The CCSM Coupler
Version 6.0

— DRAFT —

User's Guide, Source Code Reference, and Scientific Description

Brian G. Kauffman[1]
Robert Jacob[2]
Tony Craig[1]
William G. Large[1]

June 23 2004

Community Climate System Model
http://www.cesm.ucar.edu/

---

[1] National Center for Atmospheric Research, PO Box 3000, Boulder CO 80307
[2] Argonne National Laboratory, Argonne, IL

Contents

1 Preface

Version 6.0 of the CCSM Coupler, cpl6, is included with version 3 of the Community Climate System Model. The name "cpl6" is meant to reflect its place in the lineage of CCSM couplers but cpl6 is very different from its predecessors. Like cpl5 and earlier versions, cpl6 includes a specific instance of a coupler `main` program to implement CCSM3's coupling scheme. But new with this version, cpl6 also includes a class library of Fortran90 modules which provide a flexible and uniform way to construct the Coupler `main` program and interface it to the other components of CCSM. Along with more flexibility, cpl6 allows the Coupler to be run as a distributed memory parallel application.

This guide is divided into three parts. Part I is the traditional User's Guide and describes how to use the Coupler `main` included within CCSM3, modify its runtime options, and understand its output. Part II, the Source Code Reference, describes the structure of the Coupler `main` and also describes the objects and methods of cpl6 and how to use them to modify or construct a new coupled system. Part III contains the scientific description of the Coupler including details on the physical calculations performed in the Coupler.

Part I
User's Guide

2 Introduction

This part of the Coupler documentation provides information needed for users of the Coupler within CCSM3. This includes how to build and run the Coupler. It also provides a complete list of the *namelist* parameters which can be set by the user and guide to the output produced by the Coupler.

3 Building and Running the Coupler

The CCSM system consists of 5 separate executables, one of which is the Coupler. A series of c-shell scripts and GNU Makefiles are used to build CCSM including the Coupler. Usage of the scripts that build CCSM is described in detail in the CCSM 3.0 User's Guide, and the Coupler user should consult this guide for information on building the complete system.

The CCSM Coupler requires two libraries, the Model Coupling Toolkit (MCT) and MPH3. These libraries are distributed with CCSM and are compiled before the Coupler within the CCSM3 build system. These libraries must compile successfully for the Coupler to complete it's build.

The Coupler can only execute within the complete CCSM system. In CCSM3, this requires atmosphere, ice, land, and ocean models along with the Coupler. There is no "standalone" Coupler. Again, the CCSM3.0 User's Guide should be consulted for information on how to run CCSM3. CCSM3 allows several possible combinations of active and data models to be executed as a coupled system with the Coupler. The simplest system to run the Coupler with is to use all data models for the ocean, atmosphere, land and sea ice. This will still result in a 5 executable system but the data models are very simple and each run on only 1 processor. For advanced testing, an all-dead models case can be run with the Coupler. Unlike the data models, the dead models do not produce physically realistic fields but they can be run on any number of processors. An all-dead-model case is useful for testing the scalability of the model-coupler communication routines.

The Coupler can run on any number of processors. After creating a case (See CCSM3 Users Guide), edit the *env_mach.$MACH* file and change `ntasks_cpl` to change the number of MPI processors the Coupler runs on. `nthrds_cpl` should always be left as 1 because the Coupler currently has no thread-level parallelism.

4 Input Namelist Parameters

To some extent, the behavior of the Coupler can be specified or modified at run time. This is done almost exclusively through the use of an F90 namelist. This section contains a list and description of available namelist input parameters. The Coupler reads this namelist from a data file, called cpl.nml, at runtime.

Many of the Coupler namelist values are set as part of creating a *case* within CCSM (See the CCSM3 Users Guide). Most namelist variables which can be changed by the user within an already existing case are in the *env_run* file in the it $CASE directory. Remaining variables can be changed by editing `Buildnml_Prestage/cpl.buildnml_prestage.csh` in the *$CASE* directory.

4.1 Conventions

Coupler input variables follow a naming convention in which the first few letters of the variable identify a basic functionality group or "event":

- `case_xxx` selects options with respect to specifying a case name and description
- `start_xxx` selects options with respect to selecting restart files or the simulation start date.
- `rest_xxx` selects options with respect to when restart files are created
- `stop_xxx` selects options with respect to when a simulation will stop
- `hist_xxx` selects options with respect to when and what type of history data is created
- `diag_xxx` selects options with respect to run-time diagnostics of the physical simulation,
- `flx_xxx` selects options with respect to flux calculation specifics
- `orb_xxx` selects options with respect to solar orbit calculations
- `info_xxx` selects options with respect to monitoring the progress or computational performance of the model

The following list of input namelist parameters includes this information:

TYPE: the variable's data type. **Note:** some character strings have `length = CL`. "CL" is a single global constant used throughout the Coupler to define a "long" character string. The CL is hard-coded to be 256. If necessary, a one line change to the source code will lengthen all such character variables.

DEFAULT: the default value of the variable. While reasonable default values are selected when possible, generally users need to alter some of these values according to their application.

REQUIRED: tells if and when the variable is required input. Very few input parameters are required. In a few cases, changing one default value will cause other input parameters to become required.

DESCRIPTION: a brief description of the purpose and effect of the parameter.

EXAMPLE: a reference to an example namelist which uses the variable (examples follow this list).

4.1.1 Generic periodic event specification.

Many namelist variables are used to specify periodic events such as restart file creation, history file creation, creation of diagnostics data, etc. All such periodic events are triggered by the same underlying calendar/clock/alarm functionality and thus they all have the same set of namelist variables. This general functionality and the related variables is described here, and then referred to below as necessary. `<event>` refers to one of the functional groups listed above (**case start** etc.).

<event>_option

TYPE: `character(len=32)`
DEFAULT: <varies, see the particular namelist variable>
REQUIRED: <varies, see the particular namelist variable>
DESCRIPTION: This is the option that selects how often an event will occur. When an event happens on a day, it happens at the start of the day.
OPTIONS:

`"date"`	occurs only once, on the given date
`"daily"`	occurs every day
`"yearly"`	occurs on every January 1st
`"monthly"`	occurs on the 1st of every month
`"ndays"`	occurs every n days relative to an offset date
`"nmonths"`	occurs every n months relative to an offset date
`"never"`	never occurs

EXAMPLES: `start_option stop_option`
If `<event>_option = "date"` then namelist parameters `<event>_date` is required to specify the date (see below).
If `<event>_option = "ndays"` or `nmonths`, then namelist parameters `<event>_n` and `<event>_date` are both required (see below), to specify n and the offset date, respectively.

<event>_n

TYPE: integer
DEFAULT: <varies, see the particular namelist variable>
REQUIRED: maybe, only used if event_option = "ndays" or "nmonths"
DESCRIPTION:
EXAMPLES: `rest_n stop_n`
If `<event>_option = "ndays"`, the event will occur every n days, where = <event>_n.
If `<event>_option = "nmonths"`, the event will occur every n months, where $n = event\_n$.

But in either case, the exact timing of the event is not yet completely specified, one must specify an "offset date" (see below), such that an event occurs on the offset date and every n days (or n months) before or after the offset date. It is not necessary that a simulation ever encounters this offset date.

There is a special case that occurs when, for example, `event_option = "monthly"`, `event_n = 1`, `event_date = 31`. Apparently the event should occur on February 31st, a non-existent date. In such a case the event will occur on the last day of the month. Thus the events will occur on January 31st, February 28th, March 31st, April 30th, May 31st, etc.

<event>_date
TYPE: integer
DEFAULT: 0
REQUIRED: no
DESCRIPTION:
EXAMPLES: `start_date stop_date`
If `<event>_option = "date"`, then this is the date when the event will occur.

If `<event>_option = "ndays"` or `"nmonths"`, then this is the offset date described above. This offset date must either be a valid calendar date, encoded: yyyymmdd, or, if *event_date* < 1, the offset date is taken to be the starting date of the simulation (this is the default value).

4.2 List of Parameters

case_name
TYPE: `character(len=CL)`
DEFAULT: `"unset"`
REQUIRED: no, but highly recommended
DESCRIPTION: This is the case name text string which is used to create output file names and is also included in output files to help identify the model run. Because this variable is used to construct file names, it must contain only those characters that are valid in unix file names. While the name can be quite long, it is recommended that it be rather short, for example, 8 to 16 characters.
See Example: 1, 2, 3

case_desc
TYPE: `character(len=CL)`
DEFAULT: `"unset"`
REQUIRED: no, but highly recommended
DESCRIPTION: This is a short text string (typically less than 80 chars) which is included in output files to help identify the model run.
See Example: 1, 2, 3

start_type TYPE: `character(len=16)`
DEFAULT: `"initial"`
REQUIRED: no (but default is of limited usefulness)
DESCRIPTION: This selects the run type. Valid choices are: `"initial"`, `"continue"` or `"branch`. Selecting "branch" makes `start_bfile` a required input.
See Example: 1, 2, 3

start_pfile
TYPE: `character(len=CL)`
DEFAULT: `$HOME/cpl6.<case_name>.rpointer` or `./rpointer` if `case_name` is unspecified.

REQUIRED: no
DESCRIPTION: This is the complete path and name of the restart "pointer file." This must include an existing, NFS mounted directory. All run types will update this file (and create it, if necessary), but only a continuation run requires that this file exists prior to the start of the run.
See Example: 1, 2, 3

start_bfile
TYPE: `character(len=CL)`
DEFAULT: `"unset"`
REQUIRED: yes, if `start_type = "branch"`, ignored otherwise.
DESCRIPTION: This is the file name of the "branch file" (the IC data file). Note that a prefix like "mss:" is used to indicate a file archival device, Valid prefix options are:

"cp:" or no-prefix indicates a normal unix file copy from an NFS mounted file system.

"mss:" indicates a file on NCAR's MSS

"null:" indicates no archival – the file name, stripped of any directory information, indicates a file in the current working directory.

See Example: 3

start_date
TYPE: `integer`
DEFAULT: 00010101 (January 1st, year 1, encoded yyyymmdd)
REQUIRED: no (ignored for "continue" and "branch" runs)
DESCRIPTION: This is the start date for "initial" runs

On "inital" runs , `start_date` is the initial date of the simulation.

On "branch" runs , the start date will be the date found in the `start_bfile` file, this is the IC/restart file.

On "continue" runs , the start date will be the date found in the restart file.

See Example: 1

rest_option
TYPE: `character(len=32)`
DEFAULT: `"monthly"`
REQUIRED: no
DESCRIPTION: This is the restart option that selects how often restart The generic periodic event specification (Sec. 4.1.1) applies to this option with the exception is that a restart file will never be created at the start of a run.
See Example: 1, 2, 3

rest_n
TYPE: `integer`
DEFAULT: 3
REQUIRED: maybe (only used if `rest_option = "ndays"` or `"nmonths"`)
DESCRIPTION: The generic periodic event specification (Sec. 4.1.1) applies to this option.
See Example: 2

rest_date
TYPE: `integer`
DEFAULT: 0
REQUIRED: no
DESCRIPTION: The generic periodic event specification (Sec. 4.1.1) applies to this option.
See Example: 3

stop_option
TYPE: `character(len=32)`
DEFAULT: `"monthly"`
REQUIRED: no
DESCRIPTION: This is the stop option that selects when the simulation The generic periodic event specification (Sec. 4.1.1) applies to this option with the exception that every run must be at least two days long. Also note that if `stop_option = "date"`, and the the given date is before the start of the model run, the model will stop with an error message.
See Example: 1, 2, 3

stop_n
TYPE: `integer`
DEFAULT: 3
REQUIRED: maybe (only used if `stop_option = "ndays"` or `"nmonths"`)
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.
See Example: 2

stop_date
TYPE: `integer`
DEFAULT: 0
REQUIRED: no
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.
See Example: 3

hist_option
TYPE: `character(len=32)`
DEFAULT: `"monthly"`
REQUIRED: no
DESCRIPTION: Selects how often history data files are created. The generic periodic event specification (Sec 4.1.1) applies to this option.
See Example: 2

hist_n
TYPE: `integer`
DEFAULT: 3
REQUIRED: maybe (only used if `hist_option = "ndays"` or `"nmonths"`)
DESCRIPTION:
The generic periodic event specification (Sec 4.1.1) applies to this option.
See Example: 2

hist_date
TYPE: `integer`
DEFAULT: 0

REQUIRED: no
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.

hist_64bit
TYPE: `logical`
DEFAULT: false
REQUIRED: no
DESCRIPTION: If true, history files contain 64-bit binary data, otherwise history files contain 32-bit binary data. This option applies to both instantaneous and time averaged history files.

avHist_option
TYPE: `character(len=32)`
DEFAULT: `"monthly"`
REQUIRED: no
DESCRIPTION: Selects how often time average history data files are created. The generic periodic event specification (Sec 4.1.1) applies to this option. Note: time average history data is averaged over the interval defined by these file creation events. For example, `avHist_option = "monthly"` results in monthly average data and `avHist_option = "yearly"` results in annual average data.

avHist_n
TYPE: `integer`
DEFAULT: 3
REQUIRED: maybe (only used if `avHist_option = "ndays"` or `"nmonths"`)
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.

avHist_date
TYPE: `integer`
DEFAULT: 0
REQUIRED: no
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.

diag_option
TYPE: `character(len=32)`
DEFAULT: `"monthly"`
REQUIRED: no
DESCRIPTION: Selects how often instantaneous diagnostics data is written to stdout. The generic periodic event specification (Sec 4.1.1) applies to this option.
See Example: 2

diag_n
TYPE: `integer`
DEFAULT: 3
REQUIRED: maybe (only used if `diag_option = "ndays"` or `"nmonths"`)
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.
See Example: 2

diag_date
TYPE: `integer`
DEFAULT: 0
REQUIRED: no
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.

avDiag_option
TYPE: `character(len=32)`
DEFAULT: `"monthly"`
REQUIRED: no
DESCRIPTION: Selects how often time averaged diagnostics data is written to stdout. The generic periodic event specification (Sec 4.1.1) applies to this option.

avDiag_n
TYPE: `integer`
DEFAULT: 3
REQUIRED: maybe (only used if `diag_option = "ndays"` or `"nmonths"`)
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.

avDiag_date
TYPE: `integer`
DEFAULT: 0
REQUIRED: no
DESCRIPTION: The generic periodic event specification (Sec 4.1.1) applies to this option.

flx_albav
TYPE: `logical`
DEFAULT: `.false.`
REQUIRED: no
DESCRIPTION: The Coupler computes ocean albedos and sometimes modifies ice albedos. See Sections 16.3.2 and 16.3.3.

if flx_albav = true , the ocean albedos are computed such that they have no zenith angle dependence and ice albedos, which are computed by the ice model, are unaltered.

if flx_albav = false , the ocean albedos are computed with a zenith angle dependence and ice albedos are set to unity on the dark side of the earth.

