CIME Users Guide (PDF)

version

Mariana Vertenstein, Rob Jacob, Jim Edwards, Jim Foucar, Alice Bertini, Bill Sacks

April 25, 2017

Contents

Welcome to the CIME documentation!

What is CIME?

User's guide - Part 1: basic use of the scripting infrastructure.

User's guide - Part 2: CIME internals, porting, testing and use cases

Data Models: design and usage of the data models.

Driver/Coupler (cpl7): design and usage of the driver/coupler.

Building a Coupled Model with CIME: how to bring a new model in to CIME.

Miscellaneous Tools: various tools for climate modeling

Table of contents

What is CIME?

CIME contains the support scripts (configure, build, run, test), data models, essential utility libraries, a “main” and other tools that are needed to build a single-executable coupled Earth System Model. CIME is available in a stand-alone package that can be compiled and tested without active prognostic components but is typically included in the source of a climate model. CIME does not contain: any active components, any intra-component coupling capability (such as atmosphere physics-dynamics coupling).

Overview

CIME is comprised of:

1. A default coupled model architecture:
    i. A programmer interface and libraries to implement a hub-and-spoke inter-component coupling architecture.
    ii. An implementation of a “hub” that needs 7 components (atm, ocn, lnd, sea-ice, land-ice, river, wave). a.k.a. “the driver”.
    iii. The ability to allow active and data components to be mixed in any combination as long as each component implements the coupling programmer interface.
2. Non-active Data and Stub components:
    i. “Data-only” versions of 6 of the 7 components that can replace active components at build-time.
    ii. “Stub” versions of all 7 components for building a complete system.
3. **Additional libraries useful in scientific applications in general and climate models in particular.**
    i. Parallel I/O library.
    ii. The Model Coupling Toolkit.
    iii. Timing library.
4. A system of scripts (python) to support case configuration, executable compilation, workflow, system testing and unit testing infrastructure:
    i. Scripts to enable simple generation of model executables and associated input files for different scientific cases, component resolutions and combinations of full, data and stub components with a handful of commands.
    ii. Testing utilities to run defined system tests and report results for different configurations of the coupled system.
5. Additional stand-alone tools:
    i. Parallel regridding weight generation program

ii. Scripts to automate off-line load-balancing.

iii. Scripts to conduct ensemble-based statistical consistency tests.

iv. Netcdf file comparison program (for bit-for-bit).

Where is CIME developed?

At http://github.com/ESMCI/cime

Indices and tables

- **genindex**
- **modindex**
- **search**

CIME User's Guide Part 1: Basic Usage

Introduction and Overview

The Common Infrastructure for Modeling the Earth (CIME) provides a UNIX command-line-based user interface for performing all of the necessary functions to configure, compile and execute an earth system model. Part 1 of this guide will explain all of the basic commands needed to get a model running.

Prerequisites

You should be familiar with the basic concepts of climate modeling.

You should be familiar with UNIX command line terminals and the UNIX development environment.

CIME's commands are python scripts and require a correct version of the Python interpreter to be installed before any use.

The python version must be greater then 2.7 but less the 3.0. You can check which version of python you have by typing:

```
> python --version
```

NOTE: Part 1 of this user's guide assumes that CIME and necessary input files have already been installed on the computer you are using. If it is not, see Installing CIME.

Terms and concepts

The following concepts are ingrained in CIME and will occur frequently in this documentation.

component set or compset:

CIME allows several sub-models and other tools to be linked together in to a climate model. These sub-models and tools are called *components* of the climate model. We say a climate model has an atmosphere component, an ocean component, etc. The resulting set of components is called the *component set* or *compset*.

active vs data vs stub models:

A component for the atmosphere or ocean that solve a complex set of equations to describe their behavior are called *active* models, and will sometimes be refered to as *prognostic* or *full* models.

CIME recognizes 7 different active models of a climate model, those are:

atmosphere, ocean, sea-ice, land surface, river, glacier, wave

In addition, an "external processing system" or ESP component is also allowed.

For some climate problems, its necessary to reduce feedbacks within the system by replacing a active model with a version that sends and receives the same variables to and from other models but the values sent are not computed from the equations but instead read from files. The values received are ignored. We call these active-model substitutes *data models* and CIME provides data models for each of the supported components.

For some configurations, a data model is not needed and so CIME provides *stub* versions that simply occupy the required place in the climate execution sequence and do not send or receive any data.

grid set or grid:

Each active model must solve its equations on a numerical grid. CIME allows several models within the system to have different grids. The resulting set of numerical grids is called the *grid set* or sometimes just the *grid* where *grid* is a unique abbreviation denoting a set of numerical grids. Sometimes the *resolution* will also refer to a specific set of grids with different resolutions.

machine:

The computer you are using to run CIME and build and run the climate model is called the *machine*. It could be a workstation or a national supercomputer. The *machine* is typically the UNIX host name but could be any string.

compiler:

CIME will control compiling the source code (Fortran, C and C++) of your model in to an executable. Some machines support multiple compilers and so you may need to specify which one to use.

case:

The most important concept in CIME is a *case*. To build and execute a CIME-enabled climate model, you have to make choices of compset, grid set, machine and compiler. The collection of these choices, and any additional customizations you may make, is called the *case*.

Setting defaults

Before using any CIME commands, you should set the CIME_MODEL environmental variable. In csh, this would be:

```
setenv CIME_MODEL <model>
```

Where <model> is one of "acme" or "cesm".

Directory content

If you use CIME as part of a climate model or stand alone, the content of the cime directory is the same.

If you are using it as part of a climate model, cime is usually one of the first subdirectories under the main directory:

CIME directory in a climate model

Directory or Filename	Description
README, etc.	typical top-level directory content.
components/	source code for all of the active models.
cime/	All of CIME code

CIME's content is split in to several subdirectories:

CIME directory content

Directory or Filename	Description
CMakeLists.txt	For building with CMake
ChangeLog	Developer-maintained record of changes to CIME
ChangeLog_template	template for an entry in ChangeLog
LICENSE.TXT	The CIME license
README	Brief intro to CIME
README.md	README in markdown language
README.unit_testing	Instructions on running unit tests with CIME
cime_config/	shared and model-specific configuration files

components/	CIME-provided components including data and stub models
driver_cpl/	CIME-provided main driver for a climate model
externals/	Software provided with CIME for building a climate model
scripts/	The CIME user interface
share/	Source code provided by CIME and used by multiple components
tests/	tests
tools/	Stand-alone tools useful for climate modeling
utils/	source code for CIME scripts and functions

Users should start in the "scripts" subdirectory.

Discovering available cases

You can find what compsets, grids and machines your CIME-enabled model supports using the manage_case command found in cime/scripts. Use the "--help" option for more information.

```
> ./manage_case --help
```

Quick Start

If you would like to quickly see how a case is created, configured, built and run with CIME, try these commands (assuming CIME has been ported to your current machine):

```
> cd cime/scripts
> ./create_newcase --case mycase --compset X --res f19_g16
> cd mycase
> ./case.setup
> ./case.build
> ./case submit
```

The output from each command will be explained in the sections below. You can follow progress by monitoring the CaseStatus file:

```
> tail CaseStatus
```

Repeat the above command until you see the message "Run SUCCESSFUL". This tells you the case finished successfully.

The basics of CIME cases

Two neccessary concepts to understand before working with CIME are component sets and model grids.

- Component sets (usually referred to as *compsets*) define both the specific model components that will be used in a CIME case, *and* any component-specific namelist or configuration settings that are specific to this case.
- Model grids specify the grid for each component making up the model.
- At a minimum creating a CIME experiment - referred to normally as a case - requires specifying a component set and a model grid.
- Out of the box compsets and models grids are associated with two names: a longname and an alias name.
- Aliases are required by the CIME regression test system but can also be used for user convenience. Compset aliases are unique - each compset alias is associated with one and only one compset. Grid aliases, on the other hand, are overloaded and the same grid alias may result in a different grid depending on the compset the alias is associated with. We recommend that the user always confirm that the compset longname and grid longname are the expected result when using aliases to create a case.

Component Sets

The component set (compset) longname has the form:

```
TIME_ATM[%phys]_LND[%phys]_ICE[%phys]_OCN[%phys]_ROF[%phys]_GLC[%phys]_WAV[%phys]_ESP[_BGC%p

TIME = model time period (e.g. 2000, 20TR, RCP8...)

CIME supports the following values for ATM,LND,ICE,OCN,ROF,GLC,WAV and ESP
ATM  = [DATM, SATM, XATM]
LND  = [DLND, SLND, XLND]
ICE  = [DICE, SICE, SICE]
OCN  = [DOCN, SOCN, XOCN]
ROF  = [DROF, SROF, XROF]
GLC  = [SGLC, XGLC]
WAV  = [SWAV, XWAV]
ESP  = [SESP]

If CIME is run with CESM active components, the following additional values are permitted:
ATM  = [CAM40, CAM50, CAM55, CAM60]
LND  = [CLM45, CLM50]
ICE  = [CICE]
OCN  = [POP2, AQUAP]
ROF  = [RTM, MOSART]
GLC  = [CISM1, CISM2]
WAV  = [WW]
BGC  = optional BGC scenario

If CIME is run with ACME active components, the following additional values are permitted:
ATM  = []
LND  = []
ICE  = []
OCN  = []
ROF  = []
GLC  = []
WAV  = []
BGC  = optional BGC scenario

The OPTIONAL %phys attributes specify sub-modes of the given system
For example DOCN%DOM is the DOCN data ocean (rather than slab-ocean) mode.
ALL the possible %phys choices for each component are listed by
calling **manage_case** with the **-list** compsets argument.
ALL data models have a %phys option that corresponds to the data model mode.
```

As an example, the CESM compset longname:

```
1850_CAM60_CLM50%BGC_CICE_POP2%ECO_MOSART_CISM2%NOEVOLVE_WW3_BGC%BDRD
```

refers to running a pre-industrial control with active CESM components CAM, CLM, CICE, POP2, MOSART, CISM2 and WW3 in a BDRD BGC coupling scenario. The alias for this compset is B1850. Either a compset longname or a compset alias can be used as input to **create_newcase**. It is also possible to create your own custom compset (see *How do I create my own compset? in the FAQ*)

Model Grids

The model grid longname has the form:

```
a%name_l%name_oi%name_r%name_m%mask_g%name_w%name

a%  = atmosphere grid
l%  = land grid
oi% = ocean/sea-ice grid (must be the same)
r%  = river grid
m%  = ocean mask grid
g%  = internal land-ice (CISM) grid
w%  = wave component grid
```

```
The ocean mask grid determines land/ocean boundaries in the model.
It is assumed that on the ocean grid, a gridcell is either all ocean or all land.
The land mask on the land grid is then obtained by mapping the ocean mask
(using first order conservative mapping) from the ocean grid to the land grid.

From the point of view of model coupling - the glc (CISM) grid is assumed to
be identical to the land grid. However, the internal CISM grid can be different,
and is specified by the g% value.
```

As an example, the longname:

```
a%ne30np4_l%ne30np4_oi%gx1v6_r%r05_m%gx1v6_g%null_w%null
```

refers to a model grid with a ne30np4 spectral element 1-degree atmosphere and land grids, gx1v6 Greenland pole 1-degree ocean and sea-ice grids, a 1/2 degree river routing grid, null wave and internal cism grids and an gx1v6 ocean mask. The alias for this grid is ne30_g16. Either the grid longname or alias can be used as input to **create_newcase**.

