

CCSM Coupler Architecture
Version 6, the “Next Generation” Coupler

DRAFT June 2001 DRAFT

CCSM Coupler 6 Development Team

Community Climate System Model
National Center for Atmospheric Research, Boulder, CO
<http://www.cgd.ucar.edu/csm/models>

CVS tag \$Name: \$ Build date: June 21, 2001

Contents

1	Synopsis	3
2	Requirements	3
2.1	Scientific Requirements	3
2.1.1	Component Model Control	3
2.1.2	Flexibility	3
2.1.3	Extensibility	3
2.1.4	Time Stepping Method	4
2.1.5	Computations	4
2.1.6	Grids	4
2.1.7	Fault Detection	4
2.2	Computational Requirements	5
2.2.1	Target Platforms	5
2.2.2	Language	5
2.2.3	Parallel Implementation	5
2.2.4	Parallel Reproducibility	5
2.2.5	Sequencing	5
2.2.6	Performance	5
2.2.7	Single Vs. Multiple Executables	6
2.2.8	Processor Allocation	6
2.2.9	Communications	6
2.2.10	Mapping	6
2.2.11	I/O	6
2.2.12	Fault Detection	7
2.3	Imposed Functionality Requirements	7
2.3.1	Single vs. Multiple Executable	7
2.3.2	Fault Detection	7
2.3.3	Handshaking	7
3	System overview	8
4	Design considerations	9
4.1	The coupler, MCT, and MPH	9
4.2	Processor layout	9
4.3	Single vs. multiple executables	11
4.4	Control and sequencing	11
4.5	F90 source code	12
4.6	Staged delivery development	13
4.7	Future Considerations	13
4.7.1	SPMD Configuration	13
4.7.2	Component References	13
4.7.3	Wrapper Code Functionality	13

5	System architecture	14
5.1	Layering strategy	14
5.2	Layer 1a: Command and control	15
5.2.1	Main event loop and sequencing	15
5.2.2	Control module	16
5.2.3	Data declarations	17
5.3	Layer 1b: major subsystems and modules	17
5.3.1	Mapping module	17
5.3.2	Message passing module	17
5.3.3	Merge module	18
5.3.4	Flux module	18
5.3.5	Restart module	18
5.3.6	History module	18
5.3.7	Diagnostic module	18
5.4	Layer 1c: Shared data types and utilities	18
5.4.1	Data type definitions	18
5.4.2	Calendar module	19
5.5	Layers 2-5: MCT and other external libraries	20
5.6	MCT wrapper code provided to component models	20
5.7	Distributed and threaded multi-processing	21
6	Review Status	21
7	Glossary	21

1 Synopsis

The primary function of the CCSM coupler is to coordinate the execution of, and interactions between, the various component models which comprise a single configuration of the CCSM. Critical functionality in the coupler includes synchronization of the time integration of the coupled system, managing inter-model data communication, conservatively mapping data between the various model grids, and computing certain interfacial fluxes between components. The coupler must perform these functions whenever and wherever performance, software engineering, or scientific requirements dictate. The design and implementation of CCSM coupler version 6, the “next generation” coupler, will represent significant advances over previous coupler implementations, in particular, the CCSM and PCM couplers.

The primary motivations for creating a new coupler are (1) to create a single coupler that will replace the existing PCM and CCSM couplers, thus unifying the PCM and CCSM modeling efforts, and (2) to support the creation of a performance-portable CCSM, one which can execute efficiently on a range of platforms, in a variety of configurations. Thus, the coupler requirements should clearly address the goal of delivering performance portability across scalable parallel supercomputers, commodity clusters, shared memory multiprocessors, and vector supercomputers.

Continuing the naming sequence familiar to the CCSM community, this coupler is called "cpl6", or "the CCSM coupler, version six." Among developers, cpl6 is also known as the "next generation coupler" or "ngc."

2 Requirements

2.1 Scientific Requirements

2.1.1 Component Model Control

The coupler steers the execution of the components of the CCSM, and coordinates and supports the transfer of data between the components. Each component model is unaware of the timestep or grid used by the other component models, and the component models communicate only through the coupler at a specified communication interval.

2.1.2 Flexibility

The coupler’s interfaces to the component models must be well defined, such that existing component models can be swapped for new models, or benign substitutes, in a reasonably straightforward manner.

2.1.3 Extensibility

The coupler must allow the user to modify (add/delete/change) the types and amounts of data which are being exchanged between the coupler and the com-

ponent models, or add a new model to the configuration, with minimal changes to the coupler code.

2.1.4 Time Stepping Method

The coupler is required to support either process split or time split advancement of the component models. This requirement is closely associated with the requirement contained in Section 2.2.5.