Typically `flx_albav` ("daily average albedos") is only turned on when the atm component is a climatological data atm model that is sending daily average data to the coupler, and thus "daily average albedos" are an appropriate complement to the daily average solar fluxes sent by the atm component.

flx_epbal
TYPE: `character(len=16)`
DEFAULT: `"off"`
REQUIRED: no

DESCRIPTION: Non-default values can be used to conserve globally integrated salinity in ocn and ice components that are coupled to a climatological data atm model. The conservation takes place by multiplying precipitation, P, and runoff, R, by a single scalar (spatially constant) factor f to balance evaporation, E: $\langle E\rangle + f\langle P\rangle + f\langle R\rangle = 0$, where $\langle x\rangle$ denotes a globally averaged value. There are three valid values for `flx_epbal`:

"off" no factor is applied to $P + R$

"inst" the Coupler computes a factor f so that $\langle E\rangle + f\langle P\rangle + f\langle R\rangle = 0$ at each time step ("instantaneously").

"ocn" the ocn component provides the Coupler with a factor f and the Coupler applies this factor to P and R. Typically this factor is chosen, by the ocn component, so that $\langle E\rangle + f\langle P\rangle + f\langle R\rangle = 0$, but perhaps not instantaneously, maybe as an annual average. This can be used to allow some fluctuation in the global average of salinity on shorter time scales while enforcing a constant global average of salinity on longer time scales. The ocn component is responsible for providing an appropriate factor to the Coupler.

orb_year
TYPE: `integer`
DEFAULT: <none>
REQUIRED: yes
DESCRIPTION: This is the calendar year which is used to determine the solar orbit and resulting solar angles. This is necessary, for example, to compute ocn surface albedo, which may be zenith angle dependent. Valid values are in the range $[-1000000, +1000000]$. A typical value might be 1990.
See Example: 1, 2, 3

info_dbug
TYPE: `integer`
DEFAULT: 1
REQUIRED: no
DESCRIPTION: Debugging information level: 0, 1, 2, or 3. Level 1 is recommended, levels 2 and three are generally for debugging purposes.

0: do not write any extra debugging information to stdout

1: write a small amount of extra debugging information to stdout

2: write a medium amount of extra debugging information to stdout

3: write a large amount of extra debugging information to stdout

See Example: 3

info_bitCheck
TYPE: `integer`
DEFAULT: 0 (off)
REQUIRED: no (normally not recommended)
DESCRIPTION: computes and writes to stdout information that can be used when comparing two simulations to see if they are bit-for-bit identical. The information written can verify two runs are NOT bit-for-bit, and it can

strongly suggest two runs are bit-for-bit, but it is insufficient to prove that two runs are bit-for-bit identical.

0: do not compute or write any information

1: compute and write information once per month

2: compute and write information once per day

3: compute and write information every time step

4.3 Example Input Namelists

```
Example 1: a "startup" run

    $inparm
    case_name   = "test.01"
    case_desc   = "testing a startup run "
    start_type   = "initial"
    start_pfile  = "$HOME/cpl6.test.01.rpointer"
    start_date   = 19800101
    rest_option = "monthly"
    stop_option = "monthly"
    orb_year    = 1990
    /
```

Here the inputs specify a `initial` run starting on 1980 Jan 1st and stopping every month, in this case on February 1st, 1980. No initial condition data is needed – "initial" runs don't use any IC data. Restart files will also be created every month. A restart pointer file, `cpl6.test.01.rpointer`, will be created in the user's home directory and will contain the name of the most recently created restart file. History data files and diagnostic data will be created at the default frequencies. Solar orbit calculations will be based on the year 1990.

```
Example 2: a "continuation" run

    $inparm
    case_name   = "test.01"
    case_desc   = "testing a continuation run "
    start_type   = "continue"
    start_pfile  = "$HOME/cpl6.test.01.rpointer"
    stop_option = "nmonths"
    stop_n      = 6
    rest_option = "nmonths"
    stop_n      = 3
    hist_option = "ndays"
    hist_n      = 10
    diag_option = "ndays"
    diag_n      = 10
    orb_year    = 1990
    info_dbug   = 0
    /
```

Here the inputs specify a continuation run. Assuming this run continues from where example 1 finished, this run will start on 1980 February 1st and stop six months later on 1981 August 1st. Exactly where this run continues from is specified by the restart file – the restart file which will be used is specified by the restart pointer file. Restart files will be created every three months. History data and diagnostic data both will be created every 10 days relative to the start of the run. Setting `info_dbug = 0` will minimize the amount of debugging information written to stdout.

```
Example 3: a "branch" run

    $inparm
    case_name   = "test.02"
    case_desc   = "testing a branch run "
    start_type  = "branch"
    start_pfile = "$HOME/cpl6.test.02.rpointer"
    start_bfile = "test.01.cpl6.r.1980-08-01-00000 "
    stop_option = "date"
    stop_date   = 19810101
    rest_option = "date"
    rest_date   = 19810101
    orb_year    = 1990
    info_dbug   = 2
    /
```

Here the input parameter `start_type` specifies a branch run. The branch file (a restart file from a previous run) must be specified. In this case it's expected to be pre-positioned in the execution directory. The start date will be taken from the branch file (apparently 1980 August 1). The simulation will stop on the given date (1981 January 1st). A restart file will also be created on the given date (1981 January 1st). Setting `info_dbug = 2` will increase the amount of debugging information written to stdout.

5 Output Data

The coupler outputs three types of data: standard out (stdout) diagnostic data, restart files, and diagnostic history files.

5.1 Stdout Data

Stdout output consists mostly of brief messages that indicate how the simulation is progressing and whether any error conditions have been detected. Stdout also contains a record of the values of all Coupler input parameters. If global diagnostics have been activated (see the `diag_option` namelist parameter), stdout will also contain some diagnostic information, specifically global averages and time averages of various flux fields flowing through the Coupler.

Exactly where the Coupler's stdout (and stderr, standard error) shows up is determined outside of the Coupler source code and executable. Certain environment variables found in the CCSM run scripts are used to determine where the Coupler will execute (its current working directory, or `cwd`) and also to redirect stdin and stdout – these env variables determine where stdout text will end up. Normally the run script arranges for stdout text to show up in a "log" file in the Coupler's execution directory (current working directory) while the simulation is in progress, and then later, after the simulation ends, the run script moves this log file to an archival location. See the CCSM User's Guide for details.

5.2 Restart Files

Restart files are in a machine dependent binary format, written using standard Fortran write statements. Restart files provide the Coupler with all the IC data necessary to do an "exact restart" of a previous simulation. "Exact restart" means stopping and restarting a simulation while preserving, bit-for-bit, the results that would have be created if the simulation had not been stopped and restarted.

"Continuation" or "branch" runs both require a restart file and are capable of exact restart. For "initial" runs, the Coupler does not use a restart file. Initial runs cannot do exact restarts. The CCSM User's Guide describes the related concepts of "startup", "branch", "hybrid", and "continuations" runs. At the system-wide level there are four types of runs, but for the Coupler alone, there are only three modes of operation. Both "startup" and "hybrid" at the system level invoke an "initial" run in the Coupler component. See the CCSM User's Guide for more information.

Normally there is no need to examine the contents of a restart file. All fields found in the restart file can be saved into history files, which are machine independent NetCDF files.

5.3 History Files

5.3.1 Instantaneous and Time Averaged File

The Coupler can create either instantaneous history file, time averaged history files, or both. Their creation is controlled by two independent sets of namelist variables. When doing an ncdump the contents of both files appear to be the same, but the values in the time coordinate variables (time and time_bound) clearly define the time averaging interval, if any, as required by the NetCDF Climate and Forecast Metadata Conventions, version 1.0 (CF 1.0). The Coupler's NetCDF files conform to the CF 1.0 format.
See http://www.cgd.ucar.edu/cms/eaton/cf-metadata.

5.3.2 History File Format

NetCDF (network Common Data Form) was chosen as the history data format because many common visualization tools already exist that handle this data format. NetCDF is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine independent format for representing scientific data. Together the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado.
See http://www.unidata.ucar.edu/packages/netcdf.

5.3.3 History File Content

Because NetCDF files are self-describing, the most complete and accurate description of the contents of a Coupler history file (or any NetCDF file) will always come from the NetCDF data file itself. The NetCDF tool "ncdump" will generate the CDL text representation of a NetCDF file on standard output, optionally excluding some or all of the variable data in the output.

The Coupler's NetCDF files conform to the CF 1.0 format. The CF conventions generalize and extend the COARDS conventions.
See http://www.cgd.ucar.edu/cms/eaton/cf-metadata.

Three types of data are found in Coupler NetCDF history data files: global attributes, domain data, and two dimensional state and flux data.

(1) Global attributes
This is text information, including the case name corresponding to the history data, and the date the data was created.

(2) Model domain data
This includes the spatial coordinates of the grid cells as well as the cell areas and a domain mask for each surface model. Each model has two sets of latitude and longitude coordinates, one corresponding to grid cell "centers" and one corresponding to grid cell "vertices". Each cell is defined by four vertices which describe a quadrilateral. Grid cell "centers" lie within this polygon, typically near its center.

A state variable $S(i,j)$ is understood to be a point value located at the center of cell (i,j). A flux field $F(i,j)$ can be thought of as a point value located at the cell center, but more accurately it is an area average value that applies uniformly over the entire grid cell.

Currently only the grid cell centers are put onto history data files.

The Coupler deals with five model domains: atmosphere, ice, land, land runoff, and ocean. For each of these model domains, the following domain information is on the history files:

- latitude and longitude – degrees north or east for grid cell centers
- area – the cell area in radians squared. This is the area provided to the coupler by a component model.
- aream – the cell area in radians squared. This is the area provided to the coupler by a mapping data file. The coupler accounts for differences in area and aream.
- cell mask – a value is zero if and only if this is an inactive point. For example, for the land model domain, a cell located in the central Atlantic Ocean would be inactive (because there is no land there), hence its mask would have a zero value there.
- cell index – for an array A(i,j), sized ni * nj, the index n = (j-1)*ni + ni This is useful for identifying a grid cell when decomposed on an MPMD machine.
- process id – This is useful for identifying how a domain was decomposed when the simulation was running on an MPMD machine.

(3) Two dimensional state and flux data

This includes model state variables, component flux fields, and merged input flux fields.

A variable naming convention has been adopted to help organize and identify the literally hundreds of state and flux fields managed by the Coupler.

The cpl6 Coupler manages fields by grouping them into bundles of fields. A bundle is a group of fields that share the same domain. A domain includes grid cell coordinates, mask, areas, and decomposition. More information on these Coupler datatypes can be found in Section 8. The variables found in NetCDF history files are based on the same naming convention as the variables found in the source code. The variable names found in the history file are formed by appending a field name onto a bundle name: <bundle>_<field>

Bundles have names like `Xa2c_a` where the `_a` indicates the bundle's domain is the atm domain and `Xa2c` is a meaningful shorthand description, in this case `Xa2c` means "everything sent from the atm to the cpl." The `X` means "everything", the `a` is for "atm", the `c` is for "cpl."

Fields have names like `Sa_t` or `Faoc_lat` which are also a meaningful shorthand description, in this case `Sa_t` means "atm state, temperature" and `Faoc_lat` means "atm/ocn flux, computed by cpl, latent heat". Additional information on the naming convention for variables in the Coupler is in Section 9.1.

Here are more examples from the Coupler history file:

```
Xa2c_a_Sa_u
   bundle: Xa2c_a    ~ atm-to-cpl bundle on atm domain
   field:  Sa_u      ~ atm meridional velocity
Xc2o_o_Foxx_taux
   bundle: Xc2o_o    ~ cpl-to-ocn bundle on ocn domain
   field : Foxx_taux ~ merged zonal ocn surface stress
domain_o_lat
   bundle: domain_o  ~ time invariant ocn domain information
   field : lat       ~ grid cell latitude
frac_o_ifrac
   bundle: frac_o    ~ surface fractions on ocn domain
   field:  ifrac     ~ ice fraction
```

Each variable in the NetCDF history file has `long_name` and units attributes which further describe the variable. Also see the Data Exchanged section of this document.

Part II
Source Code Reference

6 Introduction

This part of the coupler documentation provides details of the cpl6 source code. This includes where to find the code and build it, variable and file naming conventions, descriptions of the cpl6 version of the Coupler `main` and the objects and methods of cpl6. If a user wishes to modify the coupler, including changing or adding fields transferred between models, this part should be reviewed carefully. Normal users of CCSM3 may skip this part.

One of the reasons cpl6 was created was to allow the Coupler in CCSM to be run as a distributed memory parallel application. Previous versions of the coupler allowed some shared-memory parallelism using OpenMP calls however distributed memory parallelism is sufficiently different and presented so many challenges that it was necessary to build a new coupler from scratch. Some familiarity with parallel programming may be necessary to understand some of the datatypes and methods in cpl6.

7 Source Code Overview

Source code for the coupler is available as part of the CCSM3 distribution at http://www.cesm.ucar.edu/models/ . This distribution includes the source code for all CCSM component models. Documentation for other CCSM component models, as well as input data for running the models, is also available at this site.

The coupler code consists of two parts. The coupler `main` specific to this version of CCSM is located in `.../models/cpl/cpl6/` (All path names are relative to the top level directory created when untarring the CCSM3 source code distribution.) The general-purpose cpl6 datatypes and methods are located in CCSM's location for code shared by all the component models, `.../models/csm_share/cpl`.

7.1 cpl6 programming conventions

The Coupler source code is written entirely using standard Fortran 90. Interfacing with the Coupler requires at least a portion of an interfacing model's code, the part that communicates with the Coupler, to be compiled with a Fortran90 compiler. The Coupler uses a naming convention for source files and the functions they contain. In general, the source code for `cpl_<module name>_<method name>` can be found in `cpl_<module name>_mod.F90`. Files containing `_mod.F90` are Fortran90 modules.

The source files are self documenting with headers and comments written for post-processing with the Protex system. Protex generates LaTeXversions of the header information and allows a close linkage between source and documentation.

The Coupler source code was developed using the CVS revision control system, but only one "tagged" version of the Coupler is available within any CCSM source code distribution. This information can be used to identify the coupler version contained in a particular distribution and is printed as part of the output when the coupler starts.

The Coupler can only be built within CCSM's build system which uses GNUMake. There is no "standalone" coupler. Within the CCSM build system, the coupler is treated as a component like the atmosphere, ocean, etc. and is built automatically with the rest of CCSM3's components. See the CCSM3 User's Guide for more information on the build system.

7.2 The cpl6 version of the CCSM coupler

The source code for the CCSM coupler is located in `.../models/cpl/cpl6`. The directory path follows the CCSM convention: `.../models/cpl` contains all of the couplers that could be used in the CCSM system just like `../models/atm` contains the atmosphere model's available in the system. `.../models/cpl/cpl6` contains the implementation of the CCSM coupler using cpl6 datatypes and methods.

A summary of the modules and subroutines in `.../models/cpl/cpl6` is given below. Additional details can be found below or in Part III for code which handles scientific calculations in the coupler.

