 4.3.

2. The coupling timestep must be an integral multiple of each of the component model timesteps;

3. The order of calls to component models in the main program may depend on which of the component models has the shortest timestep.

4.1.5 Units

FMS uses SI units.

4.2 Column Physics

Historically, the first attempt to write down a formal FMS design specification began with the 1D column physics specification. While that document continues to be separately available, many of its recommendations have now become part of the general programming standards for FMS (Section 2.5). Considerations specific to the column physics routines are outlined here.

4.2.1 Definition of column physics

Various unresolved and other phenomena are represented in models through *column physics* routines, which operate entirely on vertical columns of data. The column orientation is a distinguishing feature in that the model domain decomposition on scalable architectures has a horizontal orientation (Section 2.5.16).

4.2.2 Horizontal grid

Column physics routines in FMS are constructed to operate with no knowledge of the parallel decomposition of the model. The main computational routine must be thread-safe (Section 3.2).

1. All model fields modified by a column physics routine share the same 2D horizontal grid.
2. The standard latitude (-π/2, π/2) and standard longitude (0, 2π) location of each column is passed in arguments which are arrays on this grid.
3. There are no horizontal data dependencies (in the limit, the routine must be able to operate on a single vertical column).
4. The horizontal grid indices `(i,j)` must be the two innermost indices on any model field.
5. If lateral boundary conditions are required by the column computations, they, and the locations of the grid faces, are passed in arrays with one extra `(i,j)` grid point in each horizontal direction. There are no current cases requiring lateral boundary conditions.

4.2.3 Vertical grid

1. The vertical grid index `k` must be the 3rd index of any model field of rank 3 or higher. Other indices (e.g tracer number, timestep) must follow `k`.
2. Vertical positions are numbered from top to bottom, with points nearest the ground having the highest value of `k`.
3. Vertical grid locations are passed as arrays on the column physics grid. Vertical grid faces, if required, are passed as arrays with one extra `k` location. An optional mask array can be passed to identify non-existent grid locations (e.g locations below the topography in a non-terrain-following atmospheric vertical co-ordinate).

4.2.4 Shared physical quantities

Some physical quantities (e.g moisture) are shared between multiple parameterizations.

1. Shared physical quantities are passed in and out as optional arguments to avoid redundant computations.
2. All quantities are in SI units, as in all of FMS. Dimensionless tracer concentrations, by convention, are in “specific humidity”-like notation, not “mixing ratio”-like.

4.2.5 Procedural interfaces

There are three basic types of public procedural interfaces, the constructor, destructor, and main computational routine.

1. The constructor and destructor perform all the functions outlined earlier in Section 2.5.19 and Section 2.5.20, principally I/O and memory management. They are not required to be thread-safe. They will never be called within a code region that might be partitioned across shared-memory execution threads.

2. The main computational routine performs the basic calculation associated with the physics parameterization. The result is returned as either a time tendency or an adjustment.

   This routine is required to be thread-safe, as it might potentially be called in a code region partitioned across shared-memory execution threads. Partitioning is only in the horizontal. This places some restrictions on the actions such a routine might perform:

   (a) No I/O may be performed, barring simple notes, or error messages. The main I/O activity must be performed by the constructor and destructor. If diagnostic I/O must be performed during the run, it must be from a separate diagnostic I/O routine called from outside the threaded region.

   (b) Shared memory locations may not be written to by this routine. This applies to scalar variables: however, model fields may be updated, as their horizontal indices are distributed among the shared memory threads, and are guaranteed to have no dependencies.

4.3 Coupler

Chapter 5

The MPP modules

Bibliography

Metcalf, M., and J. Reid, 1999. *Fortran 90/95 Explained*. 2nd ed. Oxford University Press.