CIME also permits users to introduce their own <user defined grids.

Component grids (such as the atmosphere grid or ocean grid above) are denoted by the following naming convention:

- "[dlat]x[dlon]" are regular lon/lat finite volume grids where dlat and dlon are the approximate grid spacing. The shorthand convention is "fnn" where nn is generally a pair of numbers indicating the resolution. An example is 1.9x2.5 or f19 for the approximately "2-degree" finite volume grid. Note that CAM uses an [nlat]x[nlon] naming convention internally for this grid.
- "Tnn" are spectral lon/lat grids where nn is the spectral truncation value for the resolution. The shorthand name is identical. An example is T85.
- "ne[X]np[Y]" are cubed sphere resolutions where X and Y are integers. The short name is generally ne[X]. An example is ne30np4 or ne30.
- "pt1" is a single grid point.
- "gx[D]v[n]" is a displaced pole grid where D is the approximate resolution in degrees and n is the grid version. The short name is generally g[D][n]. An example is gx1v7 or g17 for a grid of approximately 1-degree resolution.
- "tx[D]v[n]" is a tripole grid where D is the approximate resolution in degrees and n is the grid version.

Querying CIME - calling manage_case

The utility **$CIMEROOT/scripts/manage_case** permits you to query the out-of-the-box compsets, grids and machines that are available for either CESM or ACME. If CIME is downloaded in a stand-alone mode, then this will only permit you to query the stand-alone CIME compsets. However, if CIME is part of a larger checkout that includes the prognostic components of either CESM or ACME, then **manage_case** will allow you to query all prognostic component compsets as well.

manage_case usage is summarized below:

```
usage: manage_case [-h] [-d] [-v] [-s]
                   [--query-compsets-setby QUERY_COMPSETS_SETBY]
                   [--query-component-name QUERY_COMPONENT_NAME]
                   [--query-machines] [--long]

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Print debug information (very verbose) to file /glade/
                        p/work/mvertens/cime.data_model_fields/scripts/manage_
                        case.log
  -v, --verbose         Add additional context (time and file) to log messages
  -s, --silent          Print only warnings and error messages
```

```
  --query-compsets-setby QUERY_COMPSETS_SETBY
                         Query compsets that are set by the target component
                         for cesm model
  --query-component-name QUERY_COMPONENT_NAME
                         Query component settings that are set by the target
                         component for cesm model
  --query-grids          Query supported model grids for cesm model
  --query-machines       Query supported machines for cesm model
  --long                 Provide long output for queries
```

To query manage-case for compset information, use the `--query-compsets-setby` option. For CESM, the value of `QUERY_COMPSETS_SETBY` can be one of the following:

```
allactive, drv, cam, cism, clm, cice, pop
```

As an example, If you want to see what the compsets are for stand-alone CIME, issue the command

```
manage_case --query-compsets-setby drv
```

And the output will be

```
-------------------------------------
Compset Short Name: Compset Long Name
-------------------------------------
         A : 2000_DATM%NYF_SLND_DICE%SSMI_DOCN%DOM_DROF%NYF_SGLC_SWAV
      AWAV : 2000_DATM%WW3_SLND_DICE%COPY_DOCN%COPY_SROF_SGLC_WW3
         S : 2000_SATM_SLND_SICE_SOCN_SROF_SGLC_SWAV_SESP
ADESP_TEST : 2000_DATM%NYF_SLND_DICE%SSMI_DOCN%DOM_DROF%NYF_SGLC_SWAV_DESP%TEST
         X : 2000_XATM_XLND_XICE_XOCN_XROF_XGLC_XWAV
     ADESP : 2000_DATM%NYF_SLND_DICE%SSMI_DOCN%DOM_DROF%NYF_SGLC_SWAV_DESP
      AIAF : 2000_DATM%IAF_SLND_DICE%IAF_DOCN%IAF_DROF%IAF_SGLC_SWAV
```

CIME only sets compsets associated with stand-alone CIME - i.e. primarily compsets associated only with data models. Each prognostic component that is CIME compliant is responsible for setting those compsets that have the appropriate target feedbacks turned off. As an example, in CESM, CAM is responsible for setting all compsets that have CAM, CLM, CICE (in prescribed ice mode) and DOCN as the components.

To query manage_case for component-specific compsets settings, use the `--query-component-name` option.

Every model component specifies its own definitions of what can appear after its `%` modifier in the compset longname, (e.g. `DOM` in `DOCN%DOM`).

For CESM, the value of `QUERY_COMPONENT_NAME` can be one of the following:

```
cam, datm, clm, dlnd, cice, dice, pop, aquap, docn, socn, rtm, mosart, drof, cism, ww3, c
```

If you want to see what supported modifiers are for `DOCN`, issue the command

```
manage_case --query-compsets_setby docn
```

and the output will be

```
=========================================
DOCN naming conventions
=========================================

_DOCN%NULL : docn null mode:
_DOCN%COPY : docn copy mode:
_DOCN%US20 : docn us20 mode:
_DOCN%DOM  : docn data mode:
_DOCN%SOM  : docn slab ocean mode:
_DOCN%IAF  : docn interannual mode:
```

To query **manage_case** for the out-of-the box model grids that are supported, use the `query-grids` option.

To query **manage_case** for the out-of-the box machines that are supported, use the `query-machines` option.

For more details of how CIME determines the output for **manage_case** see CIME internals.

Creating a Case

Calling create_newcase

The first step in creating a CIME based experiment is to use **create_newcase**.

If you are not on an out-of-the box CIME supported platform, you will need to first port CIME to your system.

You should first use the --help option in calling **create_newcase** to document its input options. The only required arguments to **create_newcase** are:

```
create_newcase --case [CASE] --compset [COMPSET] --res [GRID]
```

CIME supports out of the box `component sets`, `model grids` and `hardware platforms`.

- Component sets (usually referred to as *compsets*) define both the specific model components that will be used in a CIME case, *and* any component-specific namelist or configuration settings that are specific to this case.
- Model grids specify the grid for each component making up the model.
- At a minimum creating a CIME experiment requires specifying a component set and a model grid.
- Out of the box compsets and models grids are associated with two names: a longname and an alias name.
- Aliases are required by the CIME regression test system but can also be used for user convenience. Compset aliases are unique - each compset alias is associated with one and only one compset. Grid aliases, on the other hand, are overloaded and the same grid alias may result in a different grid depending on the compset the alias is associated with. We recommend that the user always confirm that the compset longname and grid longname are the expected result when using aliases to create a case.

Result of calling create_newcase

Following is a simple example of using **create_newcase** using aliases for both compset and grid names. The complete example appears in the basic example. In what follows, `$CIMEROOT` is the full pathname of the root directory of the CIME distribution.

```
> cd $CIMEROOT/scripts
> create_newcase --case ~/cime/example1 --compset A --res f09_g16_rx1
```

This example

- creates the `$CASEROOT` directory `~/cime/example1` (if the directory already exists, a warning is printed and `create_newcase` aborts)
- `$CASE` is `"example1"` (`$CASE` can include letters, numbers, ".", and "_")
- the model resolution is `a%0.9x1.25_l%0.9x1.25_oi%gx1v6_r%r05_m%gx1v6_g%null_w%null`
- the compset is `2000_DATM%NYF_SLND_DICE%SSMI_DOCN%DOM_DROF%NYF_SGLC_SWAV`.
- in `$CASEROOT`, **create_newcase** installs files to build and run the model and optionally perform archiving of the case on the target platform.

Various scripts, files and directories are created in `$CASEROOT` by **create_newcase**:

- `user scripts`

case.setup	Script used to set up the case (create the case.run script, the Macros file and user_nl_xxx files)
case.build	Script to build component and utility libraries and model executable
case.submit	Script to submit the case to run using the machine's batch queueing system

case.st_archive	Script to perform short-term archiving of output data
case.lt_archive	Script to perform long-term archiving of output data
xmlchange	Script to modify values in the xml files
xmlquery	Script to query values in the xml files
preview_namelists	Script for users to see their component namelists in `$CASEROOT/CaseDocs` before running the model ***Warning*** the namelists generated in `$CASEROOT/CaseDocs` should not be edited by the user they are only there to document model behavior.
check_input_data	Script for checking for various input datasets and moves them into place.
pelayout	Script to query and modify the NTASKS, ROOTPE, and NTHRDS for each component model. This a convenience script that can be used in place of xmlchange and xmlquery.

- `XML files`

env_mach_specific.xml	Sets a number of machine-specific environment variables for building and/or running. You can edit this file at any time.
env_case.xml	Sets case specific variables (e.g. model components, model and case root directories). Cannot be modified after a case has been created. To make changes, your should re-run **create_newcase** with different options.
env_build.xml	Sets model build settings. This includes component resolutions and component compile-time configuration options. You must run the case.build command after changing this file.
env_mach_pes.xml	Sets component machine-specific processor layout (see changing pe layout). The settings in this are critical to a well-load-balanced simulation (see load balancing).
env_run.xml	Sets run-time settings such as length of run, frequency of restarts, output of coupler diagnostics, and short-term and long-term archiving. This file can be edited at any time before a job starts.
env_batch.xml	Sets batch system specific settings such as wallclock time and queue name.

- `User Source Mods Directory`

SourceMods	Top-level directory containing sub-directories for each compset component where you can place modified source code for that component.

- `Provenance`

README.case	File detailing **create_newcase** usage. This is a good place to keep track of runtime problems and changes.
CaseStatus	File containing a list of operations done in the current case.

- non-modifiable work directories

Buildconf/	Work directory containing scripts to generate component namelists and component and utility libraries (e.g., PIO, MCT) You should never have to edit the contents of this directory.
LockedFiles/	Work directory that holds copies of files that should not be changed. Certain xml files are *locked* after their variables have been used by should no longer be changed. CIME does this by *locking* a file and not permitting you to modify that file unless, depending on the file, **case.setup --clean** or **case.build --clean** are called.
Tools/	Work directory containing support utility scripts. You should never need to edit the contents of this directory.

In CIME, the $CASEROOT xml files are organized so that variables can be locked after different phases of the **create_newcase** and **case.setup**. Locking these files prevents users from changing variables after they have been resolved (used) in other parts of the scripts system. CIME locking currently does the following: - variables in `env_case.xml` are locked after **create_newcase**. - variables in `env_mach_pes.xml` are locked after **case.setup**. - variables in `env_build.xml` are locked after completion of **case.build**. - variables in `env_run.xml`, `env_batch.xml` and `env_archive.xml` variables are never locked and most can be changed at anytime. There are some exceptions in the env_batch.xml file.

These files can be "unlocked" as follows. - `env_case.xml can never by unlocked` - **case.setup --clean** unlocks `env_mach_pes.xml` - **case.build --clean** unlocks `env_build.xml`

Setting up a Case

Calling case.setup

After creating a case or changing aspects of the case such as the pe-layout, you need to call the **case.setup** command from $CASEROOT. To see the options to **case.setup** use the `--help` option.