`main.F90` Contains all the code to implement the "hub" in the CCSM3 hub-and-spoke, concurrent execution system (See Section 12) including all communication calls with component models and appropriate mapping and flux calculation calls.
(remaining routines are in alphabetical order)

`areafact_mod.F90` Module to hold and calculate grid area ratios. See Section 16.4

`bitCheck_mod.F90` Module with routines to calculate and output statistics for a field at high precision for bit-for-bit checking.

`data_mod.F90` Module to hold and initialize most variables used in `main.F90`

`diag_mod.F90` Module with routines for calculating global diagnostics.

`flux_mod.F90` Module with routines for calculating inter-model fluxes and surface albedos. See Section 16.2

`frac_mod.F90` Module to hold, initialize and update inter-model fractional weights. See Section 16.5

`history_mod.F90` Module with routines to write instantaneous and time average coupler history files.

`merge_mod.F90` Module with routines for merging fluxes between models.

`restart_mod.F90`	Module with routines to read/write Coupler restart files.
`tStamp_mod.F90`	Module with routine to output current model date and time
`timeCheck.F90`	Module with routine to verify/enforce time coordination between models.

Code for many of the Coupler's tasks (diagnostics, flux calculation) are naturally separated into files as above. However we should note that some modules, such as `merge_mod.F90` and `data_mod.F90` were created to prevent `main.F90` from becoming too long.

7.3 Cpl6 General Datatypes and Methods.

Besides a new version of the CCSM coupler `main()`, cpl6 also includes a set of Fortran90 modules consisting of derived datatypes and methods which act on them. The contents of these modules are used to construct the new coupler and provide a uniform, flexible and extendable interface between the Coupler and the component models of CCSM.

The cpl6 modules make extensive use of another set of Fortran90 modules called the Model Coupling Toolkit (MCT). MCT provides generic datatypes and methods for the construction of parallel couplers in a distributed memory parallel application. cpl6 combines and extends many of the datatypes in MCT and provides additional datatypes to meet the specific requirements of the CCSM Coupler. The reader of this section may also need to refer to *The Model Coupler Toolkit API Reference Manual* and the *User's Guide to the Model Coupling Toolkit* for additional information on the concepts underlying cpl6.
See http://www.mcs.anl.gov/mct.

A summary of what each module in cpl6 handles is below in alphabetical order:

`cpl_bundle_mod.F90`	The gridded data values exchanged between model's and the domain associated with that data.
`cpl_comm_mod.F90`	MPI communication groups and model ID's. This module uses methods from the MPH3 library to divide `MPI_COMM_WORLD` into sections for each component.
`cpl_const_mod.F90`	Physical and other constants used in the Coupler.
`cpl_contract_mod.F90`	Information needed to exchange data between models including a bundle and a domain.
`cpl_control_mod.F90`	Integration control and Fortran namelist processing for the Coupler.
`cpl_domain_mod.F90`	Data about the grid (latitude, longitude values, etc.) and its parallel decomposition.

`cpl_fields_mod.F90`	Master list of all fields exchanged between models and the Coupler.
`cpl_infobuf_mod.F90`	Scalar data exchanged between models.
`cpl_interface_mod.F90`	Wrapper routines with simple arguments used to interface between component models and the Coupler.
`cpl_iobin_mod.F90`	Binary I/O methods for the Coupler
`cpl_iocdf_mod.F90`	NetCDF I/O methods for the Coupler
`cpl_kind_mod.F90`	Fortran90 KIND typing for the Coupler
`cpl_map_mod.F90`	Interpolation between different grids
`cpl_mct_mod.F90`	Master list of MCT functions and datatypes used in cpl6

7.4 Summary of CPL6 structure

The software which makes up cpl6 is summarized in Figure 1.

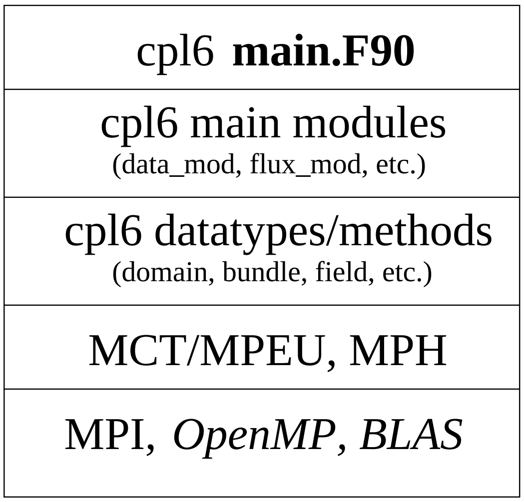

Figure 1: The software structure of cpl6.

The cpl6 `main` is the top level program and is built on top of the software layers beneath it. MPEU, the Message Passing Environment Utilities, is part of MCT. OpenMP and BLAS calls are planned for future releases.

8 Major cpl6 Modules

This section describes some of the cpl6 modules and how they are used in the Coupler and CCSM3. Detailed information on each module and its contents can be found in the *CPL6 API Reference Manual*.

8.1 The `domain` module

One of the fundamental datatypes in cpl6 is the domain which is defined in `cpl_domain_mod.F90`. The domain contains information about the physical grid that a quantity, such as Temperature, is defined on. This includes a descriptive name for the grid, the total number of points and number of points in each horizontal dimension. (Since all the fields exchanged by the Coupler in CCSM3 are two dimensional, the domain currently only supports two-dimensional grids.) Finally, the domain contains information about the how the global grid is decomposed over processors. This information is stored using an MCT datatype called the GlobalSegmentMap. All of this information is identical on each processor the Coupler runs on.

The domain also includes numerical data about the grid such as the latitude and longitude values and grid-cell area but only for points local to the processor. The contents of this component of a domain will thus vary from processor to processor. The values are stored in another MCT datatype, the AttributeVector. The complete list of values saved for each grid point is:

- `lat` latitude value in degrees
- `lon` longitude value in degrees
- `area` the component's (e.g. ocean) value for the area of a grid cell
- `aream` the mapping program's (e.g. SCRIP) value for the area of a grid cell
- `mask` list of active/inactive cells. 0 for inactive cells
- `pid` Processor id (identical for all points)

8.2 The `bundle` module

The bundle is another fundamental datatype in cpl6 and is defined in `cpl_bundle_mod.F90`. A bundle contains the actual values of fields passed in and out of the coupler such as temperature, wind speed, etc. These values are stored as a one-dimensional vector using an MCT AttributeVector. The bundle also contains a pointer to the domain associated with the data in the bundle. Thus all the data in a bundle must be on the same domain however more than one bundle can point to a given domain.

Within the coupler, there are many bundles. Most are named according to what component they're involved with but their content can also be regrouped using methods such as `cpl_bundle_split` according to how the Coupler treats them. For example `bun_Si2c_i` contains all the state data passed from the ice model to the Coupler while `bun_Fi2c_i` contains the flux data.

8.3 The infobuf module

While the bundle is for gridded data, the infobuffer, defined in cpl_infobuf_mod.F90 is used to hold scalar data, integers and reals, exchanged between models and the Coupler. "info" is used because much of the integer data acts as logical control flags so the Coupler can tell a component model to perform actions such as write a history file or do diagnostics or halt execution. The size of the real and integer parts of the infobuffer and the location of each real and integer number in the arrays is the same in all models and is set by parameters in cpl_fields_mod.F90 (See Sec. 8.6). It is not necessary for each model to define every possible value in the infobuffer. For example, the eccentricity of Earth's orbit is set in the atmosphere model and passed to the Coupler through the infobuffer while the other models never receive or set this value.

8.4 The contract module

The contract is a key concept and an important datatype in cpl6 and is defined in cpl_contract_mod.F90. A contract contains all of the information needed for a single model-coupler exchange. Thus there is one contract for the data sent by the atmosphere model to the Coupler and a second contract for the data received by the atmosphere from the Coupler. The main interaction between a model and the Coupler is conceptualized as first setting and then sending or receiving contracts.

To contain all the information needed for a data exchange, the contract contains an infobuffer, a bundle and a domain (which the bundle points to). The contract also contains an MCT datatype called the Router which contains all the information needed to do a parallel data transfer between a distributed memory parallel model running on one set of processors and the distributed memory parallel Coupler running on a different set of processors.

8.5 The interface module

cpl_interface_mod.F90 contains the routines component models and the Coupler call to interact with each other. The interface is closely related to the contract and a contract is an argument in nearly all the interface routines.

The purpose of the interface module is to provide a simple, compact interface for component models to talk to the Coupler *while making no constraints on how the component models represent data internally.* Four methods from cpl_interface_mod.F90 do nearly all the work in allowing a component model to interact with the Coupler:

cpl_interface_init	Initialize the communication infrastructure, e.g. MPI communicator groups.
cpl_interface_contractInit	Initialize a contract with the Coupler. This is where the coupler is told the grid of the component model and what fields will be exchanged.
cpl_interface_contractSend	Send data (bundles and infobuffers) to the Coupler

`cpl_interface_contractRecv` Receive data (bundles and infobuffers) from the Coupler

Along with reducing the needed routines to a handful, the interface module also simplifies the argument list. Aside from the contract, which is a Fortran90 derived datatype, the remaining arguments in `cpl_interface_` methods are simple native Fortran (90 or 77) types such as real and integer scalars and arrays. Access to these simple arrays is coordinated through the use of integer indicies defined in the fields module discussed below. By making no assumptions on the relation between Coupler datatypes and model datatypes, moving data between the model's internal data structures and the arguments to `cpl_interface_*` is done with a copy. Testing shows that the cost of all copying is insignificant compared to the total time to simulate a day in CCSM3. More detail on the use of INTERFACE is given in Section 10.2.

8.6 The `fields` module

`cpl_fields_mod.F90` contains the master list of all scalar and gridded data transferred between the coupler and component models. The coupler and each component model shares this data through Fortran90 USE association of this module. Localizing this information in one module and requiring all component model subroutines which read or set transferred data to use it helps coordinate the access of the simple arrays used in the `cpl_interface_mod` routines.

The integer indicies for the data in the infobuffer is set in `cpl_fields_mod.F90`. Elements of the infobuffer are named according to their type and a descriptive name:

`cpl_fields_ibuf_`<*name*>: index of an integer scalar quantity in the integer portion of the infobuffer

`cpl_fields_rbuf_`<*name*>: index of a real scalar quantity in the real portion of the infobuffer

Examples include `cpl_fields_ibuf_rcode`, an inter-model error code and `cpl_fields_rbuf_eccen`, the eccentricity of the Earth's orbit. The values given to these indicies in `cpl_fields_mod.F90` are arbitrary. CCSM developers should ignore them and use the long names like `cpl_fields_rbuf_eccen` to retrieve/set values in the interface arrays.

The indicies for data sent in the bundle portion of the contract are also set in `cpl_fields_mod.F90`. These indicies are named according to the direction of travel (Coupler to Atmosphere, **c2a**, or Atmosphere to Coupler, **a2c**) and a descriptive name. For example:

`cpl_fields_a2c_lwdn` Longwave Downward radiation passed from the Atmosphere to the Coupler

`cpl_fields_c2i_ot` Ocean Sea Surface Temperature passed from the Coupler to the Ocean

Along with integer indicies for each field exchanged, the fields module also contains a colon-delimited character string defining the fields exchanged between each model and the coupler. One example is the string defining all the fields sent from the atmosphere to the coupler, `cpl_fields_a2c_fields`. This and similar character strings are used by the Coupler to allocate memory in a BUNDLE at runtime. The number of colon-separated items in the string is counted and that number, along with the local grid size, is used to allocate local storage. The individual string names, also called "attributes" of the BUNDLE (and the underlying MCT ATTRIBUTEVECTOR), are used internally by the Coupler to access data in the bundle. The exact name of each field is up to the user however repeating names between strings is part of the how the Coupler controls the flow of information between components (Section 9.2.4).

Each `*_fields` string is actually assembled by joining two sub-strings, `*_fluxes` and `*_states`. The distinction is used to route the correct data to the correct interpolation routine. See Section 9.2.4 for more information.

Finally, the fields module also contains the total number of fields exchanged between each model and the coupler, e.g. `cpl_fields_a2c_total`. This parameter is used in setting the size of the simple arrays used in the interface routines at compile-time.

NOTE: There is a strong relation between the colon separated character string `*_fields` and the integer parameters like `cpl_fields_a2c_lwdn`. Since "Faxa_lwdn" (longwave down from the atmosphere) is the 10th item in the `cpl_fields_a2c_fields` character string, `cpl_fields_a2c_lwdn` must be set equal to 10. (See `cpl_fields_mod.F90`). *This relationship must be maintained by the programmer when modifying* `cpl_fields_mod.F90`.

8.7 The `map` module

The map module represents a major subsystem of cpl6. Cpl6 uses "Mapping" to refer to the interpolation of gridded data from one grid to another, e.g. mapping atmosphere data onto the ocean grid. This is sometimes called "regridding".

`cpl_map_mod.F90` contains datatypes for holding all of the information needed to perform a mapping of data between two grids and methods for initializing that data, `cpl_map_init`, and performing the mapping between two bundles, `cpl_map_bun`.

The details of the mapping calculation are described in Section 16.1. The cpl_map datatype includes not just storage for the mapping weights, but also information needed to perform any communication necessary to complete the mapping. Because the source and destination grids are each decomposed over processors in their own way, all the data needed to complete the mapping may not be present on a processor. Thus `cpl_map_bun` performs necessary communication using additional information in cpl_map and methods from MCT such as Rearranger.

9 The cpl6 `main`

This section provides guidance to understanding the cpl6 `main` program which coordinates the transfer of data between components in CCSM3.

9.1 Variable Naming Conventions

A variable naming convention has been adopted to help organize and identify the literally hundreds of state and flux fields managed by the Coupler. Understanding this naming convention will aid understanding the `main` code and pseudo-code below.

9.1.1 State Variables

Model state variables are denoted Sx, where x denotes the corresponding component model:

* `Sa` = atm state variables, e.g. atm wind velocity
* `Si` = ice state variables, e.g. ice temperature
* `Sl` = lnd state variables, e.g. lnd albedo
* `So` = ocn state variables, e.g. ocn SST
* `Ss` = all surface states merged together, e.g. global surface temperature

Next, a suffix `_x` indicates what model grid the states reside on:

* `Sa_a:` atm states on atm grid
* `Sa_o:` atm states on ocn grid

9.1.2 Flux Fields

Input flux means fluxes given by the Coupler to a model. Output flux means fluxes computed within a model and given to the Coupler. All input fluxes for one model were either computed by the Coupler or were the output flux of another model. In general, a given flux field between any two component models may have been computed in one of three places: within either of the two models or within the Coupler.