2.1.5 Computations

- **Fluxes** The coupler is required to compute and combine interfacial fluxes, as needed, among the various component models.
- **Mapping** The coupler must perform the necessary mapping operations required to transform data between different component model grids. Each individual component model is unaware about the grid structures of the other component models. When mapping fluxes, conservation must be insured (using the LANL SCRIP package or similar software).
- **Merging** When data is being prepared for a destination grid, the coupler must be able to merge (spatially average) data originating from multiple source grids. For example, a flux to the atmosphere, within an atmospheric grid cell, may be composed of a combination of the flux from the land, sea ice, and ocean models.
- **Time-Accumulation and Time-Averaging** The coupler must be able to do time accumulation and time averaging of data during the integration for subsequent distribution to other component models. This may be necessary for flux conservation when different models use different communication intervals.
- **Diagnostics** The coupler must be able to perform and report basic diagnostic calculations, such as spatial and temporal averages of the quantities exchanged between component models.

2.1.6 Grids

The coupler is required to support general grid shapes (not just logically rectangular grids). The coupler must have the ability to work around masked areas.

2.1.7 Fault Detection

The coupler must perform an initial coherence check to insure that all component models are set-up in the proper manner for the expected integration. The coupler must allow periodic coherence checks to assure proper synchronization. In situations where the component models fail, the coupler is required to have a mechanism to detect the failure and shutdown the entire system.

2.2 Computational Requirements

2.2.1 Target Platforms

The target platform for the coupler design will be clusters of RISC based multiprocessors. This architecture is a superset of traditional one processor per node MPPs (Cray-T3E) and large scalable shared memory systems (SGI Origin 2000). Examples of such clusters include the IBM SP and Compaq ES-40 clusters. Achieving high performance on both vector and RISC processors is complicated. Experience has shown that in many circumstances redundant pieces of code must be generated, which will necessarily slow the coupler development effort. Therefore, the goal of achieving efficient vector processing performance, although desirable, is not a requirement at this time.

2.2.2 Language

All of the general purpose libraries written for the coupler are required to have a Fortran 90 API or a Fortran 90 module interface. All physics packages written explicitly for the coupler are required to be written in Fortran 90.

2.2.3 Parallel Implementation

The coupler is required to give the user control in selecting pure message passing, pure shared memory multithreading, or hybrid parallel algorithms to take advantage of different platforms and the relative performance of parallel modes on those platforms.

2.2.4 Parallel Reproducibility

Under certain conditions, bit-for-bit reproducibility is required under the different parallel implementations listed in Section 2.2.3. At a minimum the coupler must give bitwise reproducibility when a statically loaded executable (same model with identical linked libraries) is rerun on the same machine at the same site on the same number of processors. It is noted that a strict requirement for bitwise parallel reproducibility may have a negative impact on performance.

2.2.5 Sequencing

The coupler functions will support both sequential and concurrent execution of the component models, in a fashion such that the user can configure the model/coupler with the sequencing method which makes the most sense for a particular problem/platform. This requirement is closely associated with the requirement contained in Section 2.1.4.

2.2.6 Performance

At a minimum the coupler is required to meet or exceed the performance of the current (March 2000) Climate System Model, Version 1.2 and Parallel Climate

Model, Version 1.0 when configured (component model resolutions and communication intervals) similarly. That is, for concurrent execution, the coupler must at least outperform the most time intensive component model in terms of seconds per model day; for sequential execution, the coupler must not comprise more than 20% of the entire CCSM execution time. It is noted that the user can select a model/coupler configuration which easily can violate this requirement (such as configuring a shorter coupling interval requiring more mappings and communication).

2.2.7 Single Vs. Multiple Executables

The coupler is required to accommodate executing in an environment where the coupler and the component models comprise a single executable, or where the coupler and each component model run as separate executables. It is worth noting that single vs. multiple executables is independent of the sequencing method selected (e.g., concurrent execution can be accomplished with a single executable).

2.2.8 Processor Allocation

For flexibility, the coupler must be capable of running decomposed into an arbitrary numbers of processors. This is transparent to the component models.

2.2.9 Communications

Only communications between the coupler and the component models are allowed. Component model to component model communications are disallowed. Functionally, n coupler nodes should be able to communicate in parallel with m model nodes, for an arbitrary (n,m). Parallel communications between coupler and component models should be explored, developed and optimized, if possible.

2.2.10 Mapping

The regridded must be capable of performing arbitrary regriddings; i.e., any mapping from a source grid vector into a destination grid vector, representable by a transformation matrix, should be supported. The regridded must be capable of performing fast, efficient regriddings in parallel either through threading, or message passing, or both. The regridded must be able to perform a masked regridding operation.

2.2.11 I/O

The coupler must be able to read/write restart datasets that insure CCSM bit-for-bit restartability, and write history datasets consisting of user selected sets and subsets of data being exchanged between the component models. If possible, the NCAR/Unidata I/O Library will be used.

2.2.12 Fault Detection

Similar to the scientific functionality requirement in Section 2.1.7, the coupler is required to have a mechanism which will recognize in a rudimentary fashion that a fatal computational problem has arisen and perform the necessary operations to shutdown the entire model system.