Calling `case.setup` creates the following **additional** files and directories in $CASEROOT:

case.run	Run script containing the batch directives The batch directives are generated using the contents of env_mach_pes.xml. Calling **case.setup --clean** will remove this file.
Macros.make	File containing machine-specific makefile directives for your target platform/compiler. This file is only created if it does not already exist in the case is called. This file can be modified by the user to change certain aspects of the build such as compiler flags. Calling **case.setup -clean** will not remove the Macros.make file once it has been created. However if you remove or rename the Macros.make file case.setup will recreate it for you.
user_nl_xxx[_NNNN] files	Files where all user modifications to component namelists are made. xxx denotes any one of the set of components targeted for the specific case For example, for a full active CESM compset, xxx would denote [cam,clm,rtm,cice,pop2,cism2,ww3,cpl] NNNN goes from 0001 to the number of instances of that component (see multiple instances) For a case with 1 instance of each component (default), NNNN will not appear in the user_nl file names. A user_nl file of a given name will only be created once. Calling **case.setup -clean** will *not remove* any user_nl files. Changing the number of instances in the `env_mach_pes.xml` will only cause new user_nl files to be added to $CASEROOT.

CaseDocs/	Directory that contains all the component namelists for the run. This is for reference only and files in this directory SHOULD NOT BE EDITED since they will be overwritten at build time and run time.
.env_mach_specific.[cs,sh]	Files summarizing the module load commands and environment variables that are set when the scripts in `$CASEROOT` are called. These files are not used by the case but can be useful for debugging case module load and environment settings.
software_environment.txt	This file records some aspects of the computing system the case is built on, such as the shell environment.

Building a Case

The following summarizes details of building the model exectuable.

Calling case.build

After calling **case.setup**, you can build the model executable by running **case.build**. Running this will:

1. Create the component namelists in `$RUNDIR` and `$CASEROOT/CaseDocs`.
2. Create the necessary utility libraries for `mct`, `pio`, `gptl` and `csm_share`. These will be placed in a path below `$SHAREDLIBROOT`.
3. Create the necessary component libraries. These will be placed in `$EXEROOT/bld/lib`.
4. Create the model executable (`$MODEL.exe`). This will be placed in `$EXEROOT`.

`$CASEROOT/Tools/Makefile` and `$CASEROOT/Macros.make` are used to generate the utility libraries, the component libraries and the model executable. You do not need to change the default build settings to create the executable. However, since the CIME scripts provide you with a great deal of flexibility in customizing various aspects of the build process, it is useful to become familiar with these in order to make optimal use of the system.

The `env_build.xml` variables control various aspects of building the model executable. Most of the variables should not be modified by users. Among the variables that you can modify are `$BUILD_THREADED`, `$DEBUG` and `$GMAKE_J`. The best way to see what xml variables are in your `$CASEROOT` is to use the `xmlquery` command. To see its usage, simply call

```
./xmlquery --help
```

To build the model:

```
> cd $CASEROOT
> ./case.build
```

Diagnostic comments will appear as the build proceeds.

`case.build` generates the utility and component libraries and the model executable, as well as build logs for each component. Each build log file is date stamped, and a pointer to the build log file for that library or component is shown. The build log files have names of the form $component.bldlog.$datestamp and are located in `$BLDDIR`. If they are compressed (indicated by a .gz file extension), then the build ran successfully.

Invoking **case.build** creates the following directory structure in `$EXEROOT` if the `intel` compiler is used:

```
atm/, cpl/, esp/, glc/, ice/, intel/, lib/, lnd/, ocn/, rof/, wav/
```

All of the above directories, other than `intel/` and `lib/`, each contain an `obj/` directory where the compiled object files for the target model component are placed. These object files are collected into libraries that are placed in `lib/`. The `mct`, `pio`, `gptl`, and `csm_share` libraries are placed in a directory tree reflecting dependancies, see the bldlog for a given component to locate the library for that component. Special include modules are also placed in `lib/include`. The model executable (either `cesm.exe` or `acme.exe`) is placed directly in `$EXEROOT`. Component namelists, component logs, output datasets, and restart files are placed in `$RUNDIR`. It is important to note that `$RUNDIR` and `$EXEROOT` are independent variables which are set `$CASEROOT/env_run.xml`.

Rebuilding the model

You should rebuild the model under the following circumstances:

If either env_build.xml or Macros.make has been modified, and/or if code is added to SourceMods/src.*, then it's safest to clean the build and rebuild from scratch as follows,

```
> cd $CASEROOT
> ./case.build --clean
```

If you have ONLY modified the PE layout in env_mach_pes.xml, then it's possible that a clean is not required.

```
> cd $CASEROOT
> ./case.build
```

But if the threading has been turned on or off in any component relative to the previous build, then the build script should fail with the following error

```
ERROR SMP STATUS HAS CHANGED
SMP_BUILD = a0l0i0o0g0c0
SMP_VALUE = a1l0i0o0g0c0
A manual clean of your obj directories is strongly recommended
You should execute the following:
   ./case.build --clean
   ./case.build

 ---- OR ----
 You can override this error message at your own risk by executing
   ./xmlchange SMP_BUILD=0
 Then rerun the build script interactively
```

and suggest that the model be rebuilt from scratch.

You are responsible for manually rebuilding the model when needed. If there is any doubt, you should rebuild.

case.build --clean will clean all of the model components but not the support libraries such as mct and gptl. Use case.build --clean-all to clean everything associated with the build. You can also clean a specific component only using case.build --clean compname where compname is the name of the component you want to clean (eg 'atm', 'clm', 'pio' ...) see case.build --help for details.

Input Data

All active and data components use input datasets. A local disk needs the directory tree specified by the xml variable $DIN_LOC_ROOT to be populated with input data in order to run CIME and the CIME complaint active components. Input data is provided as part of the CIME release via data from a subversion input data server. Data is downloaded from the server on an as needed basis determined by the case, data may already exists in the default local filesystem input data area as specified by $DIN_LOC_ROOT (see below). There is an extensive amount of input data available and it can take significant space on a system, it is recommended that users share a common $DIN_LOC_ROOT directory on each system if possible.

Input data is handled by the build process as follows:

- The buildnml scripts in the various component cime_config directories create listings of required component input datasets in the Buildconf/$component.input_data_list files.
- check_input_data (which is called by case.build) checks for the presence of the required input data files in the root directory $DIN_LOC_ROOT.
    - If all required data sets are found on local disk, then the build can proceed.
    - If any of the required input data sets are not found locally, the files that are missing will be listed. At this point, you must obtain the required data from the input data server using check_input_data with the -export option.

The env_run.xml variables $DIN_LOC_ROOT and $DIN_LOC_ROOT_CLMFORC determine where you should expect input data to reside on local disk.

Running a Case

Calling case.submit

Before you submit the case using case.submit, you need to make sure the batch queue variables are set correctly for the specific run being targeted. The batch submissions variables are contained in the file `$CASEROOT/env_batch.xml` under the XML `<group id="case.run">` and `<group id="case.st_archive">` elements. Make sure that you have appropriate account numbers (`PROJECT`), time limits (`JOB_WALLCLOCK_TIME`), and queue (`JOB_QUEUE`) for those groups..

You should also modify `$CASEROOT/env_run.xml` for your particular needs using xmlchange.

Once you have run **case.setup** and **case.build**, you need to run **case.submit** to submit the run to your machine's batch queue system.

```
> cd $CASEROOT
> ./case.submit
```

Result of running case.submit

When called, the script, `case.submit` will:

- Load the necessary environment.
- Check that locked files (in `$CASEROOT/LockedFiles`) are consistent with the current xml files.
- Run **preview_namelist** which in turn will run each component's **buildnml**
- Run **check_input_data** to verify that the required input data for the case is present on local disk in `$DIN_LOC_ROOT`.
- Submit the job to the batch queue which in turn will run the `$CASEROOT` script **case.run**.

Upon successful completion of the run, **case.run** will:

- Put timing information in `$CASEROOT/timing`. A summary timing output file is produced after every model run. This file is `$CASEROOT/timing/ccsm_timing.$CASE.$date.gz` (where `$date` is a datestamp set by CIME at runtime) and contains a summary of various information. See model timing data for more details.
- Copy log files back to `$LOGDIR`.
- Submit the short term archiver script **case.st_archive** to the batch queue if `$DOUT_S` is TRUE.
- Resubmit **case.run** if `$RESUBMIT` >0.

Short term archiving will copy and move component history, log, diagnostic, and restart files from `$RUNDIR` to the short-term archive directory, `$DOUT_S_ROOT`.

Monitoring case job statuses

The `$CASEROOT/CaseStatus` file contains a log of all the job states and xmlchange commands in chronological order. An example of status messages from a CaseStatus file with successful job completions includes:

```
2017-02-14 15:29:50: case.setup starting
---------------------------------------------------
2017-02-14 15:29:54: case.setup success
---------------------------------------------------
2017-02-14 15:30:58: xmlchange success <command> ./xmlchange STOP_N=2,STOP_OPTION=nmonths  <
---------------------------------------------------
2017-02-14 15:31:26: xmlchange success <command> ./xmlchange STOP_N=1  </command>
---------------------------------------------------
2017-02-14 15:33:51: case.build starting
---------------------------------------------------
2017-02-14 15:53:34: case.build success
---------------------------------------------------
2017-02-14 16:02:35: case.run starting
---------------------------------------------------
2017-02-14 16:20:31: case.run success
---------------------------------------------------
2017-02-14 16:20:45: st_archive starting
---------------------------------------------------
```

```
2017-02-14 16:20:58: st_archive success
---------------------------------------------------
```

After the short term archiver completes, data from the $RUNDIR is in the subdirectories under $DOUT_S_ROOT.

Note

Once you have a successful first run, you must set the env_run.xml variable $CONTINUE_RUN to TRUE in env_run.xml before resubmitting, otherwise the job will not progress. You may also need to modify the env_run.xml variables $STOP_OPTION, $STOP_N and/or $STOP_DATE as well as $REST_OPTION, $REST_N and/or $REST_DATE, and $RESUBMIT before resubmitting.

You can see a complete example of how to run a case in the basic example.

Troubleshooting a job that fails

There are several places where you should look for information if a job fails. Start with the STDOUT and STDERR file(s) in $CASEROOT. If you don't find an obvious error message there, the $RUNDIR/$model.log.$datestamp files will probably give you a hint. First check cpl.log.$datestamp, because it will often tell you when the model failed. Then check the rest of the component log files. Please see troubleshooting run-time problems for more information.