One function of the Coupler is to gather, merge, sum, and/or time-average the various component flux fields from various sources and form a set of complete input fluxes for each component model. This gathering and merging process will generally involve mapping flux fields between various model grids and combining like fields from several grids onto one grid. A summation might be required, e.g., **net heat flux** = **solar** + **latent** + **sensible** + **longwave**. Also, for some flux fields the Coupler might be required to form time-averaged quantities. Thus component fluxes are mapped, merged, summed, and/or time-averaged by the Coupler in order to form complete input fluxes for the models.

Flux fields are denoted `Fxyz`, where `xy` denotes the two model between which a quantity is being fluxed, and `z` denotes the model which computed the flux (i.e. it is an output flux of model z).

Component fluxes that are gathered, merged, summed, and/or time–averaged to form the complete input fluxes are:

* `Faia` = atm/ice flux, computed by atm, e.g. precipitation
* `Faii` = atm/ice flux, computed by ice, e.g. sensible heat flux

* `Fala` = atm/lnd flux, computed by atm, e.g. precipitation
* `Fall` = atm/lnd flux, computed by lnd, e.g. sensible heat flux
* `Faoc` = atm/ocn flux, computed by cpl, e.g. momentum flux
* `Faoa` = atm/ocn flux, computed by atm, e.g. precipitation
* `Fioo` = ice/ocn flux, computed by ocn, e.g. ice formed within the ocn
* `Fioi` = ice/ocn flux, computed by ice, e.g. penetrating shortwave radiation
* `Flol` = lnd/ocn flux, computed by lnd, e.g. river runoff

Complete input fluxes (an "a" prefix denotes a daily average):

* `Faxx` = all atm input fluxes: a map/merge/sum of: Faoc, Faii, Fall
* `Flxx` = all lnd input fluxes: a map/merge/sum of: Fala
* `Foxx` = all ocn input fluxes: a map/merge/sum of: Faoc, Faoa, Fioi, Flol
* `aFoxx` = a time average of: Foxx
* `Fixx` = all ice input fluxes: a map/merge/sum of: Faia, Fioo

Finally, just like the state variables above, a suffix `_x` indicates what model grid the fluxes reside on:

* `Faoa_a:` atm/ocn fluxes, computed by the atm, on the atm grid
* `Faoa_o:` atm/ocn fluxes, computed by the atm, on the ocn grid

9.1.3 Domain Maps

Mapping between domains is implemented by matrix multiplies. Generally there are different maps for state fields vs. flux fields, as state fields maps are generally smoother (e.g. bilinear) whereas flux field maps must be conservative (e.g.. area averaging, aka Riemann sum integrals). In the case of mapping between identical domains, the map would be the identity map. The necessary mapping matrices are:

* `map_Fa2i:` map for fluxes, atm -> ice
* `map_Sa2i:` map for states, atm -> ice
* `map_Fa2l:` map for fluxes, atm -> lnd
* `map_Sa2l:` map for states, atm -> lnd
* `map_Fa2o:` map for fluxes, atm -> ocn
* `map_Sa2o:` map for states, atm -> ocn
* `map_Fi2a:` map for fluxes, ice -> atm
* `map_Si2a:` map for states, ice -> atm
  etc.

9.1.4 Bundles and Contracts

The Coupler declares several instances of the BUNDLE and CONTRACT datatypes and they also follow a naming convention. Bundles have names similar to the map's but with an extra postfix to identify which grid the data is on. For example:

* `bun_Sa2c_a`: State data from the atmosphere on the atmosphere grid.
* `bun_Sa2c_o`: State data from the atmosphere on the ocean grid.
* `bun_Fa2c_a`: Flux data from the atmosphere on the atmosphere grid.
* `bun_Fa2c_o`: Flux data from the atmosphere on the ocean grid.

The `bun_Sa2c_o` bundle contains the `bun_Sa2c_a` data after it has been mapped (interpolated) onto the ocean grid.

The names of Contracts also contain a direction but, since the BUNDLE in the CONTRACT contains both State and Flux data, an "X" is used in the name.

* `con_Xa2c`: The atmosphere-to-coupler CONTRACT.
* `con_Xc2a`: The coupler-to-atmosphere CONTRACT.
* `con_Xi2c`: The ice-to-coupler CONTRACT.
* `con_Xc2i`: The coupler-to-ice CONTRACT.

9.2 Scientific Assumptions in `main`

The code of the cpl6 `main` fixes many of the assumptions and scientific requirements of CCSM3. This section describes which assumptions/requirements have been "hard-coded" into CCSM3's coupler. However, because of the modular structure of the cpl6 code, it is straightforward to remove most of these assumptions and allow alternate configurations of CCSM3 components.

9.2.1 Number of Components

The total number of components is fixed at 6 corresponding to the number of components in CCSM3: the Coupler itself, and models of the atmosphere, land, river runoff, sea ice and ocean. However the number of separate executables is limited to 5 because in CCSM3 the land and river runoff are contained in the same model, CLM. This assumption is in both `main` and the `cpl_comm_mod` module which contains public data members with parallel programming information for only 5 components and a method to initialize them.

Since the river model runs on a different grid, the Coupler establishes a separate contract for the one-way exchange of information from the river model to the Coupler and the river model is treated as a sixth component. But in the MPI view of CCSM, the land and river models are on the same processors and share the same MPI communicator.

9.2.2 Number of Grids

The Coupler `main` assumes there are only 3 different grids however the size and type of each grid may change. The atmosphere and land models are assumed to share the same grid and the ocean and sea ice models are also assumed to share the same grid (but different from the atmosphere-land). The river model's grid is the third grid in the CCSM3 Coupler.

9.2.3 Time Integration and Coordination

The Coupler code is configured for a particular time coordination scheme. Model time coordination involves two communication intervals, with the longer interval being an integer multiple of the shorter interval. The atmosphere, ice, and land models communicate once per short interval while the ocean model communicates once per long interval. Also, one day (24 hours) is an integer multiple of the longer communication interval. Typically the longer interval is exactly one day, while the shorter interval is several hours or less. While this configuration is hard-coded within the Coupler, its modular design facilitates code modifications to implement alternate configurations. A variety of time coordination schemes can be, and have been, implemented by rearranging subroutine calls at the highest level (within the main program), requiring a minimal amount of code modifications or new code.

9.2.4 Data Pathways through the Coupler

The Coupler code allows only certain data pathways between models. We define a "pathway" as the sequence of operations performed on data sent from a component model to the Coupler on it's way to another model. An example is shown in Figure 2. Figure 2 shows what happens to the BUNDLE of data from the atmosphere after it enters the coupler. The number of pathway's and the sequence of events in each is fixed in a given release of the Coupler. Indeed the code in the integration loop of the Coupler's `main` consists mostly of calls to cpl6 routines to route data through these pathway's.

Some pathways are not implemented in the current release of the Coupler. There is no path directly from the land model to the ocean model (although there is a path from the river model to the ocean model.) Pathways which do exist have only certain operations. For example in the path between the ice and the ocean model, there is no mapping because it is assumed that the ice and ocean models are on the same grid (Section 9.2.2).

9.3 Main Program Detail

To understand the details of the Coupler's functionality, it is useful to have a basic understanding of the Coupler `main`'s source code. To that end, pseudocode for `main` is shown below. If one wishes to modify the Coupler source code, it is strongly recommended that one first study this section in conjunction with studying the source code file, `main.F90`. This should provide a good overview of how the Coupler works; a necessary pre-requisite for successful code modification.

9.3.1 Psuedocode for `main`

In the pseudocode below, code from the Coupler is in `typewriter font`. Comments about the code are in *italics* and summaries representing sections of code are in [ALL CAPS].

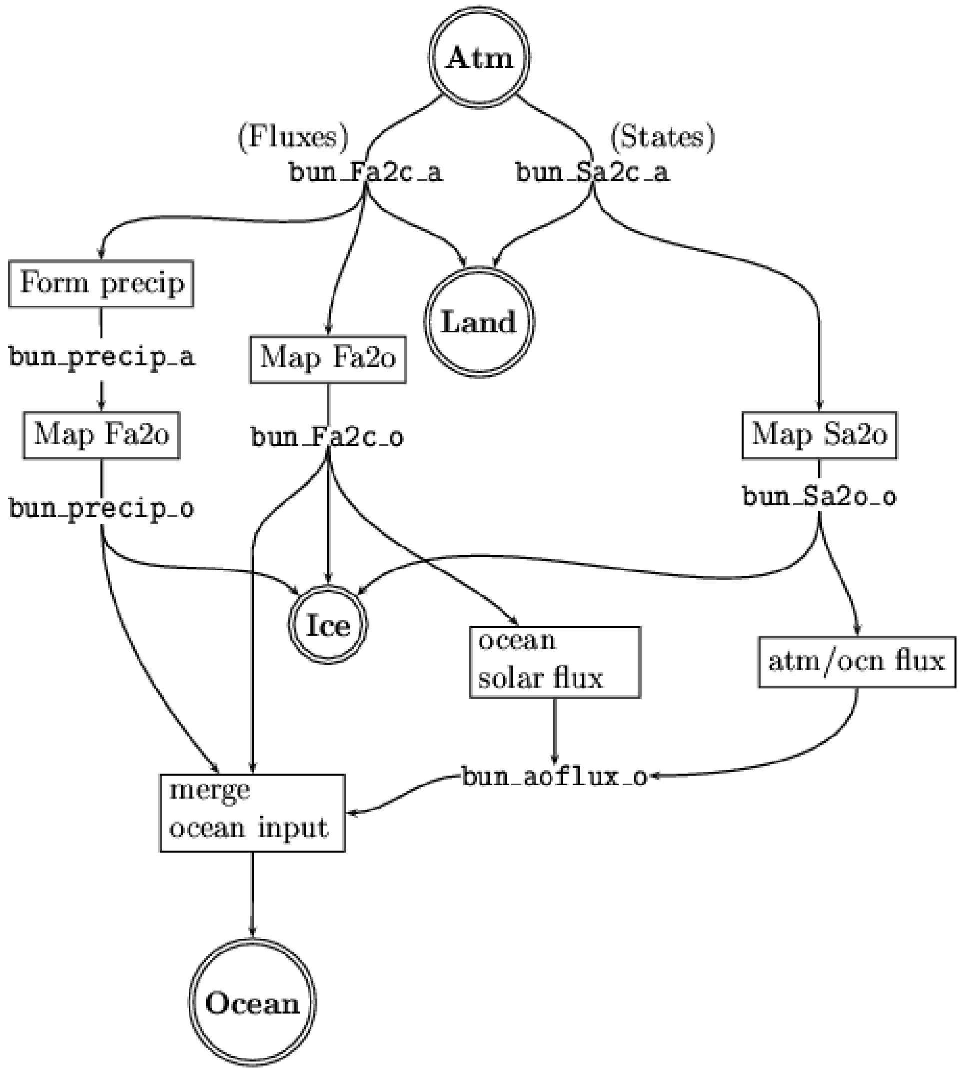

Figure 2: The pathway of data from the atmosphere model through the coupler. Note that not all operations are shown

```
PROGRAM main

  [INITIALIZE TIMERS]

  Initialize MPI and get local MPI communicator
  call cpl_interface_init(cpl_fields_cplname,local_comm)

  Read the Coupler namelist
  call cpl_control_readNList()

  Get simulation start date
  call restart_readDate(cDate)

  Initialize the Coupler infobuffers in each Contract
  call cpl_infobuf_init(con_Xa2c%infobuf)
  [INITIALIZE OTHER INFOBUFFERS]

  Initialize receiving Contract with atmosphere
  call cpl_interface_contractInit(con_Xa2c,cpl_fields_cplname, &
    cpl_fields_atmname,cpl_fields_a2c_fields,bunname="Xa2c_a", &
    decomp=cpl_control_decomp_a)

  Initialize sending Contract with atmosphere
  call cpl_interface_contractInit(con_Xc2a,cpl_fields_cplname, &
    cpl_fields_atmname,cpl_fields_c2a_fields,bunname="Xc2a_a", &
    decomp=cpl_control_decomp_a)

  dom_a = con_Xa2c%domain   Make sure domains are identical
  con_Xc2a%domain = dom_a

  [INITIALIZE CONTRACTS WITH ICE AND OCEAN MODEL]

  [INITIALIZE SPECIAL DOMAIN CONTRACT WITH LAND MODEL.
      DETERMINE IF COUPLER SHOULD SEND LAND DOMAIN.]

  Initialize inter-domain area fractions
  call frac_init(map_Fo2a,dom_a,dom_i,dom_l,dom_o)

  [IF REQUESTED, DETERMINE LAND DOMAIN AND SEND TO LAND MODEL]

  [INITIALIZE CONTRACTS WITH LAND AND RIVER MODEL]

  [CHECK THAT LAND AND ATMOSPHERE DOMAINS ARE CONSISTENT]
  [CHECK THAT ICE AND OCEAN DOMAINS ARE CONSISTENT]

  call data_bundleInit()   Initialize Coupler bundles

  call data_mapInit()   Initialize Coupler maps for interpolation
```

```
call history_avbundleInit()   Initialize history buffers

[SET VALUES FOR INITIAL INFOBUFFER]

Send infobuffer to atmosphere
call cpl_interface_infobufSend(cpl_fields_atmname,
               con_Xc2a%infobuf%ibuf,con_Xc2a%infobuf%rbuf)
[SEND INFOBUFFERS TO OCEAN, ICE, LAND AND RIVER MODELS]

[VERIFY ACCEPTABLE COUPLING INTERVALS.]

[CHECK FOR DEAD COMPONENTS.]

date = shr_date_initCDate(cDate,ncpl_a)  Initalize model start date
call cpl_control_init(date)                   and control flags
call cpl_control_update(date)

[RESET BUNDLE DOMAINS AREAS TO MAP FILES AREAS]

Receive initial condition data from atmosphere
call cpl_interface_contractRecv(cpl_fields_atmname,con_Xa2c)
Multiply received data by normalized area
call cpl_bundle_mult(con_Xa2c%bundle,bun_areafact_a,'comp2cpl',  &
                     bunlist=cpl_fields_a2c_fluxes)
[RECEIVE INITIAL DATA FROM ICE, LAND, RIVER AND OCEAN MODELS]

call restart_read(date) Read restart

[PROCESS (SPLIT INTO BUNDLES AND MAP) INITIAL CONDITION DATA]

[IF REQUESTED, SEND ALBEDO TO ATMOSPHERE AND GET NEW INITIAL DATA]

DO WHILE ( .not. cpl_control_stopNow )   Main integration loop

   DO n=0,ncpl_a    Loop over number of atmosphere communications per day

       Send message to ocean
      if(mod(n-1,ncpl_a/ncpl_o) == 0 ) then   If time to talk to ocean
         if (.not. cpl_control_lagOcn) then
           multiply sent data by normalized area
           call cpl_bundle_mult(con_Xc2o%bundle,bun_areafact_o, &
                             'cpl2comp',bunlist=cpl_fields_c2o_fluxes)
           Send the data
           call cpl_interface_contractSend(cpl_fields_ocnname,con_Xc2o)
           multiply sent data by inverse of normalized area
           call cpl_bundle_mult(con_Xc2o%bundle,bun_areafact_o, &
                     'comp2cpl', bunlist=cpl_fields_c2o_fluxes)
         endif
         call cpl_bundle_zero(bun_Xc2oPSUM_o) zero out the partial sum
```

```
endif

if (mod(n-1,ncpl_a/ncpl_l) == 0 ) then
```
[SEND DATA TO LAND MODEL]