2.3 Imposed Functionality Requirements

In order to meet the desired requirements of the coupler, it is apparent that the component models will need to have a priori knowledge of, and must be able to fulfill, some coupler requirements. A list of possible requirements of the component models, imposed by the coupler requirements, is given in this section. Note that the reverse is also true - the models may impose a requirement of the coupler. Any requirements of this latter type are unknown at the present time, but this document recognizes and notes the possibility.

2.3.1 Single vs. Multiple Executable

The component models are required to make accommodations for system configurations of either a single or multiple executable running environment. Furthermore, when the system is configured as concurrently running component models within a single executable, the component models must have the flexibility to execute within a designated processor space.

2.3.2 Fault Detection

The coupler is required to detect and/or react to a system problem, and therefore must receive appropriate fault detection handshaking signals from the component models. Thus, the component models are required to provide such signals. The component models may also be required to detect a coupler problem and act appropriately.

2.3.3 Handshaking

Other handshaking requirements between the coupler and the component models include:

- interface design
- on-command AUTO-restart writing capability
- possibly use the CCSM shared code (physical constants, orbital parameters, etc.)
- calendar manager

This list is certainly not all-inclusive.

3 System overview

The coupler component is a central component in the CCSM framework. This framework divides the complete climate system into component models connected by a coupler. Currently the CCSM design couples four component models – atmosphere, land, ocean, and sea-ice – although the system could be configured with more or fewer components. Each component is connected to the coupler, and each exchanges data with the coupler only (Figure 1). The data exchanged consists of 2d flux fields or 2d state fields necessary to compute flux fields. From a software engineering point of view, the CCSM is not a particular climate model, but a framework for building and testing various configurations of climate models. More than any particular component model, the coupler defines the high-level design of the CCSM framework.

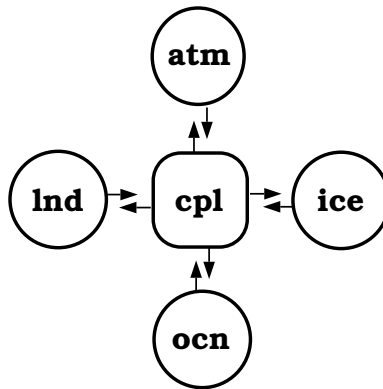


Figure 1: The CCSM Coupled Model Framework in its most basic form.

The coupler code has several key functions within the CCSM framework:

- It allows the CCSM to be broken down into separate components, atmosphere, sea-ice, land, and ocean, that are "plugged into" the coupler. Each component model is a separate code that is free to choose its own spatial resolution and time step. Individual components can be created, modified, or replaced without necessitating code changes in other components.
- It controls the execution and time evolution of the complete CCSM by controlling the exchange of information between the various components.
- It routes component model interfacial fluxes (2d fields) between all component models while insuring the conservation of fluxed quantities. It may also be required to compute certain interfacial fluxes.

4 Design considerations

4.1 The coupler, MCT, and MPH

A major consideration for cpl6 software design is that it will be built upon the infrastructure provided by MCT and MPH. The coupler falls in the "coupler application" layer shown in Figure 2. The coupler is built upon MCT, which is in turn built upon other lower-level libraries.

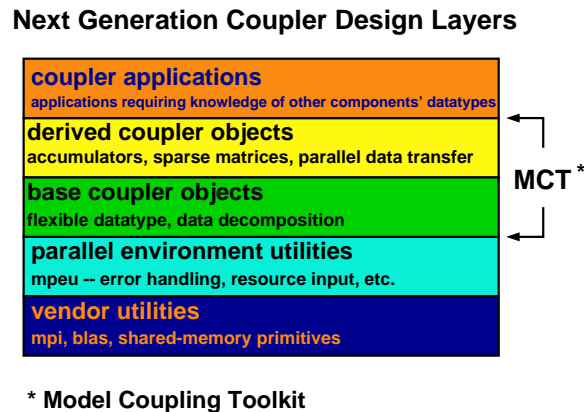


Figure 2: Software Design Layers.

Cpl6, MCT, and MPH are products of the same project, are to be developed concurrently, and are intended to work together according to this layering strategy. MCT is a library of lower level data types and routines that perform basic data transfer operations commonly required by coupled climate models. MPH is a library that initializes MPI communicator groups for distinguishing between inter and intra-model communication patterns. MCT's and MPH's intended usage includes, but is not limited to, software like the coupler.

It is important to note that the MCT layer in Figure 2 does not insulate the coupler application layer from lower layers – software in the coupler application layer may be required to interact directly with software found in all lower layers (for example MPEU and MPI).

4.2 Processor layout

The coupler is required to coordinate the execution of several major climate model components. It is important to be clear about how these components are to be arranged in processor space.