Input data

The $CASEROOT script **check_input_data** determines if the required data files for your case exist on local disk in the appropriate subdirectory of $DIN_LOC_ROOT. As part of submitting the run to the batch queueing system, **case.submit** also calls **check_input_data** to verify that required input data sets are accessible on local disk and download missing data automatically to local disk from in the appropriate subdirectory of $DIN_LOC_ROOT

The required input datasets needed for each component are found in the $CASEROOT/Buildconf directory in the files xxx.input_data_list, where xxx is the component name. These files are generated via a call to **preview_namlists** and are in turn created by each component's **buildnml** script. As an example, for compsets consisting only of data models (i.e. A compsets), the following files will be created in $CASEROOT/Buildconf when **case.submit** is called:

```
datm.input_data_list
dice.input_data_list
docn.input_data_list
drof.input_data_list
```

check_input_data verifies that each file listed in the xxx.input_data_list files exists in $DIN_LOC_ROOT. If any of the required datasets do not exist locally, **check_input_data** provides the capability for downloading them to the $DIN_LOC_ROOT directory hierarchy via interaction with the Subversion input data server. You can independently verify that the required data is present locally by using the following commands:

```
> cd $CASEROOT
> check_input_data -help
> check_input_data -inputdata $DIN_LOC_ROOT -check
```

If input data sets are missing, you must obtain the datasets from the input data server:

```
> cd $CASEROOT
> check_input_data -inputdata $DIN_LOC_ROOT -export
```

Required data files not on local disk will be downloaded through interaction with the Subversion input data server. These will be placed in the appropriate subdirectory of $DIN_LOC_ROOT.

Controlling starting, stopping and restarting a run

env_run.xml contains variables which may be modified at initialization and any time during the course of the model run. Among other features, the env_run.xml file variables comprise coupler namelist settings for the model stop time, model restart frequency, coupler history frequency and a flag to determine if the run should be flagged as a continuation run. At a minimum, you will only need to set the variables $STOP_OPTION and $STOP_N. The other

driver namelist settings will then be given consistent and reasonable default values. These default settings guarantee that restart files are produced at the end of the model run.

Run-type initialization

The case initialization type is set using the $RUN_TYPE variable in env_run.xml. A CIME run can be initialized in one of three ways; startup, branch, or hybrid.

`startup`

In a startup run (the default), all components are initialized using baseline states. These baseline states are set independently by each component and can include the use of restart files, initial files, external observed data files, or internal initialization (i.e., a "cold start"). In a startup run, the coupler sends the start date to the components at initialization. In addition, the coupler does not need an input data file. In a startup initialization, the ocean model does not start until the second ocean coupling step.

`branch`

In a branch run, all components are initialized using a consistent set of restart files from a previous run (determined by the $RUN_REFCASE and $RUN_REFDATE variables in env_run.xml). The case name is generally changed for a branch run, although it does not have to be. In a branch run, setting $RUN_STARTDATE is ignored because the model components obtain the start date from their restart datasets. Therefore, the start date cannot be changed for a branch run. This is the same mechanism that is used for performing a restart run (where $CONTINUE_RUN is set to TRUE in the env_run.xml file). Branch runs are typically used when sensitivity or parameter studies are required, or when settings for history file output streams need to be modified while still maintaining bit-for-bit reproducibility. Under this scenario, the new case is able to produce an exact bit-for-bit restart in the same manner as a continuation run if no source code or component namelist inputs are modified. All models use restart files to perform this type of run. $RUN_REFCASE and $RUN_REFDATE are required for branch runs. To set up a branch run, locate the restart tar file or restart directory for $RUN_REFCASE and $RUN_REFDATE from a previous run, then place those files in the $RUNDIR directory. See setting up a branch run.

`hybrid`

A hybrid run indicates that the model is initialized more like a startup, but uses initialization datasets from a previous case. This is somewhat analogous to a branch run with relaxed restart constraints. A hybrid run allows users to bring together combinations of initial/restart files from a previous case (specified by $RUN_REFCASE) at a given model output date (specified by $RUN_REFDATE). Unlike a branch run, the starting date of a hybrid run (specified by $RUN_STARTDATE) can be modified relative to the reference case. In a hybrid run, the model does not continue in a bit-for-bit fashion with respect to the reference case. The resulting climate, however, should be continuous provided that no model source code or namelists are changed in the hybrid run. In a hybrid initialization, the ocean model does not start until the second ocean coupling step, and the coupler does a "cold start" without a restart file.

The variable $RUN_TYPE determines the initialization type. This setting is only important for the initial run of a production run when the $CONTINUE_RUN variable is set to FALSE. After the initial run, the $CONTINUE_RUN variable is set to TRUE, and the model restarts exactly using input files in a case, date, and bit-for-bit continuous fashion.

The variable $RUN_STARTDATE is the start date (in yyyy-mm-dd format) for either a startup or hybrid run. If the run is targeted to be a hybrid or branch run, you must also specify values for $RUN_REFCASE and $RUN_REFDATE. All run startup variables are discussed in run start control variables.

A brief note on restarting runs; when you first begin a branch, hybrid or startup run, CONTINUE_RUN must be set to FALSE. After the job has successfully run and there are restart files, you will need to change CONTINUE_RUN to TRUE for the remainder of your run. Setting the RESUBMIT option to a value > 0 will cause the CONTINUE_RUN variable to be automatically set to TRUE upon completion of the initial run.

By default,

```
STOP_OPTION = ndays
STOP_N = 5
STOP_DATE = -999
```

The default setting is only appropriate for initial testing. Before a longer run is started, update the stop times based on the case throughput and batch queue limits. For example, if the model runs 5 model years/day, set RESUBMIT=30, STOP_OPTION= nyears, and STOP_N= 5. The model will then run in five year increments, and stop after 30 submissions.

Controlling output data

During a model run, each model component produces its own output datasets consisting of history, initial, restart, diagnostics, output log and rpointer files. Component history files and restart files are in netCDF format. Restart files are used to either exactly restart the model or to serve as initial conditions for other model cases. The rpointer files are text files listing the required component history and restart files required for restart.

Archiving is a phase of a model run where the generated output data is moved from `$RUNDIR` to a local disk area (short-term archiving). It has no impact on the production run except to clean up disk space in the `$RUNDIR` and help manage user quotas.

In `env_run.xml`, several variables control the behavior of short-term archiving. As an example for controlling the data output flow using the following example settings:

```
DOUT_S = TRUE
DOUT_S_ROOT = /$SCRATCH/$user/$CASE/archive
```

Several important points need to be made about short term archiving:

- By default, short-term archiving is enabled (`$DOUT_S = TRUE`)
- All output data is initially written to `$RUNDIR`.
- Unless a user explicitly turns off short-term archiving, files will be moved to `$DOUT_S_ROOT` at the end of a successful model run.
- Users should generally turn off short-term archiving when developing new code.

Standard output generated from each component is saved in a "log file" for each component in the `$RUNDIR`. Each time the model is run, a single coordinated datestamp is incorporated in the filenames of all output log files associated with that run. This common datestamp is generated by the run script and is of the form YYMMDD-hhmmss, where YYMMDD are the Year, Month, Day and hhmmss are the hour, minute and second that the run began (e.g. ocn.log.040526-082714). Log files are also copied to a user specified directory using the variable `$LOGDIR` in `env_run.xml`. The default is a 'logs' subdirectory in the `$CASEROOT` directory.

By default, each component also periodically writes history files (usually monthly) in netCDF format and also writes netCDF or binary restart files in the `$RUNDIR` directory. The history and log files are controlled independently by each component. History output control (i.e. output fields and frequency) is set in the `Buildconf/$component.buildnml.csh` files.

The raw history data does not lend itself well to easy time-series analysis. For example, CAM writes one or more large netCDF history file(s) at each requested output period. While this behavior is optimal for model execution, it makes it difficult to analyze time series of individual variables without having to access the entire data volume. Thus, the raw data from major model integrations is usually postprocessed into more user-friendly configurations, such as single files containing long time-series of each output fields, and made available to the community.

For CESM, please refer to the CESM2 Output Filename Conventions for a description of output data filenames.

Restarting a run

Restart files are written by each active component (and some data components) at intervals dictated by the driver via the setting of the `env_run.xml` variables, `$REST_OPTION` and `$REST_N`. Restart files allow the model to stop and then start again with bit-for-bit exact capability (i.e. the model output is exactly the same as if it had never been stopped). The driver coordinates the writing of restart files as well as the time evolution of the model. All components receive restart and stop information from the driver and write restarts or stop as specified by the driver.

It is important to note that runs that are initialized as branch or hybrid runs, will require restart/initial files from previous model runs (as specified by the variables, `$RUN_REFCASE` and `$RUN_REFDATE`). These required files must be prestaged by the user to the case `$RUNDIR` (normally `$EXEROOT/run`) before the model run starts. This is normally done by just copying the contents of the relevant `$RUN_REFCASE/rest/$RUN_REFDATE.00000` directory.

Whenever a component writes a restart file, it also writes a restart pointer file of the form, `rpointer.$component`. The restart pointer file contains the restart filename that was just written by the component. Upon a restart, each component reads its restart pointer file to determine the filename(s) to read in order to continue the model run. As examples, the following pointer files will be created for a component set using full active model components.

```
- rpointer.atm
- rpointer.drv
- rpointer.ice
- rpointer.lnd
- rpointer.rof
- rpointer.cism
- rpointer.ocn.ovf
- rpointer.ocn.restart
```

If short-term archiving is turned on, then the model archives the component restart datasets and pointer files into `$DOUT_S_ROOT/rest/yyyy-mm-dd-sssss`, where yyyy-mm-dd-sssss is the model date at the time of the restart (see below for more details).

Backing up to a previous restart

If a run encounters problems and crashes, you will normally have to back up to a previous restart. Assuming that short-term archiving is enabled, you will need to find the latest `$DOUT_S_ROOT/rest/yyyy-mm-dd-ssss/` directory that was created and copy the contents of that directory into your run directory (`$RUNDIR`). You can then continue the run and these restarts will be used. It is important to make sure the new rpointer.* files overwrite the rpointer.* files that were in `$RUNDIR`, or the job may not restart in the correct place.

Occasionally, when a run has problems restarting, it is because the rpointer files are out of sync with the restart files. The rpointer files are text files and can easily be edited to match the correct dates of the restart and history files. All the restart files should have the same date.

Archiving model output data

All component log files are copied to the directory specified by the `env_run.xml` variable `$LOGDIR` which by default is set to `$CASEROOT/logs`. This location is where log files are copied when the job completes successfully. If the job aborts, the log files will NOT be copied out of the `$RUNDIR` directory.

Once a model run has completed successfully, the output data flow will depend on whether or not short-term archiving is enabled (as set by the `env_run.xml` variable, `$DOUT_S`). By default, short-term archiving will be done (`DOUT_S=TRUE`).

No archiving

If no short-term archiving is performed, then all model output data will remain in the run directory, as specified by the `env_run.xml` variable, `$RUNDIR`.

Short-term archiving

If short-term archiving is enabled, the component output files will be moved to the short term archiving area on local disk, as specified by `$DOUT_S_ROOT`. The directory `DOUT_S_ROOT` is normally set to `$EXEROOT/../archive/$CASE.` and will contain the following directory structure:

```
rest/yyyy-mm-dd-sssss/
logs/
atm/hist/
cpl/hist
glc/hist
ice/hist
lnd/hist
ocn/hist
rof/hist
wav/hist
....
```

logs/ contains component log files created during the run. In addition to `$LOGDIR`, log files are also copied to the short-term archiving directory and therefore are available for long-term archiving.

rest/ contains a subset of directories that each contain a *consistent* set of restart files, initial files and rpointer files. Each sub-directory has a unique name corresponding to the model year, month, day and seconds into the day where

the files were created (e.g. 1852-01-01-00000/). The contents of any restart directory can be used to create a branch run or a hybrid run or back up to a previous restart date.