Map atmosphere states (from initial receive or previous step) to ocean grid
```
call cpl_map_bun(bun_Sa2c_a,bun_Sa2c_o,map_Sa2o,mvector=a2ovector)
```
Map atmosphere fluxes to ocean grid)
```
call cpl_map_bun(bun_Fa2c_a,bun_Fa2c_o,map_Fa2o,mvector=a2ovector)
```

[IF REQUESTED, FORCE BALANCE THE FRESH WATER FLUX]

Correct the a2o vector mapping near north pole
```
call cpl_map_npfix(bun_Sa2c_a,bun_Sa2c_o,'Sa_u','Sa_v')
```

[SEND DATA TO ICE MODEL]

Compute net solar flux into ocean.
```
call flux_solar(bun_Fa2c_o, bun_oalbedo_o, bun_aoflux_o )
```

`call merge_ocn()` *merge ocean inputs*

Form time average of ocean inputs
```
call cpl_bundle_accum(bun_Xc2oSNAP_o,outbun=bun_Xc2oPSUM_o)
```

[DO DIAGNOSTICS]

`call flux_albo(date,bun_oalbedo_o)` *Compute ocean albedos. (Sec. 16.3.3)*

Compute atmosphere/ocean fluxes. (Sec. 16.2)
```
call flux_atmOcn(con_Xo2c%bundle,bun_Sa2c_o,cpl_control_dead_ao,
                     bun_aoflux_o )

if (mod(n-1,ncpl_a/ncpl_i) == 0 ) then
```
[RECEIVE DATA FROM ICE MODEL]

If requested, add diurnal cycle to ice albedo. (Sec. 16.3.2)
```
call flux_albi(date,con_Xi2c%bundle)
```

Split received ice data into states and fluxes
```
call cpl_bundle_split(con_Xi2c%bundle,bun_Si2c_i,bun_Fi2c_i)
```

Map ice and ocean states to atmosphere grid
```
call cpl_map_bun(bun_Si2c_i,bun_Si2c_a,map_So2a, &
                 bun_frac_i,'ifrac',bun_frac_a,'ifrac',oi2avector)
call cpl_map_bun(bun_So2c_o,bun_So2c_a,map_So2a,  &
                 bun_frac_o,'afrac',bun_frac_a,'ofrac',oi2avector)
```

[MAP ICE AND OCEAN FLUXES TO ATMOSPHERE GRID]

```
        if (mod(n-1,ncpl_a/ncpl_l) == 0 ) then
        [RECEIVE DATA FROM LAND MODEL]

        if (mod(n,ncpl_a/ncpl_r) == 0 ) then
        [RECEIVE DATA FROM RIVER MODEL]

        call merge_atm(fix_So2c_a) Merge atmosphere state and flux fields

        if (mod(n-1,ncpl_a/ncpl_a) == 0 ) then
        [SEND DATA TO ATMOSPHERE  MODEL]

        Map river model data to ocean grid
        call cpl_map_bun(con_Xr2c%bundle,bun_Xr2c_o,map_Xr2o,mvector=r2ovector)

        call history_write(date) Create and/or update history files
        call history_avwrite(date)

        if (mod(n,ncpl_a/ncpl_o) == 0 ) then
        [RECEIVE DATA FROM OCEAN]

        if (mod(n-1,ncpl_a/ncpl_a) == 0 ) then
        [RECEIVE DATA FROM ATMOSPHERE]

        call shr_date_adv1step(date)  Advance date and update control flags
        call shr_date_getCDate(date,cDate,sec)
        call shr_date_getYMD(date,year,month,day,sec)
        call cpl_control_update(date)

     END DO

     call timeCheck(date,.false.,.true. ) Verify time coordination

  END DO WHILE   End of main integration loop

  call history_avwrite(date) Write last history file

  Send final message to atmosphere
  call cpl_interface_contractSend(cpl_fields_atmname,con_Xc2a)
  [SEND FINAL MESSAGE TO ICE, LAND AND OCEAN MODELS]

  call cpl_interface_finalize(cpl_fields_cplname) Disconnect from MPI

STOP/END OF PROGRAM
```

9.3.2 A Data Pathway in Detail

With the pseudocode, the `*_field` definitions in `cpl_fields_mod.F90`, and the BUNDLE initializations in `data_mod.F90`, one can determine how the attribute names in a BUNDLE are used to route data through pathways. Consider the incoming atmosphere bundle shown in Fig. 2. The BUNDLE in `Con_Xa2c%bundle` is initialized using the `cpl_fields_a2c_fields` string while `bun_Sa2c_a` is initialized with the `cpl_fields_a2c_states` substring and `bun_Fa2c_a` is initialized with the `cpl_fields_a2c_fluxes` substring. The call to `cpl_bundle_split` copies the data in `Con_Xa2c%bundle` into `bun_Sa2c_a` or `bun_Fa2c_a` according to the matching attribute names. Since, by construction, `*_fields` is the union of `*_states` and `*_fluxes`, the split is exact.

Attribute name matching is also used the move data through the Fa2o mapping call shown in Fig. 2. This call will automatically map all the fields in `bun_Fa2c_a` which have the same names as the fields in the output BUNDLE `bun_Fa2c_o`. Since `bun_Fa2c_o` is also initialized with `cpl_fields_a2c_fluxes` (in `data_mod.F90`), the entire contents of `bun_Fa2c_a` are mapped to the ocean grid.

Finally, attribute name matching is used to "gather" data for passing out to a component model using `cpl_bundle_gather`. Note that this does *not* mean "gather" in the MPI sense of gathering distributed data to a single processor. It means gathering data from several BUNDLES with the same attribute name into one outgoing BUNDLE. For example, `cpl_fields_a2c_states` and `cpl_fields_c2i_states` both contain an attribute called "Sa_u". Since the same name appears in both the `Con_Xa2c%bundle`, `bun_Sa2c_o`, and `Con_Xc2i%bundle`, this ensures that the data contained in the "Sa_u" slot in the `Con_Xa2c%bundle`, the atmosphere's zonal wind field, will be mapped to the ice/ocean grid with the correct mapping and sent to the ice model.

10 Modifications to the Coupled System

Some of the Coupler's behavior can be controlled through the use of a Fortran `namelist` as described in Section 4. However, some necessary changes to the coupled system, such as adding new fields, requires editing the Coupler source code. In this section, we provide some guidance on how to change the Coupler code for likely situations. This section assumes some familiarity with the CCSM code and build system.

10.1 Adding Fields to Existing Pathways

The complete list of fields exchanged between models is in `cpl_fields_mod.F90`. To add or subtract fields to existing data pathways in the Coupler (Section 9.2.4), only `cpl_fields_mod.F90` needs to be edited.

For an example, we'll consider adding a field to the ocean-to-coupler fields, `cpl_fields_o2c_*`. The first step is to add a descriptive character string to the colon-delimited list of fields. The field must be added to `cpl_fields_o2c_states` if its a state and `cpl_fields_o2c_fluxes` if it is a flux. Adding a state variable `So_new` is shown below with changes in **bold**.

```
character(*), parameter,public :: cpl_fields_o2c_states = &
   &'So_t&
   &:So_u&
   &:So_v&
   &:So_s&
```

```
   &:So_dhdx&
   &:So_dhdy&
   &:So_new'
character(*), parameter,public :: cpl_fields_o2c_fluxes = &
   &'Fioo_q'
character(*), parameter,public :: cpl_fields_o2c_fields = &
   trim(cpl_fields_o2c_states)//":"//trim(cpl_fields_o2c_fluxes)
```

Next, the integer parameters for the fields must be modified. Note that the new field must be numbered correctly according to its position in the joined `cpl_fields_o2c_field` string.

```
!----- ocn states -----
integer(IN),parameter,public :: cpl_fields_o2c_t    =  1 ! temperature
integer(IN),parameter,public :: cpl_fields_o2c_u    =  2 ! velocity, zonal
integer(IN),parameter,public :: cpl_fields_o2c_v    =  3 ! velocity, meridional
integer(IN),parameter,public :: cpl_fields_o2c_s    =  4 ! salinity
integer(IN),parameter,public :: cpl_fields_o2c_dhdx =  5 ! surface slope, zonal
integer(IN),parameter,public :: cpl_fields_o2c_dhdy =  6 ! surface slope, meridional
integer(IN),parameter,public :: cpl_fields_o2c_new  =  7 ! new field
integer(IN),parameter,public :: cpl_fields_o2c_q    =  8 ! heat of fusion (q>0)
                                                              melt pot (q<0)
```

A new integer was added for `So_new` and the subsequent integers, `cpl_fields_o2c_q` were renumbered.

Finally, the integer parameter for the total number of fields must be updated.

```
integer(IN),parameter,public :: cpl_fields_o2c_total  = 8
```

NOTE: The programmer making these changes is responsible for ensuring that the character string `cpl_fields_o2c_fields`, the integer parameters for the individual fields and the integer parameter for the total are all consistent.

The above changes will only add a field to the ocean BUNDLE passed in to the Coupler. If you want this field passed out to another model, say the ice model, similar changes should be made to the `cpl_fields_c2i_fields` string and the string for any other models which need to receive this data.

To make the Coupler aware of these changes, recompile the code. On the next run of CCSM3 with the changes, the Coupler will automatically create extra storage for the new field and perform the same operations on `So_new` as on the rest of `cpl_fields_o2c_states`.

Because the Coupler is designed to work with models which may have very different internal data structures, additional editing must be done in the component models to complete the addition of the field. In the ocean model, code must be added to copy `So_new` from the ocean model's internal data structure into the `buf` argument of the ocean's `cpl_interface_contractSend` call. And any receiving models must also be edited to copy the received data into it's appropriate internal data structure. The steps needed to interface a component model with the Coupler are described below.

10.2 Interfacing a Component with the Coupler

This section will review what is required to interface a component with the Coupler *in place of a current component*, e.g. replacing POP with a different ocean model or CAM with a different atmosphere, while sending and receiving the same fields. This only covers how to communicate with the Coupler and ignores other issues such as merging the model with the CCSM3 build system. Not every detail will be covered and programmers are encouraged to examine the code of current CCSM3 components to see how they interact with the Coupler. Clear examples of how a model is connected to the Coupler can be found in the routine `msg.F90` in any of the data models (`../models/atm/datm6`, `../models/ocn/docn6/` etc.). `msg.F90` contains all of the data model's interaction with the Coupler and mimics the steps performed by the full models. One can find all the references to the Coupler in any model within CCSM3 by searching for `cpl_` in the code.

All the routines needed to connect a model to the CCSM system are in `cpl_interface_mod.F90`. Connecting a model to the CCSM Coupler involves placing calls to these routines within the model. No other editing or reorganization of a model's source code is required. Where exactly the calls go depends on the model code but there is some generality which can be used to guide their placement.

To use the routines in `cpl_interface_mod.F90`, a Fortran90 USE statement must be placed at the top of each model. Because references must also be made to the integer parameters in `cpl_fields_mod.F90` and the CONTRACT datatype in `cpl_contract_mod.F90`, 3 use statements must appear in each model subroutine involved in Coupler communication:

```
use cpl_contract_mod
use cpl_interface_mod
use cpl_fields_mod
```

The first call which must be made is to `cpl_interface_init`:

```
call cpl_interface_init(cpl_fields_<modelname>,local_comm}
```

The first argument, `cpl_fields_<modelname>` must be one of the component names defined at the top of `cpl_fields_mod`. The second argument returns an *MPI_ Communicator* which must be used by any MPI calls made internally within the model. (`cpl_interface_init` also handles calling `MPI_Init`.) This call only needs to be made once and can be placed within one of the model's initialization routines.

The second and third calls are to `cpl_interface_contractInit`, one each to initialize the send and receive CONTRACT (Sec. 8.4). When called from a model, `cpl_interface_contractInit` should use the following set of arguments:

```
call cpl_interface_contractInit(contractS, cpl_fields_<modelname>,
         cpl_fields_cplname, cpl_fields_<x>2c_fields, ibufi, buf, ibufr)
```

`contractS` will be the output *send contract* and is the only Fortran90 derived datatype a component model needs to declare to interact with the Coupler:

```
type(cpl_contract),save :: contractS
```

The other arguments of the INTERFACE routines are native scalars and arrays. `<x>` should be replaced with o, a, i, l, or r depending on if the calling model is an ocean, atmosphere, etc. `ibufi` and `ibufr` are the integer and real parts of the INFOBUFFER (Sec. 8.3) as simple arrays declared with one of the FIELDS parameters:

```
integer :: ibufi(cpl_fields_ibuf_total)
real    :: ibufr(cpl_fields_ibuf_total)
```

During the CONTRACT initialization, only some of the integer parts of the infobuffer must be set before calling `cpl_interface_contractInit`:

```
ibufi(cpl_fields_ibuf_ncpl   ) = number of communications per day
ibufi(cpl_fields_ibuf_gsize  ) = the size of the global grid
ibufi(cpl_fields_ibuf_lsize  ) = size of the local grid
ibufi(cpl_fields_ibuf_gisize ) = global size in i-index
ibufi(cpl_fields_ibuf_gjsize ) = global size in j-index
ibufi(cpl_fields_ibuf_nfields) = cpl_fields_grid_total
ibufi(cpl_fields_ibuf_dead)    = 0 if not a dead model
```

The two-dimensional real array `buf` can be declared or allocated using another FIELDS parameter:

```
real :: buf(nx*ny, cpl_fields_grid_total)
```

where `nx` and `ny` are the *local* grid sizes in the x and y directions. The `buf` array is used to tell the Coupler about the grid the model is running on its information is used to initialize the DOMAIN and MCT datatypes. An example of filling buf can be found in `datm6/msg.F90`:

```
n=1
do j=1,ny                    ! local  j index
do i=1,nx                    ! local  i index
   buf(n,cpl_fields_grid_lon   ) = xc(i,j)
   buf(n,cpl_fields_grid_lat   ) = yc(i,j)
   buf(n,cpl_fields_grid_area  ) = area(i,j)
   buf(n,cpl_fields_grid_mask  ) = mask(i,j)
   buf(n,cpl_fields_grid_index ) = n
   n=n+1
enddo
enddo
```

In this case datm6 has very simple internal data structures to hold the local values of longitude, latitude, area and mask. datm6's internal 2D array's, `xc`, `yc`, etc., are "unrolled" into 1-dimension of `buf` while the other dimension of `buf` is used for the different fields. The latitude, longitude, area and mask values should be local values but `cpl_fields_grid_index` contains a global index number given to each point on the portion of the model's numerical grid communicating with the coupler (i.e. the lowest level of a 3D atmosphere grid). Since datm6 is a single-processor application, this assignment is trivial but care must be taken to assign the correct values to `cpl_fields_grid_index` for a processor in a parallel application since this information is used by MCT to perform the parallel communication between the model and the Coupler. On each processor, `cpl_fields_grid_index` should contain the globally-indexed values of points that processor "owns". The *order* should be noted because later, when computed data is sent to the coupler, it must be placed in the send buffer in the same order. More detail on grid point numbering and its use in parallel communication can be found in the *User's Guide to the Model Coupling Toolkit.*

The receive contract call to `cpl_interface_contractInit` can use the same input arguments as the send illustrated above but the first argument should be different, e.g. `contractR`. Like `cpl_interface_init`, the `cpl_interface_contractInit` calls also only need to be done once as part of the model's initialization. The last initialization step is to receive an INFOBUFFER from the Coupler with a call to `cpl_interface_ibufRecv`. The CONTRACTS initialized above must be passed to the routines which will communicate with the coupler during runtime.