Figure 3 shows four possible processor allocation strategies:

- (a) In this strategy all processors are assigned to all component models and the models do not run concurrently. This arrangement has been used by PCM codes.

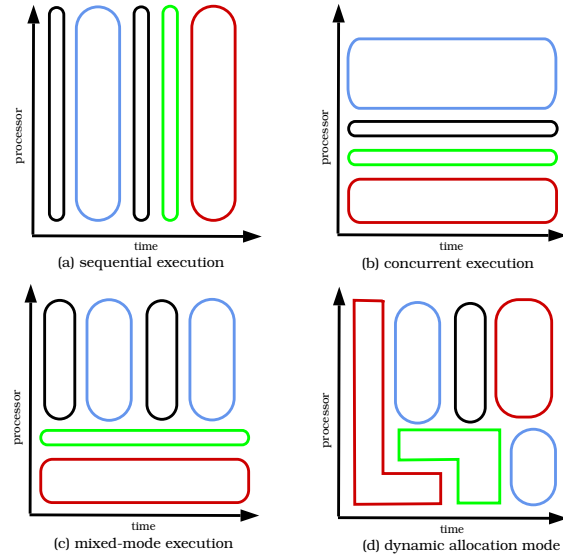


Figure 3: Processor Allocation Strategies

- (b) In this strategy disjoint sets of processors are assigned to the component models and the models run concurrently. This arrangement has been used by CCSM codes.
- (c) This represents a mixture of (a) and (b).
- (d) This is a mode the CSM has used in the past, but its implementation depended upon a shared memory machine with a robust scheduler (*e.g.* a C90) and component codes that were threaded but not distributed.

Strategy (d) represents the most flexible processor allocation strategy, but it's no longer valid to assume we have shared memory machines, robust schedulers, and component codes that were threaded but not distributed. Without being able to make these assumptions, implementing this strategy would likely be exceptionally difficult and thus is not a realistic option for this project. Both (a) and (b) have presented performance problems in the past, (a) because it requires all component models and the coupler to scale well to arbitrarily high processor counts (and this has not been true), and (b) because it assumes that all models can be made to run concurrently (and this has been difficult to achieve). This leaves (c), which is the strategy we will pursue because it seems reasonable to achieve and should minimize the performance problems experienced by (a) and (b). Further, (a) and (b) are merely a special, restricted case of (c).

4.3 Single vs. multiple executables

One of the major requirements of the new coupler is that the coupled system as a whole (the coupler together with all component models) be able to execute in both single and multiple executable modes (SPMD and MPMD). Theoretically, there is no advantage in performance by choosing one mode over the other. However, implementing the SPMD mode in cpl6 would impose on component model development teams additional code changes (i.e., creation of a subprogram interface). Because all existing component models already operate in MPMD mode within the existing CCSM framework, and because we suspect it is easier to explore the various load balancing options in MPMD mode, the cpl6 architecture will defer the goal of creating a production CCSM running in SPMD mode.

Furthermore, while SPMD would allow data movement thru argument lists at the subroutine API, we have decided that data movement between component models will always be via message passing (e.g., MPI) because it is required inherently in MPMD mode and because it is considerably more flexible in terms of where in a code structure data can be sent and received. We recognize that argument list data movement would be more efficient within SPMD, but it is desirable to retain the data movement as message passing between component models (message passing could be made to emulate argument lists of a subroutine API).

Finally, the use of MPMD mode leaves open the possibility of distributing the coupled system across multiple machines, although there are currently no plans or requirements for doing so.

4.4 Control and sequencing

The requirements specify that the coupler will control the execution of the coupled system and control the flow of data between the component models. In past versions of the coupler, two major restrictions have been imposed on component models: (1) that they can only communicate with the coupler, and (2) that they send and receive only one "bundle" of fields per communication interval (the sends and receives are not necessarily adjacent). While these are significant restrictions, they also significantly simplify the design of the coupled system. This design has worked well in the past and we see no need to relax these restrictions at this point.

Figure 4 shows how data moves between components in the coupled system. Because all the component models communicate only with the coupler, and because the component model execution can only proceed when the coupler sends them needed forcing data, its easy to see how the coupler naturally controls the flow of data and the timing of component execution for the entire coupled system.

Figure 4 illustrates why operating the coupled system in sequential vs. concurrent mode (assume for the purpose of this discussion that all components in Figure 4 are separate executables) is not something that can be arranged