Long-term archiving

Users may choose to user their institution's preferred method for long-term archiving of model output. Previous releases of CESM provided an external long-term archiver tool which supported mass tape storage and HPSS systems. However, with the industry migration away from tape archives, it is no longer feasible for CIME to support all the possible archival schemes available.

Customizing a case

All CIME_compliant components generate their namelist settings using the `cime_config/buildnml` file located in the component's directory tree. As an example, the CIME data atmosphere model (DATM), generates namelists using the script `$CIMEROOT/components/data_comps/datm/cime_config/buildnml`.

User specific component namelist changes should only be made only by: - editing the `$CASEROOT/user_nl_xxx` files - using **xmlchange** to modify xml variables in `env_run.xml`, `env_build.xml` or `env_mach_pes.xml`

You can preview the component namelists by running **preview_namelists** from `$CASEROOT`. Calling **preview_namelists** results in the creation of component namelists (e.g. atm_in, lnd_in, .etc) in `$CASEROOT/CaseDocs/`. The namelist files created in the `CaseDocs/` are there only for user reference and SHOULD NOT BE EDITED since they are overwritten every time `preview_namelists` and `case.submit` are called.

The following sections are a summary of how to modify CIME, CESM and ACME components following represents a summary of controlling and modifying component-specific run-time settings:

Modifying an xml file

Modification of `$CASEROOT` xml variables the `$CASEROOT` script **xmlchange**, which performs variable error checking as part of changing values in the xml files.

To invoke **xmlchange**:

```
xmlchange <entry id>=<value>
-- OR --
xmlchange -id <entry id> -val <name> -file <filename>
          [-help] [-silent] [-verbose] [-warn] [-append] [-file]
```

`-id`	The xml variable name to be changed. (required)
`-val`	The intended value of the variable associated with the -id argument. (required) **Note**: If you want a single quotation mark ("", also called an apostrophe) to appear in the string provided by the -val option, you must specify it as "'".
`-file`	The xml file to be edited. (optional)
`-silent`	Turns on silent mode. Only fatal messages will be issued. (optional)
`-verbose`	Echoes all settings made by **create_newcase** and **case.setup**. (optional)
`-help`	Print usage info to STDOUT. (optional)

Customizing the PE layout

`env_mach_pes.xml` determines:

- the number of processors and OpenMP threads for each component
- the number of instances of each component
- the layout of the components across the hardware processors

Optimizing the throughput and efficiency of a CIME experiment often involves customizing the processor (PE) layout for (see load balancing). CIME provides significant flexibility with respect to the layout of components across different hardware processors. In general, the CIME components -- atm, lnd, ocn, ice, glc, rof, wav, and cpl -- can run on overlapping or mutually unique processors. Whereas each component is associated with a unique MPI communicator, the CIME driver runs on the union of all processors and controls the sequencing and hardware partitioning. The component processor layout is via three settings: the number of MPI tasks, the number of OpenMP threads per task, and the root MPI processor number from the global set.

The entries in `env_mach_pes.xml` have the following meanings:

NTASKS	the total number of MPI tasks, a negative value indicates nodes rather than tasks
NTHRDS	the number of OpenMP threads per MPI task
ROOTPE	the global mpi task of the component root task, if negative, indicates nodes rather than tasks
PSTRID	the stride of MPI tasks across the global set of pes (for now set to 1)
NINST	the number of component instances (will be spread evenly across NTASKS)

For example, if a component has `NTASKS=16`, `NTHRDS=4` and `ROOTPE=32`, then it will run on 64 hardware processors using 16 MPI tasks and 4 threads per task starting at global MPI task 32. Each CIME component has corresponding entries for `NTASKS`, `NTHRDS`, `ROOTPE` and `NINST` in `env_mach_pes.xml`.

There are some important things to note.

- NTASKS must be greater or equal to 1 even for inactive (stub) components.
- NTHRDS must be greater or equal to 1.
- If NTHRDS = 1, this generally means threading parallelization will be off for that component.
- NTHRDS should never be set to zero.
- The total number of hardware processors allocated to a component is NTASKS * NTHRDS.
- The coupler processor inputs specify the pes used by coupler computation such as mapping, merging, diagnostics, and flux calculation. This is distinct from the driver which always automatically runs on the union of all processors to manage model concurrency and sequencing.
- The root processor is set relative to the MPI global communicator, not the hardware processors counts. An example of this is below.
- The layout of components on processors has no impact on the science.
- If all components have identical NTASKS, NTHRDS, and ROOTPE set, all components will run sequentially on the same hardware processors.

The scientific sequencing is hardwired into the driver. Changing processor layouts does not change intrinsic coupling lags or coupling sequencing. ONE IMPORTANT POINT is that for a fully active configuration, the atmosphere component is hardwired in the driver to never run concurrently with the land or ice component. Performance improvements associated with processor layout concurrency is therefore constrained in this case such that there is never a performance reason not to overlap the atmosphere component with the land and ice components. Beyond that constraint, the land, ice, coupler and ocean models can run concurrently, and the ocean model can also run concurrently with the atmosphere model.

An important, but often misunderstood point, is that the root processor for any given component, is set relative to the MPI global communicator, not the hardware processor counts. For instance, in the following example:

```
NTASKS(ATM)=6  NTHRRDS(ATM)=4  ROOTPE(ATM)=0
NTASKS(OCN)=64 NTHRDS(OCN)=1   ROOTPE(OCN)=16
```

The atmosphere and ocean will run concurrently, each on 64 processors with the atmosphere running on MPI tasks 0-15 and the ocean running on MPI tasks 16-79. The first 16 tasks are each threaded 4 ways for the atmosphere. CIME ensures that the batch submission script ($CASE.run) automatically request 128 hardware processors, and

the first 16 MPI tasks will be laid out on the first 64 hardware processors with a stride of 4. The next 64 MPI tasks will be laid out on the second set of 64 hardware processors. If you had set ROOTPE_OCN=64 in this example, then a total of 176 processors would have been requested and the atmosphere would have been laid out on the first 64 hardware processors in 16x4 fashion, and the ocean model would have been laid out on hardware processors 113-176. Hardware processors 65-112 would have been allocated but completely idle.

Note: `env_mach_pes.xml` *cannot* be modified after "./case.setup" has been invoked without first invoking **case.setup -clean**.

Customizing driver namelists

Driver namelist variables belong in two groups - those that are set directly from `$CASEROOT` xml variables and those that are set by the driver utility ``$CIMEROOT/driver_cpl/cime_config/buildnml``. All driver namelist variables are defined in `$CIMEROOT/driver_cpl/cime_config/namelist_definition_drv.xml`. Those variables that can only be changed by modifying xml variables appear with the `entry` attribute `modify_via_xml="xml_variable_name"`. All other variables that appear `$CIMEROOT/driver_cpl/cime_config/namelist_definition_drv.xml` can be modified by adding a key-word value pair at the end of `user_nl_cpl`. For example, to change the driver namelist value of `eps_frac` to `1.0e-15`, you should add the following line to the end of the `user_nl_cpl`

```
eps_frac = 1.0e-15
```

To see the result of this modification to `user_nl_cpl` call `preview_namelists` and verify that this new value appears in `CaseDocs/drv_in`.

Customizing data model namelists and stream files

Data Atmosphere (DATM)

DATM is discussed in detail in data atmosphere overview. DATM can be user-customized in by either changing its namelist input or its stream files. The namelist file for DATM is `datm_in` (or `datm_in_NNN` for multiple instances).

- To modify `datm_in`, add the appropriate keyword/value pair(s) for the namelist changes you want at the end of the `$CASEROOT` file `user_nl_datm` (or `user_nl_datm_NNN` for multiple instances).
- To modify the contents of a DATM stream file, first use **preview_namelists** to obtain the contents of the stream txt files in `CaseDocs/`. Then:

  1. place a *copy* of this file in `$CASEROOT` with the string *"user_"* prepended
  2. **Make sure you change the permissions of the file to be writeable** (chmod 644)
  3. modify the `user_datm.streams.txt.*` file.

As an example, if the stream txt file in `CaseDocs/` is datm.streams.txt.CORE2_NYF.GISS, the modified copy in `$CASEROOT` should be `user_datm.streams.txt.CORE2_NYF.GISS`. After calling **preview_namelists** again, you should see your new modifications appear in `CaseDocs/datm.streams.txt.CORE2_NYF.GISS`.

Data Ocean (DOCN)

DOCN is discussed in detail in data ocean overview. DOCN can be user-customized in by either changing its namelist input or its stream files. The namelist file for DOCN is `docn_in` (or `docn_in_NNN` for multiple instances) and its values can be changed by editing the `$CASEROOT` file `user_nl_docn` (or `user_nl_docn_NNN` for multiple instances).

- To modify `docn_in`, add the appropriate keyword/value pair(s) for the namelist changes you want at the end of the `$CASEROOT` file `user_nl_docn` (or `user_nl_docn_NNN` for multiple instances).
- To modify the contents of a DOCN stream file, first use **preview_namelists** to obtain the contents of the stream txt files in `CaseDocs/`. Then:

  1. place a *copy* of this file in `$CASEROOT` with the string *"user_"* prepended
  2. **Make sure you change the permissions of the file to be writeable** (chmod 644)
  3. modify the `user_docn.streams.txt.*` file.

As an example, if the stream text file in `CaseDocs/` is `docn.stream.txt.prescribed`, the modified copy in `$CASEROOT` should be `user_docn.streams.txt.prescribed`. After changing this file and calling **preview_namelists** again, you should see your new modifications appear in `CaseDocs/docn.streams.txt.prescribed`.

Data Sea-ice (DICE)

DICE is discussed in detail in data sea-ice overview. DICE can be user-customized in by either changing its namelist input or its stream files. The namelist file for DICE is `dice_in` (or `dice_in_NNN` for multiple instances) and its values can be changed by editing the `$CASEROOT` file `user_nl_dice` (or `user_nl_dice_NNN` for multiple instances).

- To modify `dice_in`, add the appropriate keyword/value pair(s) for the namelist changes you want at the end of the `$CASEROOT` file `user_nl_dice` (or `user_nl_dice_NNN` for multiple instances).
- To modify the contents of a DICE stream file, first use **preview_namelists** to obtain the contents of the stream txt files in `CaseDocs/`. Then:
  1. place a *copy* of this file in `$CASEROOT` with the string *"user_"* prepended
  2. **Make sure you change the permissions of the file to be writeable** (chmod 644)
  3. modify the `user_dice.streams.txt.*` file.

Data Land (DLND)

DLND is discussed in detail in data land overview. DLND can be user-customized in by either changing its namelist input or its stream files. The namelist file for DLND is `dlnd_in` (or `dlnd_in_NNN` for multiple instances) and its values can be changed by editing the `$CASEROOT` file `user_nl_dlnd` (or `user_nl_dlnd_NNN` for multiple instances).