Sending and receiving data during execution is very similar to the CONTRACT initialization. During a send, first real and/or integer arrays like `ibufr` and `ibufi` are updated with appropriate items in the INFOBUFFER. The a real array similar to `buf` is loaded with the latest calculated values intended for the Coupler using the indicies defined in `cpl_fields_mod` to place data in the correct location in `buf`. Then the INTERFACE send routine is called:

```
call cpl_interface_contractSend(cpl_fields_cplname, contractS, ibufi, buf)
```

A receive reverses this order. First a call is made to `cpl_interface_contractRecv`, then data is copied out of `buf` using the indicies in `cpl_fields_mod` into the appropriate internal datatypes and scalars from the INFOBUFFER are also accessed using the indicies in `cpl_fields_mod` and the `ibufi` or `ibufr` arrays.

Part III
Scientific Reference

11 Introduction

This part of the documentation describes the scientific requirements behind the design of the CCSM Coupler. It also provides the details behind the scientific calculations performed in the Coupler including atmosphere-ocean flux calculations and mapping between different grids.

12 What is a "Coupler"?

The CCSM coupled model is a framework that divides the complete climate system into component models connected by a coupler. In this design the Coupler is a hub that connects four major component models – atmosphere, land, ocean, and sea-ice (see Figure 3. We'll ignore the river model component contained in the land model). Each component model is connected to the Coupler, and each exchanges data with the Coupler only. The CCSM is not a particular climate model, but a framework for building and testing various climate models for various applications. In this sense, more than any particular component model, the Coupler defines the high-level design of CCSM software. The Coupler code

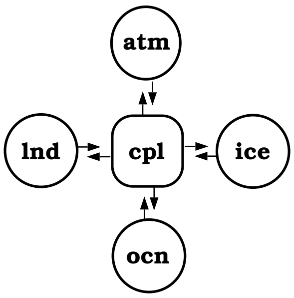

Figure 3: The CCSM Coupled Model Framework in its most basic form.

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. CCSM components run as separate executables communicating via message passing (MPI).

- It controls the execution and time evolution of the complete CCSM by synchronizing and controlling the flow of data between the various components.

- It communicates interfacial fluxes between the various component models while insuring the conservation of fluxed quantities. For certain flux fields, it also computes interfacial fluxes based on state variables. In general, the Coupler allows any given flux field to be computed once in one component, this flux field is then routed through the Coupler so that other involved components can use this flux field as boundary forcing data.

13 Development History

The origins of the Coupler are in the Oceanography Section (OS) of the Climate and Global Dynamics Division (CGD) of the National Center for Atmospheric Research (NCAR). While coupling the Community Climate Model (CCM2, the atmosphere model) with a regional Pacific Basin Model for ESNO/El-Nino studies (Gent, Tribbia, Kauffman, & Lee, 1990), it became clear that it was extremely awkward to couple the models by implementing the ocean model as a subroutine within an atmosphere model. An idea emerged that a more desirable software architecture would be to have both the ocean and atmosphere models as subroutines to a higher-level driver/coupler program. This "coupler" component would reconcile the differing component model spatial grids (handle the mapping of data fields), reconcile different time step intervals, and control when the coupled system would start and stop. Thus the two component models could be very independent and largely unaware of the software details of the other components. The modularity of this arrangement also suggested a framework in which, for example, it would be relatively easy to exchange one ocean component for another.

Other OS scientists joined in (McWilliams, Large, Bryan, 1991), and together worked out a scientific design philosophy for the Coupler and the coupled system, for example, that the appropriate boundary conditions for component models were the fluxes across the surface interfaces, that fluxes should be conserved, and that ad hoc flux corrections were undesirable. Frank Bryan suggested the models be separate executables communicating by message passing rather than subroutines in a single executable, this resulted in an MPMD implementation that proved extremely successful.

Climate Modeling Section (CMS) scientists joined the effort (Boville, et.al., 1992), working on the atmosphere component (CCM) so that it produced a quality simulation when bottom boundary conditions were fluxes instead of fixed BC's. The CMS also had experience in managing community modeling efforts – the CMS managed the development of the CCM, a very successful community atmosphere climate model. With the Coupler and its associated framework, and with a critical mass of engaged ocean and atmosphere model scientists, it now seemed plausible to start an institutional coupled climate model project.

A proposal was made to NSF for a new, NCAR-wide, Climate System Model (CSM) Project (1993), with the long-term goal of building, maintaining, and continually improving a comprehensive model of the climate system. The Coupler proto-type code and design philosophy formed the basis for CSM's software framework. The CSM project was funded and formally began in 1994.

Also during this time, the Department of Energy had funded the development of the Parallel Climate Model in a group lead by Warren Washington which included Tony Craig and Tom Bettge. Around the year 1999, it was decided to merge the two models and create one high-performance parallel Community Climate System Model. The creation of a coupler which could be both a distributed memory parallel application and still function as the "hub" in the CCSM system was a very difficult problem. To solve this problem, a team consisting of NCAR scientists from the CCSM and PCM projects (Kauffman, Craig and Bettge) and scientists from DOE laboratories at Argonne (Jacob, Larson and Ong) and Berkeley (Ding, He) was assembled. The resulting effort created not only cpl6 but two standalone software packages: the Model Coupling Toolkit, the which underlies most of cpl6 and handles all the communication intricacies presented by CCSM, and MPH, which handles high-level allocation of MPI resources.

The name, "cpl6" or "Coupler, version 6", alludes to six versions of the Coupler, they are:

`cpl1` (Kauffman, 1990): a rough proto-type code, never released. Coupler is a driver program that handles all high-level control (eg. start/stop/advance of component models) and all mapping between grids. The complete system is a single-executable with two swappable component model subroutines: ocean/ice and atmosphere/land. Written in Fortran 77.

`cpl2` (NCAR/CGD, Oceanography Section, 1992): a proto-type code, never released. New scientific design philosophy, does flux calculations, insures conservation of fluxed quantities. Complete system uses component models that are separate executables. Uses PVM for message passing.

`cpl3` (NCAR/CGD, 1996): released with CSM 1.0 Has three component models: atmosphere/land, ocean, and ice. First fully functional version. Uses MCL for message passing. Uses netCDF.

`cpl4` (NCAR/CGD, 1998): released with CSM 1.2 Has four component models: atmosphere, land, ocean, and ice. Uses MPI for message passing.

`cpl5` (NCAR/CGD, 2002): released with CCSM 2.0 Written in Fortran 90, uses SCRIP to generate mapping data, handles shifted pole grids.

`cpl6` (NCAR/CGD, ANL/MCS, LBNL 2004): released with CCSM 3.0 Has all the functionality of cpl5 but is a complete rewrite which generalizes the interface between models and the Coupler and also allows data parallelism. Built on top of MCT and also uses MPH3.

14 Scientific Requirements

The design of the Coupler and the CCSM modeling framework were motivated by a variety of scientific and software design issues. Following are some of the major design and functionality considerations which addressed the deficiencies of standard coupling strategies at the time of Coupler's inception (circa 1991), and which have continued to be important considerations.

(a) System decomposition and component model code independence
The CCSM coupling strategy and the Coupler component (see Figure 1) allows an intuitive decomposition of the large and complex climate system model. To a large extent, the separate component models, atmosphere, sea-ice, land, and ocean, can be designed, developed, and maintained independently, and later "plugged into" the Coupler to create a complete climate system model. At the highest design level, this creates a highly desirable design trait of creating natural modules (or "objects") with maximum internal cohesion and minimal external coupling.

(b) Control of the execution and time evolution of the system.
The Coupler (a natural choice), rather than one of the component models (any of which would be an arbitrary and asymmetrical choice), is responsible for controlling the execution and time evolution of the complete system.

(c) The appropriate boundary condition for all component models are the fluxes across the surface interfaces.
In terms of the climate simulation, the Coupler (originally called the "Flux Coupler") couples component models by providing flux boundary conditions to all component models. State variables flow through the Coupler only if necessary for a flux calculation in the Coupler or a component model.

(d) Conservation of fluxed quantities.
The Coupler can conserve all fluxed quantities that pass through it. Since all surface fluxes pass through the Coupler, this framework assures that all surface fluxes in the system are conserved. The Coupler also can and does monitor flux conservation by doing spatial and temporal averages of all fluxed quantities.

(e) A flux field should should be computed only once, in one component, and then the field should be given to other components, as necessary.
For example, generally the resolution of the ocean component is finer than that of the atmospheric component. Heat is not conserved if the atmosphere component computes long wave radiation using sea surface temperature (SST) averaged onto the atmosphere's grid, while the ocean component uses individual grid point SSTs. This is because the flux calculation is proportional to SST^4 (is non-linear) and the two calculations will yield two different results . Instead of trying to compute the same flux twice, on two different grids in two different components, the flux should be computed once and then be mapped, in a conservative manner, to other component grids.

(f) Fluxes can be computed in the most desirable place.
For example, when an atmospheric grid box covers both land and ocean, a problem arises if wind stress is first computed on the atmospheric grid and then interpolated to the ocean grid. If land roughness is significantly larger than ocean roughness (as is typical), and the atmosphere uses an average underlying roughness to compute the wind-stress, and then wind-stress is interpolated to coastal land and ocean cells, wind-stress will be

considerably lower (for land) or higher (for ocean) than it should be. This approach can lead to unrealistic coastal ocean circulations. In this case it is more desirable to compute the surface stresses on the surface grids and subsequently merge and map them to the atmospheric grid. Thus this calculation should take place in either the Coupler, land, ice, or ocean models.

In the case of computing precipitation, because this calculation requires full 3D atmospheric fields, and because only 2D surface fields are exchanged through the Coupler, this calculation should take place in the atmosphere component.

(g) Allowing the coupling of component models with different spatial grids.
In general it is desirable to allow the different component models to exist on different spatial grids. It is also desirable that a component model need not be aware of, or concerned with, the spatial grids that other component models are using. The Coupler handles all mapping between disparate model grids, allows component models to remain unaware and unconcerned with the spatial grids of other CCSM components.

Due to scientific considerations, the current version of CCSM requires that the atmosphere and land components be on the same spatial grid. Because it is not a requirement to do so, the Coupler does not have the functionality to allow the atmosphere and land to be on different grids. Similarly, the current version of CCSM requires that ocean and sea-ice components be on the same grid and the Coupler does not have the functionality to allow them to be on different grids.

Except as noted above, no component needs to know what spatial grid other components are using. The Coupler is responsible for all mapping between the various spatial grids of the various components models. The component models themselves have no knowledge of what other grids are involved in the coupled system and they can remain unconcerned with any issues regarding mapping between grids.

(h) Allowing the coupling of models with different time steps.
While there are some restrictions on what internal time step models are allowed to use (for example, there must be an integer number of time steps per day), the Coupler offers component models considerable freedom in choosing their internal time step. Frequently all four CCSM components operate with four different internal time steps.

15 Data Exchanged with Component Models

Each component model exchanges data with the Coupler only. Component models have no direct connection with each other – all data is routed through the Coupler. Most data is in the form of 2D fields. This data is accompanied by certain timing and control information (arrays of scalar real or integer values), such as the current simulation data and time.

15.1 Units Convention

All data exchanged conforms to the following sign convention:

$$\boxed{\text{positive value} \Longleftrightarrow \text{downward flux}}$$

And these unit conventions:

`temperature`	$Kelvin$
`salinity`	g/kg
`velocity`	m/s
`pressure`	$N/m^2 = Pa$
`humidity`	kg/kg
`air density`	kg/m^3
`momentum flux`	N/m^2
`heat flux`	W/m^2
`water flux`	$(kg/s)/m^2$
`salt flux`	$(kg/s)/m^2$
`coordinates`	degrees north or east
`area`	$radians^2$
`domain mask`	$0 \Longleftrightarrow$ an inactive grid cell

15.2 Time Invariant Data

This section provides a list of the time invariant data exchanged between the Coupler and each component model. Generally this data is the "domain" data: coordinate arrays, domain mask, cell areas, etc. It is assumed that the domain of all models is represented by a 2D array (although not necessarily a latitude/longitude grid).

15.2.1 Data sent to Coupler

domain data

- grid cell's center coordinates, zonal (degrees north)
- grid cell's center coordinates, meridional (degrees east)
- grid cell area (radians squared)
- grid cell domain mask ($0 \Longleftrightarrow$ not in active domain)
- ni,nj: the dimensions of the underlying 2D array data structure

time coordination data

- ncpl: number of times per day the component will communicate (exchange data) with the Coupler.

other information

- IC flag: indicates whether the Coupler should use model IC's contained on the Coupler's restart file or IC's in the initial message sent from the component model.

15.2.2 Data sent to Component Models

time coordination data

- date, seconds: the exact time the Coupler will start the simulation from.

15.3 Time Variant Data

This section provides a list of the time-evolving data sent exchanged between the Coupler and each component model. This list is also contained in the `cpl_fields_mod` module in cpl6. Generally a quantity is a state or a flux. Some state variables are used only as diagnostics and are denoted with a *.

Each component model provides the Coupler with a set of output fields. Output fields from a model include output states (which can be used by another component to compute fluxes) and output fluxes (fluxes that were computed within the model and which need to be exchanged with another component model.

The Coupler provides each component model with input fields. Input fields sent to a model include input states (the state variables of other models, which are needed to do a flux calculation) and input fluxes (a forcing fields computed by some other component).

Flux fields sent to, or received from, the Coupler are understood to apply over the communication interval beginning when the data was received and ending when the next message is received. The component models must insure that fluxes sent to the Coupler are computed with this in mind – failure to do so may result in the non-conservation of fluxes. For example, if the atmosphere component communicates with the Coupler once per hour, but takes three internal time steps per hour, then the precipitation (water flux) sent to the Coupler should be the average precipitation over an hour (the average precipitation over three internal time steps). Similarly, if the ocean component has a communication interval of one day, but takes 50 internal time steps per day, then the precipitation flux field it receives from the Coupler should be applied as ocean boundary condition forcing for all 50 time steps during the next communication interval.