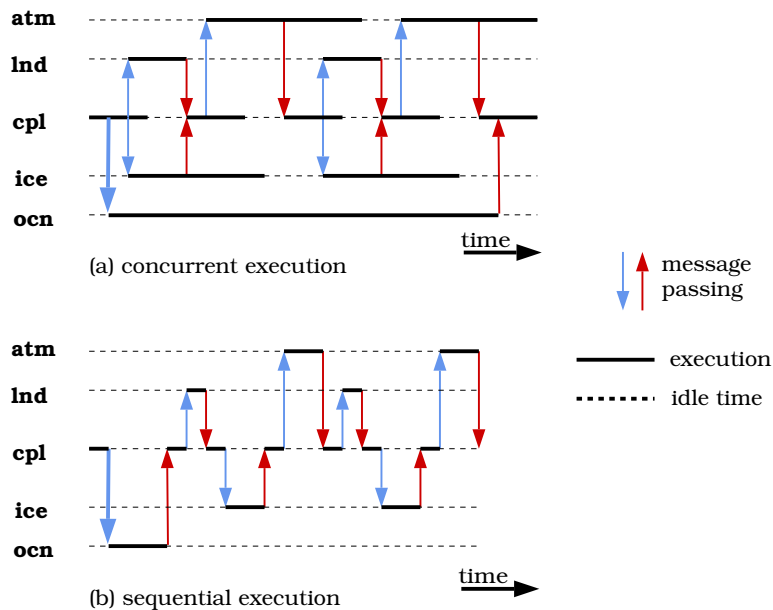


Figure 4: Flow of Data between Components.

solely by the coupler. In (a) component models are shown to execute almost continuously, before and after sends and receives from the coupler, while in (b) models are completely idle between the time they send data to the coupler and the time they receive new data from the coupler. This suggests that the decision about whether the component models should run sequentially or concurrently should be based in part on the code structure of the component models – something that is out of the control of the coupler development team.

Figure 4 also shows why component models that behave like subroutines (in the sense that you send them input data, they perform some computations, they return output data, and then go idle) are less flexible than codes that can send and receive data anywhere in their code flow. Components that behave like subroutines restrict the code flow to (b), sequential execution. It takes the flexibility of message passing to implement concurrent execution as shown in (a), although message passing can certainly implement (b) too.

4.5 F90 source code

It is a requirement that this coupler provide an F90 API and that all physics packages written for the coupler be written in F90. Further, all existing component models with which the coupler must interact are written in F90. In light of these considerations, this coupler will also be written in F90.

It is a priority to create code that is clean, simple, understandable, and maintainable. The design should take full advantage of F90 features such as user

data types, module interfaces, etc. Care should be taken not to use constructs that significantly degrade performance, for example, as heavy use of array syntax might do.

The coupler source code will contain ProTex keywords so that its internal API can be automatically generated (see <http://dao.gsfc.nasa.gov/software/protex>).

4.6 Staged delivery development

The development plans calls for a staged delivery life cycle. Thus the architecture should facilitate a minimal set of critical functionalities to being implemented and tested in the early versions of the code, allowing developers to defer the implementation of other less critical functionalities to later in the development cycle.

4.7 Future Considerations

A number of desirable coupler features have been mentioned in previous sections but cannot be implemented currently because of time constraints which limit testing. They are itemized below to provide a reference for future upgrades and/or modifications.

4.7.1 SPMD Configuration

As stated in section 4.3, the creation of SPMD mode (the PCM configuration) has been deferred. The primary reason for the delay is to allow the implications imposed upon independent component models to be explored. For example, each component model would be required to provide a subprogram interface. Such an interface has yet to be described, and would have to be built in collaboration with the model developers.

4.7.2 Component References

The current coupler architecture contains an explicit expectation that the specific component models participating in a calculation are known in both number and type. This assumption limits the extensibility of the coupler to new, as yet unknown, components. An internal coupler mechanism should be developed to generalize the the number and type of components.

One possibility is to create an infrastructure to support the concept of a "component world" similar to that of the MPI communicator concept.

Of course, at the highest coupler level (for example, an instantiation of an event loop in the main coupler program) the user must have explicit knowledge of the number and types of participating components.

4.7.3 Wrapper Code Functionality

The interface to the component models is limited currently to direct communication with the coupler itself. The functionality could be expanded to include,

for example, mapping and/or direct component-to-component routing.

5 System architecture

5.1 Layering strategy

As described earlier, the cpl6 software can be thought of as a "coupler application" that is built upon the infrastructure provide by the MCT and MPH libraries. In this document we will treat MCT and MPH as externally developed libraries, similar to vendor libraries, and introduce a new layering description that focuses on the architectural details above the MCT level.

Figure 5 illustrates this view, grouping MCT, MPEU, etc., together with the vendor supplied libraries into the lowest layer. Layers 1a, 1b, and 1c are the central subject of this document, along with the "MCT wrapper code," which is not code that is compiled as part of cpl6, but rather is provided to component model developers to include into the component model (atmosphere, ice, etc.,) source code. The component models will use MCT and MPH when communicating with the coupler, but their usage will be hidden within the MCT wrapper module.