- To modify `dlnd_in`, add the appropriate keyword/value pair(s) for the namelist changes you want at the end of the `$CASEROOT` file `user_nl_dlnd` (or `user_nl_dlnd_NNN` for multiple instances).
- To modify the contents of a DLND stream file, first use **preview_namelists** to obtain the contents of the stream txt files in `CaseDocs/`. Then:
  1. place a *copy* of this file in `$CASEROOT` with the string *"user_"* prepended
  2. **Make sure you change the permissions of the file to be writeable** (chmod 644)
  3. modify the `user_dlnd.streams.txt.*` file.

Data River (DROF)

DROF is discussed in detail in data river overview. DROF can be user-customized in by either changing its namelist input or its stream files. The namelist file for DROF is `drof_in` (or `drof_in_NNN` for multiple instances) and its values can be changed by editing the `$CASEROOT` file `user_nl_drof` (or `user_nl_drof_NNN` for multiple instances).

- To modify `drof_in`, add the appropriate keyword/value pair(s) for the namelist changes you want at the end of the `$CASEROOT` file `user_nl_drof` (or `user_nl_drof_NNN` for multiple instances).
- To modify the contents of a DROF stream file, first use **preview_namelists** to obtain the contents of the stream txt files in `CaseDocs/`. Then:
  1. place a *copy* of this file in `$CASEROOT` with the string *"user_"* prepended
  2. **Make sure you change the permissions of the file to be writeable** (chmod 644)
  3. modify the `user_drof.streams.txt.*` file.

Customizing CESM active component-specific namelist settings

CAM

CAM's configure and build-namelist utilities are called by `Buildconf/cam.buildnml.csh`. CAM_CONFIG_OPTS, CAM_NAMELIST_OPTS and CAM_NML_USECASE are used to set compset variables (e.g., "-phys cam5" for

CAM_CONFIG_OPTS) and in general should not be modified for supported compsets. For a complete documentation of namelist settings, see CAM namelist variables. To modify CAM namelist settings, you should add the appropriate keyword/value pair at the end of the `$CASEROOT/user_nl_cam` file (see the documentation for each file at the top of that file). For example, to change the solar constant to 1363.27, modify the `user_nl_cam` file to contain the following line at the end "solar_const=1363.27". To see the result of adding this, call **preview_namelists** and verify that this new value appears in `CaseDocs/atm_in`.

CLM

CIME generates the CLM namelist variables by calling `$SRCROOT/components/clm/cime_config/buildnml`. CLM-specific CIME xml variables are set in `$SRCROOT/components/clm/cime_config/config_component.xml` and are used by CLM's `buildnml` script to generate the namelist. For a complete documentation of namelist settings, see CLM namelist variables. To modify CLM namelist settings, you should add the appropriate keyword/value pair at the end of the `$CASEROOT/user_nl_clm` file To see the result of your change, call **preview_namelists** and verify that the changes appear correctly in `CaseDocs/lnd_in`.

RTM

CIME generates the RTM namelist variables by calling `$SRCROOT/components/rtm/cime_config/buildnml`. For a complete documentation of namelist settings, see RTM namelist variables. To modify RTM namelist settings you should add the appropriate keyword/value pair at the end of the `$CASEROOT/user_nl_rtm` file. To see the result of your change, call **preview_namelists** and verify that the changes appear correctly in `CaseDocs/rof_in`.

--- CICE ---

CICE's configure and build-namelist utilities are now called by `Buildconf/cice.buildnml.csh`. Note that CICE_CONFIG_OPTS, and CICE_NAMELIST_OPTS are used to set compset specific variables and in general should not be modified for supported compsets. For a complete documentation of namelist settings, see CICE namelist variables. To modify CICE namelist settings, you should add the appropriate keyword/value pair at the end of the `$CASEROOT/user_nl_cice` file (see the documentation for each file at the top of that file). To see the result of your change, call **preview_namelists** and verify that the changes appear correctly in `CaseDocs/ice_in`.

In addition, **case.setup** creates CICE's compile time block decomposition variables in `env_build.xml` as follows:

```
./case.setup
  ⇓
Buildconf/cice.buildnml.csh and $NTASKS_ICE and $NTHRDS_ICE
  ⇓
env_build.xml variables CICE_BLCKX, CICE_BLCKY, CICE_MXBLCKS, CICE_DECOMPTYPE
CPP variables in cice.buildexe.csh
```

POP2

See POP2 namelist variables for a complete description of the POP2 run-time namelist variables. Note that OCN_COUPLING, OCN_ICE_FORCING, OCN_TRANSIENT are normally utilized ONLY to set compset specific variables and should not be edited. For a complete documentation of namelist settings, see CICE namelist variables. To modify POP2 namelist settings, you should add the appropriate keyword/value pair at the end of the `$CASEROOT/user_nl_pop2` file (see the documentation for each file at the top of that file). To see the result of your change, call **preview_namelists** and verify that the changes appear correctly in `CaseDocs/ocn_in`.

In addition, **cesm_setup** also generates POP2's compile time compile time block decomposition variables in `env_build.xml` as follows:

```
./cesm_setup
    ⇓
Buildconf/pop2.buildnml.csh and $NTASKS_OCN and $NTHRDS_OCN
    ⇓
env_build.xml variables POP2_BLCKX, POP2_BLCKY, POP2_MXBLCKS, POP2_DECOMPTYPE
CPP variables in pop2.buildexe.csh
```

CISM

See CISM namelist variables for a complete description of the CISM run-time namelist variables. This includes variables that appear both in `cism_in` and in `cism.config`. To modify any of these settings, you should add the appropriate keyword/value pair at the end of the `user_nl_cism` file (see the documentation for each file at the top of that file). To see the result of your change, call **preview_namelists** and verify that the changes appear correctly in `CaseDocs/cism_in` and `CaseDocs/cism.config`.

There are also some run-time settings set via `env_run.xml`, as documented in CISM run time variables - in particular, the model resolution, set via `CISM_GRID`. The value of `CISM_GRID` determines the default value of a number of other namelist parameters.

Customizing ACME active component-specific namelist settings

Cloning a Case

If you have access to the run you want to clone, the **create_clone** command will create a new case while also preserving local modifications to the case that you want to clone. You can run the utility **create_clone** from the cime/scripts directory. It has the following arguments:

`--case`	The name or path of the new case.
`--clone`	The full pathname of the case to be cloned.
`--cime-output-root`	The root path below which the case run and bld directories will be created. You should use this option if you are cloning a case owned by another user.
`--keepexe`	Sets **$EXEROOT** to point to the original case.
`--project`	Specify a project code for the new case.
`--mach-dir`	Specify an alternate location of the machines directory.
`--silent`	Enables silent mode. Only fatal messages will be issued.
`--verbose`	Echoes all settings.
`--debug`	Print very verbose debug information to the file **create_clone.log**
`--help`	Prints usage instructions.

Here is the simplest example of using **create_clone**:

```
> cd $CIMEROOT/scripts
> create_clone --case $CASEROOT --clone $CLONEROOT
```

create_clone will preserve any local namelist modifications made in the user_nl_xxxx files as well as any source code modifications in the SourceMods tree.

Important:: Do not change anything in the `env_case.xml` file. The `$CASEROOT/` directory will now appear as if **create_newcase** had just been run -- with the exception that local modifications to the env_* files are preserved.

Another approach to duplicating a case is to use the information in that case's `README.case` and `CaseStatus` files to create a new case and duplicate the relevant `xmlchange` commands that were issued in the original case. Note that this alternative will *not* preserve any local modifications that were made to the original case, such as source-code or build-script modifications; you will need to import those changes manually.

Troubleshooting

Troubleshooting case creation

Generally, **create_newcase** errors are reported to the terminal and should provide some guidance about what caused the error.

If **create_newcase** fails on a relatively generic error, first check carefully that the command line arguments match the interfaces specification. Type:

```
> create_newcase --help
```

and review usage.

Troubleshooting job submission

Most of the problems associated with submission or launch are very site specific. The batch and run aspects of the `case.submit` script is created by parsing the xml variables in `$CASEROOT/env_batch.xml`.

First, review the batch submission options being used. These are found in `$CASEROOT/env_batch.xml`. Confirm that the options are consistent with the site specific batch environment, and that the queue names, time limits, and hardware processor request makes sense and is consistent with the case running.

Second, make sure that `case.submit` submits the script `case.run` with the correct batch job submission tool, whether that's qsub, bsub, or something else, and for instance, whether a redirection "<" character is required or not. The information for how `case.submit` submit jobs appears at the end of the standard output stream.

Troubleshooting run-time problems

To check that a run completed successfully, check the last several lines of the cpl.log file for the string `SUCCESSFUL TERMINATION OF CPL7-CCSM`. A successful job also usually copies the log files to the directory $CASEROOT/logs.

Note: The first things to check if a job fails are: - whether the model timed out - whether a disk quota limit was hit - whether a machine went down, - whether a file system became full.

If any of those things happened, take appropriate corrective action and resubmit the job.

If it is not clear any of the above caused a case to fail, then there are several places to look for error messages.

- Check component log files in `$RUNDIR`. This directory is set in the `env_run.xml` and is where the model is run. Each component writes its own log file, and there should be log files there for every component (i.e. of the form cpl.log.yymmdd-hhmmss). Check log file for each component for an error message, especially at the end or near the end of each file.
- Check for a standard out and/or standard error file in `$CASEROOT`. The standard out/err file often captures a significant amount of extra model output and also often contains significant system output when the job terminates. Sometimes, a useful error message can be found well above the bottom of a large standard out/err file. Backtrack from the bottom in search of an error message.
- Check for core files in $RUNDIR and review them using an appropriate tool.
- Check any automated email from the job about why a job failed. This is sent by the batch scheduler and is a site specific feature that may or may not exist.
- Check the archive directory. If a case failed, the log files or data may still have been archived. The archiver is turned on if DOUT_S is set to TRUE in env_run.xml. The archive directory is set by the env variable DOUT_S_ROOT and the directory to check is $DOUT_S_ROOT/$CASE.

A common error is for the job to time out which often produces minimal error messages. By reviewing the daily model date stamps in the cpl.log file and the time stamps of files in the $RUNDIR directory, there should be enough information to deduce the start and stop time of a run. If the model was running fine, but the batch wall limit was reached, either reduce the run length or increase the batch time limit request. If the model hangs and then times out, that's usually indicative of either a system problem (an MPI or file system problem) or possibly a model problem. If a system problem is suspected, try to resubmit the job to see if an intermittent problem occurred. Also send help to local site consultants to provide them with feedback about system problems and to get help.