15.3.1 Atmosphere Model

Data sent to Coupler (* = diagnostic)
states

- layer height (m)
- zonal velocity (m/s)
- meridional velocity (m/s)
- temperature ($Kelvin$)

- potential temperature ($Kelvin$)
- pressure (Pa)
- equivalent sea level pressure (Pa)
- specific humidity (kg/kg)
- air density (kg/m^3)

fluxes

- precipitation: liquid, convective ($(kg/s)/m^2$)
- precipitation: liquid, large-scale ($(kg/s)/m^2$)
- precipitation: frozen, convective ($(kg/s)/m^2$)
- precipitation: frozen, large-scale ($(kg/s)/m^2$)
- longwave radiation, downward (W/m^2)
- shortwave radiation: downward, visible , direct (W/m^2)
- shortwave radiation: downward, near-infrared, direct (W/m^2)
- shortwave radiation: downward, visible , diffuse (W/m^2)
- shortwave radiation: downward, near-infrared, diffuse (W/m^2)
- net shortwave radiation* (W/m^2)

Data received from Coupler (* = diagnostic)
states

- 2 meter reference air temperature* ($Kelvin$)
- 2 meter reference specific humidity* (kg/kg)
- albedo: visible , direct [0,1]
- albedo: near-infrared, direct [0,1]
- albedo: visible , diffuse [0,1]
- albedo: near-infrared, diffuse [0,1]
- surface temperature ($Kelvin$)
- sea surface temperature ($Kelvin$)
- snow height (m)
- ice fraction [0,1]
- ocean fraction [0,1]
- land fraction [0,1] (implied by ice and ocean fractions)

fluxes

- zonal surface stress (N/m^2)
- meridional surface stress (N/m^2)
- latent heat (W/m^2)
- sensible heat (W/m^2)
- longwave radiation, upward (W/m^2)
- evaporation ($(kg/s)/m^2$)

15.3.2 Ice Model

Data sent to Coupler (* = diagnostic)
states

- ice fraction [0,1]
- surface temperature $(Kelvin)$
- 2 meter reference air temperature* $(Kelvin)$
- 2 meter reference specific humidity* (kg/kg)
- albedo: visible , direct [0,1]
- albedo: near-infrared, direct [0,1]
- albedo: visible , diffuse [0,1]
- albedo: near-infrared, diffuse [0,1]

fluxes

- atm/ice: zonal surface stress (N/m^2)
- atm/ice: meridional surface stress (N/m^2)
- atm/ice: latent heat (W/m^2)
- atm/ice: sensible heat (W/m^2)
- atm/ice: longwave radiation, upward (W/m^2)
- atm/ice: evaporation $((kg/s)/m^2)$
- net shortwave radiation* (W/m^2)
- ice/ocn: penetrating shortwave radiation (W/m^2)
- ice/ocn: ocean heat used for melting (W/m^2)
- ice/ocn: melt water $((kg/s)/m^2)$
- ice/ocn: salt flux $((kg/s)/m^2)$
- ice/ocn: zonal surface stress (N/m^2)
- ice/ocn: meridional surface stress (N/m^2)

Data received from Coupler
states

- ocn: temperature $(Kelvin)$
- ocn: salinity (g/kg)
- ocn: zonal velocity (m/s)
- ocn: meridional velocity (m/s)
- atm: layer height (m)
- atm: zonal velocity (m/s)
- atm: meridional velocity (m/s)
- atm: potential temperature $(Kelvin)$
- atm: temperature $(Kelvin)$
- atm: specific humidity (kg/kg)

- atm: density (kg/m^3)
- ocn: dh/dx: zonal surface slope (m/m)
- ocn: dh/dy: meridional surface slope (m/m)

fluxes

- ocn: $Q > 0$: heat of fusion (W/m^2), or
  $Q < 0$: melting potential (W/m^2)
- atm: shortwave radiation: downward, visible , direct (W/m^2)
- atm: shortwave radiation: downward, near-infrared, direct (W/m^2)
- atm: shortwave radiation: downward, visible , diffuse (W/m^2)
- atm: shortwave radiation: downward, near-infrared, diffuse (W/m^2)
- atm: longwave radiation, downward (W/m^2)
- atm: precipitation: liquid ($(kg/s)/m^2$)
- atm: precipitation: frozen ($(kg/s)/m^2$)

15.3.3 Land Model

Data sent to Coupler (* = diagnostic)
states

- surface temperature ($Kelvin$)
- 2 meter reference air temperature* ($Kelvin$)
- 2 meter reference specific humidity* (kg/kg)
- albedo: visible , direct [0,1]
- albedo: near-infrared, direct [0,1]
- albedo: visible , diffuse [0,1]
- albedo: near-infrared, diffuse [0,1]
- snow depth (m)

fluxes

- zonal surface stress (N/m^2)
- meridional surface stress (N/m^2)
- latent heat (W/m^2)
- sensible heat (W/m^2)
- longwave radiation, upward (W/m^2)
- evaporation ($(kg/s)/m^2$)
- coastal runoff ($(kg/s)/m^2$)

Data received from Coupler
states

- atm layer height (m)
- atm zonal velocity (m/s)

- atm meridional velocity (m/s)
- atm potential temperature ($Kelvin$)
- atm specific humidity (kg/kg)
- atm pressure (Pa)
- atm temperature ($Kelvin$)

fluxes

- precipitation: liquid, convective ($(kg/s)/m^2$)
- precipitation: liquid, large-scale ($(kg/s)/m^2$)
- precipitation: frozen, convective ($(kg/s)/m^2$)
- precipitation: frozen, large-scale ($(kg/s)/m^2$)
- longwave radiation, downward (W/m^2)
- shortwave radiation: downward, visible , direct (W/m^2)
- shortwave radiation: downward, near-infrared, direct (W/m^2)
- shortwave radiation: downward, visible , diffuse (W/m^2)
- shortwave radiation: downward, near-infrared, diffuse (W/m^2)

15.3.4 Ocean Model

Data sent to Coupler
states

- surface temperature ($Kelvin$)
- salinity (g/kg)
- zonal velocity (m/s)
- meridional velocity (m/s)
- dh/dx: zonal surface slope (m/m)
- dh/dy: meridional surface slope (m/m)

fluxes

- $Q > 0$: heat of fusion (W/m^2), or
  $Q < 0$: melting potential (W/m^2)

Data received from Coupler
states

- equivalent sea level pressure (Pa)
- ice fraction [0,1]
- 10m wind speed squared $(m/s)^2$

fluxes

- zonal surface stress (N/m^2)
- meridional surface stress (N/m^2)

- shortwave radiation, net (W/m^2)
- latent heat (W/m^2)
- sensible heat (W/m^2)
- longwave radiation, upward (W/m^2)
- longwave radiation, downward (W/m^2)
- ocean heat used for melting (W/m^2)
- salt flux ($(kg/s)/m^2$)
- precipitation: rain ($(kg/s)/m^2$)
- precipitation: snow ($(kg/s)/m^2$)
- precipitation: rain + snow ($(kg/s)/m^2$)
- evaporation ($(kg/s)/m^2$)
- melt water ($(kg/s)/m^2$)
- coastal runoff ($(kg/s)/m^2$)

16 Calculations Performed in the Coupler

The Coupler performs crucial scientific calculations which, by design of the *hub-and-spoke* system, are not or can not be handled by the component models but are essential to the coupled integration. This section provides a description of these calculations.

16.1 Mapping

The Coupler is responsible for mapping (also called interpolation or regridding) data from one model's grid to another. The Coupler implements this mapping as a matrix-vector multiply which in case of mapping atmosphere data to the ocean grid would be:

$$\overbrace{\begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix}}^{n\ atmosphere\ grid\ points} \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{pmatrix} = \overbrace{\begin{pmatrix} o_1 & o_2 & \dots & o_m \end{pmatrix}}^{m\ ocean\ grid\ points}$$

The 1xN matrix **A** contains all of the atmosphere grid points in a 2D horizontal plane unrolled into a single vector while the Mx1 matrix **O** contains all the points in a 2D horizontal slice of the ocean grid. For a T42 atmosphere and x1 ocean grid, the matrix **W** would contain $(64 \times 128) = 8192$ rows and $(320 \times 384) = 122800$ columns! Luckily most of the elements of **W** are zero and this is really a *sparse matrix multiply*. The Coupler only stores the non-zero elements and their (i, j) locations and performs the multiply with the corresponding i^{th} element from the atmosphere grid and j^{th} element from the ocean using the `cpl_map_bun` and the underlying MCT method, `sMatAvMult`.

The mapping weights in **W** are stored in files and pre-calculated using the *Spherical Coordinate Remapping and Interpolation Package* (SCRIP). See http://climate.lanl.gov/Software/SCRIP/. Two methods for calculating the weights are used in the Coupler. All *state* data is mapped with weights calculated using SCRIP's *bilinear* interpolation scheme while all *fluxes* are mapped with weights calculated using SCRIP's *second-order conservative remapping* scheme.

Given the assumptions of only 3 grids (Sec. 9.2), there are 5 sets of mapping weights read in by the Coupler: bilinear and conservative mappings for atmosphere to ocean grids, bilinear and conservative mappings for ocean to atmosphere and a conservative remapping for the river to the ocean grid (ocean to river is obviously not needed).

Note that the grid point numbering scheme mentioned in Section 10.2 is also present in the calculation of the mapping weights: the number of a model's grid point corresponds to the number of the row or column of that point in the mapping matrix **W**.

16.2 Atmosphere/Ocean Surface Fluxes

The Coupler calculates the fluxes between the atmosphere and ocean for the following reasons: By convention, fluxes between two models with different resolutions are calculated on the grid with the higher resolution. The atmosphere model can not calculate the fluxes since that would require it to know the ocean's grid. The ocean only communicates with the Coupler once per day so to update the fluxes as often sub-diurnally, the fluxes are calculated in the Coupler. The Coupler receives ocean state data and holds it constant while calculating new fluxes with each receive of new atmosphere data. The new fluxes are mapped back to the atmosphere grid and sent to the atmosphere. The Coupler also keeps a running sum of the fluxes and sends the time average to the ocean. The atmosphere-ocean fluxes are calculated using the formulas below.

16.2.1 General Expressions

The fluxes across the interface are calculated from bulk formulae and general expressions are

$$\begin{aligned}
\vec{\tau} &= \rho_A \; u^{*2} \; \Delta\vec{U} \; |\Delta\vec{U}|^{-1} \\
E &= \rho_A \; u^* \; Q^* \\
H &= \rho_A \; Cp_A \; u^* \; \theta^* \\
L\uparrow &= -\,\sigma \; T^4 \quad \simeq \quad -\,\epsilon\,\sigma \; T^4 \;+\; \alpha^L \; L\downarrow,
\end{aligned} \tag{16.1}$$

where the turbulent velocity scales are given by

$$\begin{aligned}
u^* &= CD^{1/2} \; |\Delta\vec{U}| \\
Q^* &= CE \; |\Delta\vec{U}| \; (\Delta q) \; u^{*-1} \\
\theta^* &= CH \; |\Delta\vec{U}| \; (\Delta\theta) \; u^{*-1},
\end{aligned} \tag{16.2}$$

where ρ_A is atmospheric surface density, Cp_A is the specific heat, $\sigma = 5.67 \times 10^{-8}\mathrm{W/m^2/K^4}$ is the Stefan-Boltzmann constant, ϵ is the emissivity of the interface, and α^L is the surface albedo for incident longwave radiation, $L\downarrow$. In (16.2) the differences $\Delta\vec{U}$, Δq and $\Delta\theta$ are defined at each interface in accord with the convention of fluxes being positive down. The reflected downward incident longwave radiation is simply accounted for by assuming an emissivity, $\epsilon = 1$, and the water surface albedo for incident longwave radiation, $\alpha^L = 0.0$.

The transfer coefficients in (16.2), shifted to a height, Z, and considering the appropriate stability parameter, ζ, are :

$$
\begin{aligned}
CD &= \kappa^2\left[ln\left(\frac{Z}{Z^o}\right) - \psi_m\right]^{-2} \\
CE &= \kappa^2\left[ln\left(\frac{Z}{Z^o}\right) - \psi_m\right]^{-1}\left[ln\left(\frac{Z}{Z^e}\right) - \psi_s\right]^{-1} \\
CH &= \kappa^2\left[ln\left(\frac{Z}{Z^o}\right) - \psi_m\right]^{-1}\left[ln\left(\frac{Z}{Z^h}\right) - \psi_s\right]^{-1},
\end{aligned}
\tag{16.3}
$$

where $\kappa = 0.4$ is von Karman's constant and the integrated flux profiles, ψ_m for momentum and ψ_s for scalars, are functions of the stability parameter, ζ. These functions as used in the coupler are:

$$
\begin{aligned}
\psi_m(\zeta) &= \psi_s(\zeta) = -5\zeta & \zeta > 0 \\
\psi_m(\zeta) &= 2ln[0.5(1+X)] + ln[0.5(1+X^2)] - 2tan^{-1}X + 0.5\pi & \zeta < 0 \\
\psi_s(\zeta) &= 2ln[0.5(1+X^2)] & \zeta < 0 \\
X &= (1-16\zeta)^{1/4}
\end{aligned}
$$

Above the atmospheric interfaces $i = 1,\ 2$ and 3 the stability parameter

$$
\zeta = \frac{\kappa\ g\ Z_A}{u^{*2}}\left(\frac{\theta^*}{\theta_v} + \frac{Q^*}{(Z_v^{-1} + q_A)}\right),
$$

where virtual potential temperature is computed as $\theta_v = \theta_A(1 + Z_v q_A)$, q_A and θ_A are the lowest level atmospheric humidity, and potential temperature, respectively, and $Z_v = (\varrho(water)/\varrho(air)) - 1 = 0.606$.

In addition to surface fluxes, the atmospheric model requires effective surface albedos for both direct $\alpha(dir)$, and diffuse, $\alpha(dif)$, radiation at each wavelength. They are used in a single call to the computationally demanding atmospheric radiation routines. This call gives downward atmospheric albedo for diffuse radiation, $\alpha_a(dif)$. If direct and diffuse albedos, $\alpha^m(dir)$ and $\alpha^m(dif)$ for $m = 1, M$ different surfaces below an atmospheric grid cell are known, the multiple scattering coefficients, $R^m = (1 - \alpha_a(dif)\,\alpha^m(dif))^{-1}$ are computed, and with f^m the fractional coverage of each surface type, then the albedos are given by

$$
\begin{aligned}
\alpha(dif) &= (1 - F_2)\,/\,(1 - F_2\alpha_a(dif)) \\
\alpha(dir) &= F_1\,(1 - \alpha_a(dif)\,\alpha(dif))
\end{aligned}
$$

$$F_1 = \sum_{m=1}^{M} f^m \, R^m \, \alpha^m(dir)$$

$$F_2 = \sum_{m=1}^{M} f^m \, R^m \, (1 - \alpha^m(dif))$$

Use of these albedos ensures that the solar radiation exiting the bottom of the atmosphere is identical to the sum of M radiation calculations, each using an $\alpha^m(dif)$ and $\alpha^m(dir)$. At present, however, the albedo calculations are greatly simplified by assuming $\alpha_a(dif) = 0$, such that $R^m = 1$. This albedo then does not need to be passed from the atmosphere to the coupler, and the coupler simply computes the effective albedos to be passed to the atmosphere as

$$\begin{aligned} \alpha(dif) &= \sum_{m=1}^{M} f^m \, \alpha^m(dif) \\ \alpha(dir) &= \sum_{m=1}^{M} f^m \, \alpha^m(dir) \end{aligned}$$

In general the partition of radiation among the M surfaces is a function of R^m, $\alpha^m(dif)$, and $\alpha^m(dir)$, and hence of wavelength. Hence, correct partitioning would need to be performed for each wavelength band within the radiation transfer code of the atmospheric model. This procedure is greatly simplified by partitioning the net solar radiation within the Coupler.