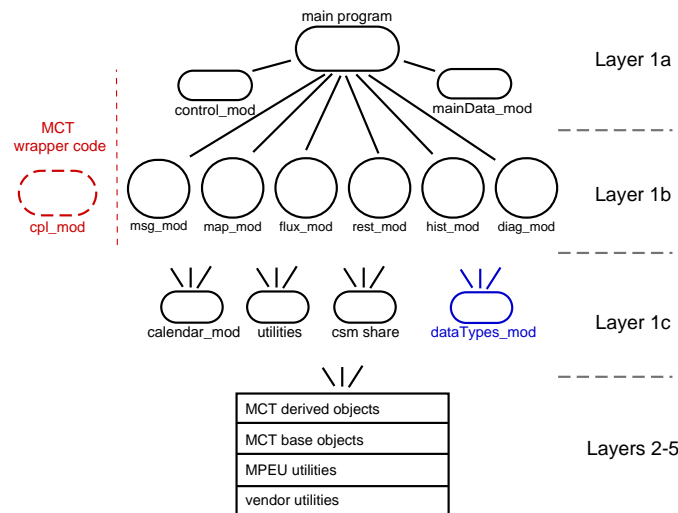


Figure 5: The coupler layering strategy. Layers 1a, 1b, and ac are the "coupler application" described in this document. This application is built upon existing software provided in layers 2-5. The MCT wrapper code is provided to component models to hide the use of MCT and MPH.

Layer 1a consists of the main program, all high-level command and control functionality, and the declarations of the data fields that flow thru the coupler. Reading the code of the main program should give a general, high-level description of how the program executes.

Layer 1b consists of various autonomous modules that implement the major subsystems of cpl6. There is no interaction between modules in layer 1b – a layer 1b module can be added or removed without impacting other layer 1b modules.

The MCT wrapper is unique in that its routines are not in the coupler calling tree, but rather its routines are provided to component models. Its routines provide API's using standard, intrinsic F90 data types and hide the use of MCT and MPH. Invoking its routines ultimately results in MPI data transfer with the coupler's message module. Because it is built upon layers 1c and below, it could be considered a a layer 1b module.

Layer 1c consists of various lower-level utility routines that might be used in layer 1a, 1b or 1c. This is a code re-use strategy for lower-level functionality. For example, wrappers for machine-dependent system calls would be found here, as would a calendar module. Layer 1c also defines fundamental cpl6 user-defined data types in `dataTypes_mod.F90`. If a data object is passed from one module to another and that data object is a user-defined data type, the definition of that data type would be found here.

Following is a list of the major modules found in cpl6. They are discussed in more detail in the following sections.

- Layer 1a
 - `main.F90` main program and event loop
 - `control_mod.F90` sets and manipulates control flags
 - `mainData_mod.F90` declares bundles, routers, matrices
- Layer 1b
 - `msg_mod.F90` does inter-model message passing
 - `map_mod.F90` does mapping (matrix multiplies)
 - `flux_mod.F90` does flux computations
 - `rest_mod.F90` reads/writes restart files
 - `hist_mod.F90` creates history files
 - `diag_mod.F90` creates diagnostic data
- Layer 1c
 - `dataTypes_mod.F90` defines cpl6 data types domain, bundle, map, etc.
 - `calendar_mod.F90` provides calendar information
 - `shr_sys_mod.F90` code shared with other CSM components
 - `globalData_mod.F90` convenient, global access to MPH information
 - more as needed*
- MCT wrapper
 - `cpl_mod.F90` wrapper code provided to component models

5.2 Layer 1a: Command and control

5.2.1 Main event loop and sequencing

The central feature of the main program is an event loop that represents the coupled system advancing in time. The cpl6 layer 1b subsystems are created

and invoked as necessary to perform the required scientific functionality. The sequence in which layer 1b subsystems are invoked can be rearranged to manipulate computational characteristics of the coupled system. For example, the following two pseudo-code fragments illustrate how the same subsystem calls can be reordered to implement sequential and concurrent model execution.

```
!--- example of sequential execution ---
while (continue_integration == .true.) do
  [...]
  call msg(router_c2a,bundle_c2a) ! send data to atm
  call msg(router_a2c,bundle_a2c) ! recv data from atm
  call msg(router_c2o,bundle_c2o) ! send data to ocn
  call msg(router_o2c,bundle_o2c) ! recv data from ocn
  call history()                  ! create cpl history file
  call diagnos()                  ! compute cpl diagnostics
  [...]
  call control()                  ! update control flags
end while

!--- example of concurrent execution ---
while (continue_integration == .true.) do
  [...]
  call msg(router_c2a,bundle_c2a) ! send data to atm
  call msg(router_c2o,bundle_c2o) ! send data to ocn
  call history()                  ! create cpl history file
  call diagnos()                  ! compute cpl diagnostics
  call msg(router_a2c,bundle_a2c) ! recv data from atm
  call msg(router_o2c,bundle_o2c) ! recv data from ocn
  [...]
  call control()                  ! update control flags
end while
```

For concurrent execution, experience has shown that one can often rearrange the sequence of calls to layer 1b subsystems to improve and help optimize the processor load balance of the entire coupled system. Notice how the coupler/model interaction in the sequential execution configuration, namely the adjacent sends and receives, closely resembles a subroutine API.