Another error that can cause a timeout is a slow or intermittently slow node. The cpl.log file normally outputs the time used for every model simulation day. To review that data, grep the cpl.log file for the string, tStamp

```
> grep tStamp cpl.log.* | more
```

which gives output that looks like this::

```
tStamp_write: model date = 10120 0 wall clock = 2009-09-28 09:10:46 avg dt = 58.58 dt = 58.1
tStamp_write: model date = 10121 0 wall clock = 2009-09-28 09:12:32 avg dt = 60.10 dt = 105.
```

Review the run times for each model day. These are indicated at the end of each line. The "avg dt = " is the running average time to simulate a model day in the current run and "dt = " is the time needed to simulate the latest model day. The model date is printed in YYYYMMDD format and the wall clock is the local date and time. So in this case 10120 is Jan 20, 0001, and it took 58 seconds to run that day. The next day, Jan 21, took 105.90 seconds. If a wide variation in the simulation time is observed for typical mid-month model days, then that is suggestive of a system problem. However, be aware that there are variations in the cost of the model over time. For instance, on the last

day of every simulated month, the model will typically write netcdf files, and this can be a significant intermittent cost. Also, some model configurations read data mid- month or run physics intermittently at a timestep longer than one day. In those cases, some run time variability would be observed and it would be caused by CESM1, not system variability. With system performance variability, the time variation is typically quite erratic and unpredictable.

Sometimes when a job times out, or it overflows disk space, the restart files will get mangled. With the exception of the CAM and CLM history files, all the restart files have consistent sizes. Compare the restart files against the sizes of a previous restart. If they don't match, then remove them and move the previous restart into place before resubmitting the job. Please see restarting a run.

On HPC systems, it is not completely uncommon for nodes to fail or for access to large file systems to hang. Please make sure a case fails consistently in the same place before filing a bug report.

CIME User's Guide Part 2: CIME internals, Porting, Testing and Use Cases

CIME internals

The file `$CIMEROOT/config/[cesm,acme]/config_files.xml` contains all model specfic information that CIME utilizes to determine compsets, compset component settings, model grids, machines, batch queue settings, and compiler settings. `config_files.xml` contains the following xml nodes that are discussed below.

```
compset definitions:
   <entry id="COMPSETS_SPEC_FILE">

component specfic compset settings:
   <entry id="CONFIG_CPL_FILE">
   <entry id="CONFIG_CPL_FILE_MODEL_SPECIFIC">
   <entry id="CONFIG_ATM_FILE">
   <entry id="CONFIG_LND_FILE">
   <entry id="CONFIG_ROF_FILE">
   <entry id="CONFIG_ICE_FILE">
   <entry id="CONFIG_OCN_FILE">
   <entry id="CONFIG_GLC_FILE">
   <entry id="CONFIG_WAV_FILE">
   <entry id="CONFIG_ESP_FILE">

pe-settings:
   <entry id="PES_SPEC_FILE">

grid definitions:
   <entry id="GRIDS_SPEC_FILE">

machine specfic definitions:
   <entry id="MACHINES_SPEC_FILE">
   <entry id="BATCH_SPEC_FILE">
   <entry id="COMPILERS_SPEC_FILE">
   <entry id="PIO_SPEC_FILE">

testing:
   <entry id="CONFIG_TESTS_FILE">
   <entry id="TESTS_SPEC_FILE">
   <entry id="TESTS_MODS_DIR">
   <entry id="SYSTEM_TESTS_DIR">

archiving:
   <entry id="LTARCHIVE_SPEC_FILE">

CIME components namelists definitions:
   <entry id="NAMELIST_DEFINITION_FILE">
```

```
user-mods directories:
   <entry id="USER_MODS_DIR">
```

Where are compsets defined?

CIME looks at the xml element `COMPSETS_SPEC_FILE` in the file `$CIMEROOT/config/[cesm,acme]/config_files.xml` to determine where to find the list of supported compsets lists all model components that set compsets.

In the case of CESM, this xml element has the following contents

```
<entry id="COMPSETS_SPEC_FILE">
  <type>char</type>
  <default_value>unset</default_value>
  <values>
    <value component="allactive">$SRCROOT/cime_config/config_compsets.xml</value>
    <value component="drv"      >$CIMEROOT/src/drivers/mct/cime_config/config_compsets.xml</
    <value component="cam"      >$SRCROOT/components/cam/cime_config/config_compsets.xml</va
    <value component="cism"     >$SRCROOT/components/cism/cime_config/config_compsets.xml</v
    <value component="clm"      >$SRCROOT/components/clm/cime_config/config_compsets.xml</va
    <value component="cice"     >$SRCROOT/components/cice/cime_config/config_compsets.xml</v
    <value component="pop"      >$SRCROOT/components/pop/cime_config/config_compsets.xml</va
  </values>
  <group>case_last</group>
  <file>env_case.xml</file>
  <desc>file containing specification of all compsets for primary component (for documentati
  <schema>$CIMEROOT/config/xml_schemas/config_compsets.xsd</schema>
</entry>
```

where `$SRCROOT` is the root of your CESM sandbox and contains `$CIMEROOT` as a sub-directory.

> ***Note***
>
> CIME searches each of the above `<config_compsets.xml>` files in the `<value>` nodes to determine if there is a match for the compset longname. If a match is found then the value of the `component` attribute is set for all other searches that appear below.

Where are component-specific settings defined for the target compset?

Every model component contains a `config_component.xml` file that has two functions:

1. Specifying the component-specfici definitions of what can appear after the `%` in the compset longname, (e.g. `DOM` in `DOCN%DOM`).
2. Specifying the compset-specific `$CASEROOT` xml variables.

CIME first parses the following nodes to determine the `config_component.xml` files for the driver. There are two such files, one is model independent

```
<entry id="CONFIG_CPL_FILE">
   ...
   <default_value>$CIMEROOT/driver_cpl/cime_config/config_component.xml</default_value>
   ..
</entry>
```

and the other is model specific

```
<entry id="CONFIG_CPL_FILE_MODEL_SPECIFIC">
   <default_value>$CIMEROOT/driver_cpl/cime_config/config_component_$MODEL.xml</default_valu
</entry>
```

CIME the parses each of the following nodes

```
<entry id="CONFIG_ATM_FILE">
<entry id="CONFIG_ESP_FILE">
<entry id="CONFIG_ICE_FILE">
<entry id="CONFIG_GLC_FILE">
<entry id="CONFIG_LND_FILE">
<entry id="CONFIG_OCN_FILE">
<entry id="CONFIG_ROF_FILE">
<entry id="CONFIG_WAV_FILE">
```

using the value of the `component` attribute determined to determine the list of `config_component.xml` files to use for the requested compset longname.

As an example, the possible atmosphere components for CESM have the following associated `config_component.xml` files.

```
<entry id="CONFIG_ATM_FILE">
  <type>char</type>
  <default_value>unset</default_value>
  <values>
    <value component="cam" >$SRCROOT/components/cam/cime_config/config_component.xml</value>
    <value component="datm">$CIMEROOT/components/data_comps/datm/cime_config/config_componen
    <value component="satm">$CIMEROOT/components/stub_comps/satm/cime_config/config_componen
    <value component="xatm">$CIMEROOT/components/xcpl_comps/xatm/cime_config/config_componen
  </values>
  <group>case_last</group>
  <file>env_case.xml</file>
  <desc>file containing specification of component specific definitions and values(for docum
  <schema>$CIMEROOT/cime_config/xml_schemas/entry_id.xsd</schema>
</entry>
```

If the compset's atm `component` attribute is `datm` then the file `$CIMEROOT/components/data_comps/datm/cime_config/config_component.xml` will specify all possible component settings for `DATM`.

The schema for every `config_component.xml` file has a `<description>` node that specfies all possible values that can follow the % character in the compset name.

As an example, use the command **manage_case --query-component-name cam** to query the possibly % modifiers for CESM cam.

Where are pe-settings defined for the target compset and model grid?

CIME looks at the xml element `PES_SPEC_FILE` in the file `$CIMEROOT/config/[cesm,acme]/config_files.xml` to determine where to find the supported out-of-the-box model grids for the target `component`. Currently, this node has the following contents (where `MODEL` can be `[acme, cesm]`.

Each `component` that sets compsets, must also have an associated `config_pes.xml` file that specifies an out-of-the-box pe-layout for those compsets. In addition, this out-of-the-box pe-layout might also have dependencies on the model grid and the target machine. Finally, there might be more than one-of-the-box pe-layout that could be used for a compset/grid/machine combination: one for a low processor setting and one for a high processor setting.

A typical entry in `config_pes.xml` looks like the following:

```
<grid name="a%T62">
  <mach name="yellowstone|pronghorn">
    <pes pesize="any" compset="DATM%IAF">
    .......
    </pes>
  </mach>
</grid>
```

Given these various dependencies, an order of precedence has been established to determine the optimal match. This order is as follows:

1. grid match

   CIME first searches the grid nodes and tries to find a grid match in `config_grids.xml`. The search is based on a regular expression match for the grid longname. All nodes that have a grid match are then used in the subsequent search. If there is no grid match, than all nodes that have

   ```
   <grid name="any">
   ```

   are used in the subsequent search.

2. machine match

   CIME next uses the list of nodes obtained in (1.) to match on the machine name using the `<mach>` nodes. If there is no machine match, then all nodes in (1.) that have

   ```
   <machine name="any">
   ```

   are used in the subsequent search.

3. pesize and compset match

   CIME next uses the list of nodes obtained in (2.) to match on pesize and compset using the `<pes>` nodes. If there is no match, then the node in (2.) that has

   ```
   <pes pesize="any" compset="any">
   ```

   will be the one used.

create_newcase outputs the matches that are found in determining the best out-of-the-box peylayout.

Where are model grids defined?

CIME looks at the xml node `GRIDS_SPEC_FILE` in the file `$CIMEROOT/config/[cesm,acme]/config_files.xml` to determine where to find the supported out-of-the-box model grids for the target model. Currently, this node has the following contents (where `MODEL` can be `[acme, cesm]`.

```
<entry id="GRIDS_SPEC_FILE">
  <type>char</type>
  <default_value>$CIMEROOT/cime_config/$MODEL/config_grids.xml</default_value>
  <group>case_last</group>
  <file>env_case.xml</file>
  <desc>file containing specification of all supported model grids, domains and mapping file
  <schema>$CIMEROOT/cime_config/xml_schemas/config_grids_v2.xsd</schema>
</entry>
```

Where are machines defined?

CIME looks at the xml node `MACHINE_SPEC_FILE` in the file `$CIMEROOT/config/[cesm,acme]/config_files.xml` to determine where to find the supported out-of-the-box machines for the target model. Currently, this node has the following contents (where `MODEL` can be `[acme, cesm]`.

```
<entry id="MACHINES_SPEC_FILE">
  <type>char</type>
  <default_value>$CIMEROOT/cime_config/$MODEL/machines/config_machines.xml</default_value>
  <group>case_last</group>
  <file>env_case.xml</file>
  <desc>file containing machine specifications for target model primary component (for docum
  <schema>$CIMEROOT/cime_config/xml_schemas/config_machines.xsd</schema>
</entry>
```

As part of porting, you will need to customize the config_machines.xml file.

Where are batch system settings defined?

CIME looks at the xml node BATCH_SPEC_FILE in the file $CIMEROOT/config/[cesm,acme]/config_files.xml to determine where to find the supported out-of-the-box batch system details for the target model. Currently, this node has the following contents (where MODEL can be [acme, cesm].

```
<entry id="BATCH_SPEC_FILE">
  <type>char</type>
  <default_value>$CIMEROOT/cime_config/$MODEL/machines/config_batch.xml</default_value>
  <group>case_last</group>
  <file>env_case.xml</file>
  <desc>file containing batch system details for target system  (for documentation only - DO
  <schema>$CIMEROOT/cime_config/xml_schemas/config_batch.xsd</schema>
</entry>
```

As part of porting, you will need to customize the config_batch.xml file.