16.2.2 Specific Expressions

At the atmosphere-ocean interface the near-surface air is also assumed to be saturated,

$$q = 0.98 \;\; \rho_A^{-1} \;\; C_5 \;\; \exp(C_6/T),$$

where the sea surface temperature is $T = SST$, $C_5 = 640380.0$ kg/m^3 and $C_6 = -5107.4$ K, and the factor 0.98 accounts for the salinity of the ocean. The differences (16.2) become

$$\begin{aligned} \Delta\vec{U} &= \vec{U}_A - \vec{U} \\ \Delta q &= q_A - q \\ \Delta\theta &= \theta_A - T, \end{aligned}$$

where the surface current, $\vec{U}$, is presently assumed to be negligible.

The roughness length for momentum, Z^o in meters, is a function of the atmospheric wind at 10 meters height, U_{10}:

$$Z^o = 10 \;\exp - \left[\kappa \left(\frac{C_1}{U_{10}} + C_2 + C_3 U_{10} \right)^{-1/2} \right],$$

where $C_1 = 0.0027$ m/s, $C_2 = 0.000142$, and $C_3 = 0.0000764$ m^{-1}s. The corresponding drag coefficient at 10m height and neutral stability is

$$C_{10}^N \;=\; C_1 \, U_{10}^{-1} \;+\; C_2 \;+\; C_3 \, U_{10} \;\; .$$

The roughness length for heat, Z^h, is a function of stability, and for evaporation, Z^e, is a different constant:

$$\begin{aligned} Z^h &= 2.2 \times 10^{-9} m \qquad \zeta > 0 \\ &= 4.9 \times 10^{-5} m \qquad \zeta \leq 0 \\ Z^e &= 9.5 \times 10^{-5} m. \end{aligned}$$

Since ζ is itself a function of the turbulent scales (16.2), and hence the fluxes, an iterative procedure is generally required to solve (16.1). First, ζ is set incrementally greater than zero when the air–sea temperature difference suggests stable stratification, otherwise it is set to zero. In either case, $\psi_m = \psi_s = 0$, and the initial transfer coefficients are then found from the roughness lengths at this ζ and $U_{10} = U_A$. As with sea–ice, these coefficients are used to approximate the initial flux scales (16.2) and the first iteration begins with an updated ζ and calculations of ψ_m and ψ_s. The wind speed, U_A, is then shifted to its equivalent neutral value at 10m height :

$$U_{10} = U_A \ \left(1 + \frac{\sqrt{C_{10}^N}}{\kappa} ln(\frac{Z_A}{10} - \psi_m(\zeta)) \right)^{-1} \quad .$$

This wind speed is used to update the transfer coefficients and hence the flux scales. The second and final iteration begins with another update of ζ. The final flux scales then give the fluxes calculated by (16.1).

16.3 Surface Albedo and Net Absorbed Solar Radiation

For the ice, land, and ocean components there are four surface albedos that are used by the atmosphere component to compute four corresponding components of downward shortwave. Subsequently, the four albedos, together with the four downward shortwave fields, are used to compute net absorbed shortwave flux from the atmosphere to the surface components:

$$SW_{net} = \sum_{m=1}^{4} SW_{down}^{m} \ (1 - \alpha^m) \quad ,$$

where $m = 1, ..., 4$ corresponds to near-infrared/diffuse, visible/diffuse, near-infrared/direct, and visible/direct shortwave components.

The ice and land components each compute their own surface albedos and, given the downward shortwave fields, compute their own net absorbed shortwave radiation. For the ocean component, it is the Coupler that computes the ocean surface albedo and computes net absorbed shortwave radiation. The Coupler then sends the net absorbed shortwave field to the ocean component.

16.3.1 Land Surface Albedos

The land surface albedos are computed by the land component and passed on to the atmosphere component. These albedos are not altered by the Coupler in any way. Note that the atmosphere component may be an active model computing downward shortwave once per hour, or it may be a data model feeding the Coupler a daily average downward shortwave. It is the user's responsibility to ensure that the albedos the land model sends are appropriate considering what type of downward shortwave fields the atmosphere component is providing.

16.3.2 Ice Surface Albedo

The ice surface albedos are computed by the ice component and passed through the Coupler and on to the atmosphere component. These albedos are "60 degree reference albedos" that have no diurnal cycle. Based on an input namelist variable, `flx_albav` (see the Section 4 of the User's Guide), the Coupler will either pass these albedos on to the atmosphere component unaltered (in which case they are considered *daily average* albedos), or impose a diurnal cycle on the albedos (in which case they are considered *instantaneous* albedos). When the Coupler does add a diurnal cycle to the ice albedo, this consists of merely setting the albedos to 1.0 on the dark side of the earth.

16.3.3 Ocean Surface Albedos

Unlike the ice and land components, the Coupler computes the ocean component's surface albedo. There are two ways the Coupler can compute the ocean albedo: with a diurnal cycle (*instantaneous*) or without a diurnal cycle (*daily average*). An input namelist variable (the `flx_albav` namelist variable) selects which option the Coupler implements.

If the albedos are computed as daily average albedos, then all four ocean albedos are set to 0.06 everywhere, regardless of time of day or time of year.

If the albedos are computed as instantaneous albedos, then all four ocean will be set to 1.0 on the dark side of the earth, and where the solar angle is greater than zero, the albedos are set to a value which has both an annual and a diurnal cycle. Ocean albedo distinguishes between direct and diffuse radiation. The direct albedo is solar zenith angle dependent, while the diffuse is not. There is no spectral dependence of the albedo, nor dependence on surface wind speed. The expressions for both direct and diffuse albedo are taken from Briegleb et al. (1986), based on fits to observations of ocean albedo, good to within $\pm 0.3\%$. The albedo expressions are valid for open ocean, and do not include the effects of suspended hydrosols in near-surface waters.

For complete details see Briegleb, B.P., P.Minnis, V.Ramanathan, and E.Harrison, 1986. "Comparison of regional clear-sky albedos inferred from satellite observations and model comparisons." *Journal of Climate and Applied Meteorology*, Vol. **25**, pp.214-226.

16.4 Area normalizing

Area normalizing is done in the Coupler to correct for slight differences between the total area of the sphere assumed by a model and the SCRIP program (Sec. 16.1). SCRIP calculates area weights for each grid as well as the mapping weights between two grids. However each model may have its own method for calculating the area of a grid cell. Thus when the conservative remapping is performed to interpolate a flux from one grid to another, it preserves the total flux over the sphere but the sphere may have a slightly different total area in the Coupler compared to each model. This will in turn effect global conservation of fluxed quantities. To correct for this effect, the Coupler multiplies all received fluxes by the ratio of the two areas immediately after receiving fluxes from a component:

$$Flux_{in\ coupler} = Flux_{from\ model} \frac{Area_{model}}{Area_{SCRIP}}$$

The corrected flux, $Flux_{in\ coupler}$ is then used within the coupler for mapping and any other calculations.

The Coupler receives $Area_{model}$ from each model during the CONTRACT initialization (Sec. 10.2) and stores it in a DOMAIN (Sec. 8.1) under the "area" attribute while $Area_{SCRIP}$ is read from the SCRIP mapping weight files during the MAP initialization and stored in the DOMAIN under the "aream" attribute. The area fractions are calculated using the `areafact_init` method from `areafact_mod.F90`. Before calculated or mapped fluxes are sent to a model from the Coupler, they are multiplied by $\frac{Area_{SCRIP}}{Area_{model}}$.

16.5 Merging and Fractional Weights

When two or more model's supply input to another model, the input field is formed by *merging* the two outputs. An example is the atmosphere where a atmosphere grid cell may overly both open ocean and sea ice covered ocean. In that case in CCSM3, the atmosphere-ice flux is calculated by the ice model while the atmosphere-ocean flux is calculated by the Coupler. Before sending to the atmosphere, these fluxes must be merged:

$$F_a = f_{ia}F_{i2a} + f_{la}F_{l2a} + f_{oa}F_{o2a}$$

where

$$f_{ia} + f_{la} + f_{oa} = 1.$$

F_a is the total flux into the atmosphere, f_{ia} is the *fraction of ice on the atmosphere grid*, an example of a surface fractional weight, and F_{i2a} is an ice-to-atmosphere flux calculated by the ice model. The other terms or for the land-atmosphere and ocean-atmosphere fluxes. States, such as surface temperature, are also merged this way. Note that this merge is performed *after* all fluxes have been mapped to the atmosphere grid. In that case f_{la} is always 1. Ocean fluxes must also be merged for ocean grid cells that are only partially ice covered. For example, the total momentum flux into the ocean is the weighted sum of the wind stress and the sea-ice drag.

The Coupler calculates all the time-invariant surface fractions using `frac_init` and updates the ice surface fractions to account for the time-varying extent of sea ice after each receive of data from the sea ice model using `frac_set`. Both methods are in `frac_mod.F90`.

17 Physical Constants

The CCSM code has a mechanism for facilitating the consistent use of constants among the various CCSM components – this is the "shared constants" module of the CCSM *shared code* library. The Coupler always uses shared constants whenever possible and only defines a constant locally if no such constant is available from the shared constants module.

The Coupler source code itself (found in `.../models/cpl/cpl6/`) is incomplete and cannot be compiled (due to missing subroutines) unless it is compiled along with *CCSM shared code* (found in `.../models/csm_share/`). This shared code includes the shared constants module.

Constants in the Coupler whose actual values come from the *shared constants...*

```
cpl_const_pi          = SHR_CONST_PI          ! pi
cpl_const_rearth      = SHR_CONST_REARTH      ! radius of earth ~ m
cpl_const_g           = SHR_CONST_G           ! acceleration of gravity
cpl_const_cpdair      = SHR_CONST_CPDAIR      ! specific heat of dry air
cpl_const_cpwv        = SHR_CONST_CPWV        ! specific heat of water vapor
cpl_const_cpvir       = cpl_const_cpwv/cpl_const_cpdair - 1.0
cpl_const_zvir        = SHR_CONST_ZVIR        ! rh2o/rair   - 1.0
cpl_const_latvap      = SHR_CONST_LATVAP      ! latent heat of evaporation
cpl_const_latice      = SHR_CONST_LATICE      ! latent heat of fusion
cpl_const_stebol      = SHR_CONST_STEBOL      ! Stefan-Boltzmann
cpl_const_karman      = SHR_CONST_KARMAN      ! Von Karman constant
cpl_const_ocn_ref_sal = SHR_CONST_OCN_REF_SAL ! ocn reference salinity
cpl_const_ice_ref_sal = SHR_CONST_ICE_REF_SAL ! ice reference salinity
cpl_const_spval       = SHR_CONST_SPVAL       ! special value
HFLXtoWFLX   = -(So-Si)/(So*latice)  ! converts heat to water for diagnostics
```

The actual values for the shared constants above...

```
SHR_CONST_PI     = 3.14159265358979323846  ! pi
SHR_CONST_REARTH = 6.37122e6    ! radius of earth ~ m
SHR_CONST_G      = 9.80616      ! acceleration of gravity ~ m/s^2
SHR_CONST_CPDAIR = 1.00464e3    ! specific heat of dry air ~ J/kg/K
SHR_CONST_CPWV   = 1.810e3      ! specific heat of water vap ~ J/kg/K
SHR_CONST_ZVIR   = (SHR_CONST_RWV/SHR_CONST_RDAIR)-1.0   ! RWV/RDAIR - 1.0
SHR_CONST_LATVAP = 2.501e6      ! latent heat of evaporation ~ J/kg
SHR_CONST_LATICE = 3.337e5      ! latent heat of fusion ~ J/kg
SHR_CONST_STEBOL = 5.67e-8      ! Stefan-Boltzmann constant ~ W/m^2/K^4
SHR_CONST_KARMAN = 0.4          ! Von Karman constant
SHR_CONST_OCN_REF_SAL = 34.7    ! ocn ref salinity (psu)
SHR_CONST_ICE_REF_SAL =  4.0    ! ice ref salinity (psu)

SHR_CONST_RWV   =
   SHR_CONST_RGAS/SHR_CONST_MWWV    ! Water vapor gas constant ~ J/K/kg
SHR_CONST_RDAIR =
   SHR_CONST_RGAS/SHR_CONST_MWDAIR  ! Dry air gas constant     ~ J/K/kg
SHR_CONST_RGAS  =
   SHR_CONST_AVOGAD*SHR_CONST_BOLTZ ! Universal gas constant   ~ J/K/mole
SHR_CONST_MWWV   = 18.016           ! molecular weight water vapor
SHR_CONST_MWDAIR = 28.966           ! molecular weight dry air ~ kg/kmole
SHR_CONST_BOLTZ  = 1.38065e-23      ! Boltzmann's constant ~ J/K/molecule
SHR_CONST_AVOGAD = 6.02214e26       ! Avogadro's number ~ molecules/kmole
```

Constants in the Coupler whose values do not come from the *shared constants...*

```
albdif = 0.06   ! 60 deg reference albedo, diffuse for ocn albedo calc
albdir = 0.07   ! 60 deg reference albedo, direct  for ocn albedo calc
umin   =  0.5   ! minimum wind speed (m/s)       for atm/ocn flux calc
zref   = 10.0   ! reference height (m)           for atm/ocn flux calc
ztref  =  2.0   ! reference height for air T (m) for atm/ocn flux calc
spval  = 1.0e30 ! flags special value (missing value)
```

Part IV
Appendix

18 Glossary

`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`
: When two or more coupled system components are executing at the same time.

`domain`
: A grid together with a domain mask.

`domain mask`
: A mask that indicates which grid cells are active or inactive.

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

`map`
: Same as "regrid"; interpolate data from one domain to another.

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

`merge`
: Combining two or more fields into one unified field.

`regrid`
: Same as "map"; interpolate data from one domain to another.

`sequential execution`
: When two or more coupled system components execute in turn.

`timestep`
: A model's internal integration increment in time. Note: the Coupler does not know a model's internal timestep, it only knows a model's communication interval.