To support staged delivery, some subsystems, for example history and diagnostic subsystems, can be completely omitted from early versions of the code.

5.2.2 Control module

Most of the layer 1b modules will need control logic that determines how they function. For example, a restart module will need to know how often restart files are created, what the restart file names are, and a mapping module will need to know the names of files that contain the mapping data. Each layer 1b module will contain and utilize a control flag data structure that contains this type of

information. The values of the control flag data will be set and manipulated in the control module, `control_mod.F90`.

A subsystem's control flags are manipulated in the control module, rather than in the subsystem module, so that a more control uniform logic can be applied. For example, it will be easier to coordinate the creation of restart, history, and diagnostic data if the logic for triggering the data creation is located in one module, rather than being scattered in three different modules.

We envision the subsystem control flags containing very basic information, *e.g.* "create data now", or "output file name is *XYZ.nc*," while the control module contains the logic to handle more complex issues, such as whether to create data once per month, every 10 day, once per year, at the start or end of a run, etc. The control module will also handle user input, such as reading namelist files.

5.2.3 Data declarations

Any data objects that need to pass from one module to another will be declared in the module `mainData_mod.F90`. These would include, for example, bundles of 2d fields exchanged with component models and control flags required by restart and history modules. This data could be declared at the beginning of the main program, but there is likely to be quite a long list of declarations, so they are separated into a data declaration module.

5.3 Layer 1b: major subsystems and modules

5.3.1 Mapping module

The mapping module, `map_mod.F90`, will contain all code specific to mapping data between grids. At the core of this functionality is a generic sparse matrix multiply. Additional functionality includes reading in mapping matrix data files, checking the matrix data for erroneous values, and doing a data copy instead of a matrix multiply if the matrix is an identity matrix.

5.3.2 Message passing module

Here "message passing" refers to the inter-model exchange of data, not the intra-model exchange of data. Given a bundle of data fields and a router, the message passing module, `msg_mod.F90`, will invoke the MCT routines necessary to do the data transfer. Additional functionality will include calling the MPH routines that establish MPI communicator groups for the various component models, calling MCT routines that establish segment maps and routers for the various required coupler/model communications, and logic that verifies that the coupler and component models are in agreement with respect to the exact list of fields that will be exchanged.

5.3.3 Merge module

The merge module, `mrg_mod.F90`, will assemble into one bundle, all the fields destined for a particular component model. Merging does not involve regridding, thus when preparing a bundle of fields for the (*e.g.*) ocean model, input to the merge module will consist of various bundles of fields that are already on the ocean model grid.

5.3.4 Flux module

The coupler may be required to compute certain fluxes. If it is required to do so, the routines that do the flux calculation will be located here the flux module, `flux_mod.F90`.

5.3.5 Restart module

The restart module, `rest_mod.F90`, implement the reading and writing of restart files. These restart files will be in binary format.

5.3.6 History module

The history module, `hist_mod.F90`, implement the creation of history files. These history files will be in netCDF format.

5.3.7 Diagnostic module

The diagnostic module, `diag_mod.F90`, will compute diagnostic quantities as necessary. Diagnostics generally would not be result in 2d fields, rather they would be global and time integrated quantities, (*e.g.*) the global, annual average heat flux into the ocean. Such quantities are useful for verifying that the coupler is conserving fluxed quantities and identifying simulations that are drifting into unrealistic states.

5.4 Layer 1c: Shared data types and utilities

5.4.1 Data type definitions

In addition to the standard intrinsic F90 data types, the `cpl6` code will make use of a small set of user defined F90 data types. These data types will be defined in the `dataTypes_mod.F90` file. In general, any derived type data objects that pass between two separate F90 modules should define the data type in `dataTypes_mod.F90`, whereas any data types that are used within one F90

module only should be defined in that module.

```
type domain
  type(mct_GeneralGrid),pointer :: grid ! x,y coordinates,area,mask
  type(mct_globalSegMap)       :: gsmap ! associated decomposition
end type domain

type bundle
  type(mct_AttrVect)           :: data ! list of fields with a common domain
  type(domain),pointer        :: domain ! their common domain
end type bundle

type map
  type(mct_SparseMatrix) :: matrix ! mapping matrix data
  type(domain),pointer   :: src    ! the map's source domain
  type(domain),pointer   :: dest   ! the map's destination domain
end type map

type control_flags
  ! ??? mpeu_list objects ???
  ! vector of string data & associated vector of string descriptions
  ! vector of integer data & associated vector of string descriptions
  ! vector of real data & associated vector of string descriptions
end type control_flags
```

Cpl6 datatypes are built using datatypes from MCT. Some understanding of the functions and datatypes in MCT is required to build and use datatypes in cpl6. See, for example:

```
type mpeu_list
type mct_attrVect
type mct_globalSegMap
type mct_generaGrid
type mct_sparseMatrix
type mct_router
```

5.4.2 Calendar module

The coupler has a frequent need for determining how many days are in a month or year, or (for example) how many days there are between two given dates. The calendar module, `calendar_mod.F90` will contain this functionality. The coupler code should be arranged so that only code in the calendar manager changes if, for example, one switches from a calendar that has no leap year to one that does.

5.5 Layers 2-5: MCT and other external libraries

Layers 2-5 consists of vendor supplied utility libraries such as MPI, netCDF, and system calls. For the purposes of this document, MCT and MPH will be considered "vendor supplied" that will be utilized, but whose design will not be described. As noted earlier, MCT and MPH were in fact developed in conjunction with cpl6, and developed explicitly to support the creation of cpl6. Certain data types defined within MCT and MPH may be pervasive in the coupler code.

Most code originating in layers 2-5 can be identified the standard prefix naming convention. MCT and MPH routines and data types, for example, can be identified by the "mct_" and "mph_" prefixes. Other heavily used external libraries include MPI and netCDF, these too can be identified by their "mpi_" and "nc_" prefixes. CSM shared code, code shared between the coupler and various component models, also uses the standard prefix, "shr_".

5.6 MCT wrapper code provided to component models

The use of this coupler will require component models to communicate with the coupler via MCT, with the help of MPH. Calls to MCT and MPH libraries will need to be compiled into the component model executables. Wrapper modules will be provided to component model development teams, these modules will hide the use of MCT and much of the complexity associated with component/coupler communication (see Figure 6).

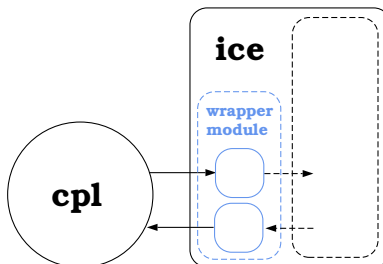


Figure 6: Wrapper modules will be provide for all component models. The use of MCT, MPH, and many other communication complexities can be hidden within this wrapper module.

The API for routines in the wrapper module will only require the use of intrinsic F90 data types. Several routines will be necessary in the wrapper module, for example, routines for initializing the message passing environment, and routines sending and receiving bundles of data during time integration.

While not part of the coupler itself, this wrapper code can be thought of as a layer 1b module. This is because it is not involved in command and control, it will use utilities found in layers 1c and below, and it functions as an autonomous subsystem within a component model.

While there are currently no plans to do so, it has been considered that this wrapper code could contain more functionality than basic data transfer. For example, mapping functionality could be put into the wrapper code, thus allowing a component model to send data to the coupler that is not on the component model's native grid. Putting such functionality into the wrapper code may create additional opportunities for load balancing. Also, putting such functionality into the wrapper code will hide it from component model developers so that they needn't be concerned about the more complex load balancing strategy.

5.7 Distributed and threaded multi-processing

TBD, issues wrt running in distributed & threaded modes such as having a master process do all i/o, including netCDF, using OMP directives for optional threaded regions.

6 Review Status

Architecture Review

Review Date: <Date>

Reviewers:	<Reviewer>	<Institution>
	<Reviewer>	<Institution>
	<Reviewer>	<Institution>

7 Glossary

bundle Several data fields, all of which share the same domain, grouped together in one data structure.

combine Add together two or more fields that are on the same domain; e.g., latent heat flux plus sensible heat flux on a T42 grid with the same land mask

communication interval The time interval at which a component model exchanges data with the coupler.

concurrent execution Running two or more coupled system components at the same time.

domain Coordinates arrays together with mask, cell area, and processor decomposition information.

grid Coordinate arrays without a mask, cell area, decomposition information; e.g., a T42 grid.

hybrid parallel A combination of messaging passing and threading.

map Regrid; move a data field from one domain to another.

mask True/false flags associated with a specific grid; e.g., a land mask.

merge Take two or more fields of the same type (e.g., land, ocean, and ice surface temperatures), which are all on the same domain (e.g., all on a T42 grid, global domain), and the same number of corresponding cell fractions (fraction of land, ocean, and ice, which together sum to unity), and create one field (e.g., (merged surface temperature) = (ice frac)*(ice T) + (ocn frac)*(ocn T) + (lnd frac)*(lnd T)).

process split Time stepping strategy whereby when component models integrate thru a time interval, say [t1,t2], the only data available to the models corresponds to a previous time interval, [t0,t1]; generally associated with concurrent component model execution.

reentrant A characteristic of a function which can be called concurrently by multiple threads without mishap.

regrid Map; move a data field from one domain to another.

sequential execution Running only one coupled system component at a time.

time split Time stepping strategy whereby when component models integrate thru a time interval, say [t1,t2], the data made available to some models may correspond to the time interval [t1,t2]; generally associated with sequential component model execution.

timestep A model's internal integration increment in time.