Where are compiler settings defined?

CIME looks at the xml element COMPILERS_SPEC_FILE in the file $CIMEROOT/config/[cesm,acme]/config_files.xml to determine where to find the supported out-of-the-box batch system details for the target model. Currently, this node has the following contents (where MODEL can be [acme, cesm].

```
<entry id="COMPILERS_SPEC_FILE">
  <type>char</type>
  <default_value>$CIMEROOT/cime_config/$MODEL/machines/config_compilers.xml</default_value>
  <group>case_last</group>
  <file>env_case.xml</file>
  <desc>file containing compiler specifications for target model primary component (for docu
  <schema>$CIMEROOT/cime_config/xml_schemas/config_compilers_v2.xsd</schema>
</entry>
```

As part of porting, you will need to customize the config_batch.xml file.

Porting and Validating CIME on a new Platform

Porting Overview

One of the first steps many users will have to address is getting CIME based models running on their local machine. This section will describe that process. It is usually very helpful to assure that you can run a basic mpi parallel program on your machine prior to attempting a CIME port. Understanding how to compile and run the program fhello_world_mpi.F90 shown here could potentially save many hours of frustration.

```
program fhello_world_mpi.F90
  use mpi
  implicit none
  integer ( kind = 4 ) error
  integer ( kind = 4 ) id
  integer p
  character(len=MPI_MAX_PROCESSOR_NAME) :: name
  integer clen
  integer, allocatable :: mype(:)
  real ( kind = 8 ) wtime

  call MPI_Init ( error )
  call MPI_Comm_size ( MPI_COMM_WORLD, p, error )
  call MPI_Comm_rank ( MPI_COMM_WORLD, id, error )
  if ( id == 0 ) then
     wtime = MPI_Wtime ( )

     write ( *, '(a)' ) ' '
     write ( *, '(a)' ) 'HELLO_MPI - Master process:'
```

```
    write ( *, '(a)' ) '  FORTRAN90/MPI version'
    write ( *, '(a)' ) ' '
    write ( *, '(a)' ) '  An MPI test program.'
    write ( *, '(a)' ) ' '
    write ( *, '(a,i8)' ) '  The number of processes is ', p
    write ( *, '(a)' ) ' '
  end if
  call MPI_GET_PROCESSOR_NAME(NAME, CLEN, ERROR)
  write ( *, '(a)' ) ' '
  write ( *, '(a,i8,a,a)' ) '  Process ', id, ' says "Hello, world!" ',name(1:clen)

  call MPI_Finalize ( error )
end program
```

Once you are assured that you have a basic functional MPI environment you will need to provide a few prerequisite tools for building and running CIME.

- A python interpreter version 2.7 or newer
- Build tools gmake and cmake
- A netcdf library version 4.3 or newer built with the same compiler you will use for CIME
- Optionally a pnetcdf library.

The following steps should be followed:

1. Create a `$HOME/.cime` directory
2. Copy the template file `$CIME/config/xml_schemas/config_machines_template.xml` to `$HOME/.cime/config_machines.xml`
3. Fill in the contents of `$HOME/.cime/config_machines.xml` that specific to your machine. This file contains all the information that a user needs to set to configure a new machine to be CIME complaint. For more details see customize the config_machines.xml file. The completed file should conform to the schema definition provided, check it using:

   ```
   xmllint --noout --schema $CIME/config/xml_schemas/config_machines.xsd $HOME/.cime/config
   ```

4. The files `config_batch.xml` and `config_compilers.xml` may also need specific adjustments for your batch system and compiler. You can edit these files in place to add your machine configuration or you can place your custom configuration files in the directory `$HOME/.cime/`. We recommend the latter approach. All files in `$HOME/.cime/` are appended to the xml objects read into memory.
5. Once you have a basic configuration for your machine defined in your new `$HOME/.cime` XML files, you should try the `scripts_regression_test` in directory `$CIME/utils/python/tests`. This script will run a number of basic unit tests starting from the simplest issues and working toward more complicated ones.
6. Finally when all the previous steps have run correctly, you are ready to try a case at your target compset and resolution. Once you have successfully created the required xml files in your .cime directory and are satisfied with the results you can merge them into the default files in the `config/$CIME_MODEL/machines` directory. If you would like to make this machine definition available generally you may then issue a pull request to add your changes to the git repository.

Enabling out-of-the-box capability for your machine

Customizing the machines file for your machine

Machine specific files are defined in the model-specific config_machines.xml.

The first step a user must take to make their machine CIME-compatible is to add the appropriate entries for their machine in `config_machines.xml`.

Each `<machine>` tag requires the following input:

- `DESC`: a text description of the machine, this field is current not used

- NODENAME_REGEX: a regular expression used to identify this machine it must work on compute nodes as well as login nodes, use machine option to create_test or create_newcase if this flag is not available
- OS: the operating system of this machine.
- PROXY: optional http proxy for access to the internet
- COMPILERS: compilers supported on this machine, comma seperated list, first is default
- MPILIBS: mpilibs supported on this machine, comma seperated list, first is default
- PROJECT: A project or account number used for batch jobs can be overridden in environment or $HOME/.cime/config
- SAVE_TIMING_DIR: (Acme only) directory to write timing output to
- CIME_OUTPUT_ROOT: Base directory for case output, the bld and run directories are written below here
- DIN_LOC_ROOT: location of the inputdata directory
- DIN_LOC_ROOT_CLMFORC: optional input location for clm forcing data
- DOUT_S_ROOT: root directory of short term archive files
- DOUT_L_MSROOT: root directory on mass store system of long term archive files
- BASELINE_ROOT: Root directory for system test baseline files
- CCSM_CPRNC: location of the cprnc tool, compares model output in testing
- GMAKE: gnu compatible make tool, default is 'gmake'
- GMAKE_J: optional number of threads to pass to the gmake flag
- TESTS: (acme only) list of tests to run on this machine
- BATCH_SYSTEM: batch system used on this machine (none is okay)
- SUPPORTED_BY: contact information for support for this system
- MAX_TASKS_PER_NODE: maximum number of threads*tasks per shared memory node on this machine
- PES_PER_NODE: number of physical PES per shared node on this machine, in practice the MPI tasks per node will not exceed this value
- PROJECT_REQUIRED: Does this machine require a project to be specified to the batch system?
- mpirun: The mpi exec to start a job on this machine. This is itself an element that has sub elements that must be filled:
  - Must have a required <executable> element
  - May have optional attributes of compiler, mpilib and/or threaded
  - May have an optional <arguments> element which in turn contain one or more <arg> elements. These specify the arguments to the mpi executable and as a result are dependent on your mpi library implementation.
- module_system: How and what modules to load on this system. Module systems allow you to easily load multiple compiler environments on a given machine. CIME provides support for two types of module tools: module and soft. If neither of these are available on your machine, the simply set <module_system type="none"\>.
- environment_variables**: environment_variables to set on this system.**

  This contains sub elements, <env> with the name attribute specifying the environment variable name, and the element value specifying the corresponding environment variable value. If the element value is not set, then the corresponding environment variable will be unset in your shell.

  As an example, the following sets the environment variable OMP_STACKSIZE to 256M.

  ```
  <env name="OMP_STACKSIZE">256M</env>
  ```

  and the following unsets this environment variable in the shell:

```
<env name="OMP_STACKSIZE"></env>
```

> ***Note***
>
> These changes are **ONLY** activated for the CIME build and run environment, **BUT NOT** for your login shell. To activate them for your login shell, you would source either `$CASEROOT/.env_mach_specific.sh` or `$CASEROOT/.env_mach_specific.csh`, depending on your shell.

Customizing the batch directives for your machine

The **config_batch.xml** schema is defined in `$CIMEROOT/config/xml_schemas/config_batch.xml` config batch currently supports batch systems: pbs, cobalt, lsf and slurm. General configurations for each system are provided at the top of the file and specific modifications for a given machine are provided below. In particular each machine should define its own queues. Following machine specific queue details is a batch_jobs section, this section of the file describes each job that will be submitted to the queue for a CIME workflow, the template file that will be used to generate that job and the prerequisets that must be met before the job is submitted and dependancies that must be satisfied before that job is run. By default the CIME workflow consists of three jobs (case.run, case.st_archive, and case.lt_archive) there is also a case.test file that is used by the CIME system test workflow.

Customize the compiler options for your machine

> ***Todo***
>
> Jim Edwards add the contents of this section

Validating your port

The following port validation is recommended for any new machine. Carrying out these steps does not guarantee the model is running properly in all cases nor that the model is scientifically valid on the new machine. In addition to these tests, detailed validation should be carried out for any new production run. That means verifying that model restarts are bit-for-bit identical with a baseline run, that the model is bit-for-bit reproducible when identical cases are run for several months, and that production cases are monitored very carefully as they integrate forward to identify any potential problems as early as possible. These are recommended steps for validating a port and are largely functional tests. Users are responsible for their own validation process, especially with respect to science validation.

1. Verify functionality by performing these functionality tests.

```
ERS_D.f19_g16.X
ERS_D.T31_g37.A
ERS_D.f19_g16.B1850CN
ERI.ne30_g16.X
ERI.T31_g37.A
ERI.f19_g16.B1850CN
ERS.ne30_ne30.F
ERS.f19_g16.I
ERS.T62_g16.C
ERS.T62_g16.DTEST
ERT.ne30_g16.B1850CN
```

2. Verify performance and scaling analysis.

    a. Create one or two load-balanced configurations to check into `Machines/config_pes.xml` for the new machine.

    b. Verify that performance and scaling are reasonable.

c. Review timing summaries in $CASEROOT for load balance and throughput.

d. Review coupler "daily" timing output for timing inconsistencies. As has been mentioned in the section on load balancing a case, useful timing information is contained in cpl.log.$date file that is produced for every run. The cpl.log file contains the run time for each model day during the model run. This diagnostic is output as the model runs. You can search for tStamp in this file to see this information. This timing information is useful for tracking down temporal variability in model cost either due to inherent model variability cost (I/O, spin-up, seasonal, etc) or possibly due to variability due to hardware. The model daily cost is generally pretty constant unless I/O is written intermittently such as at the end of the month.

3. Perform validation (both functional and scientific):

   a. Perform a new CIME validation test (**TODO: fill this in**)

   b. Follow the CCSM4.0 CICE port-validation procedure.

   c. Follow the CCSM4.0 POP2 port-validation procedure.

4. Perform two, one-year runs (using the expected load-balanced configuration) as separate job submissions and verify that atmosphere history files are bfb for the last month. Do this after some performance testing is complete; you may also combine this with the production test by running the first year as a single run and the second year as a multi-submission production run. This will test reproducibility, exact restart over the one-year timescale, and production capability all in one test.

5. Carry out a 20-30 year 1.9x2.5_gx1v6 resolution, B_1850_CN compset simulation and compare the results with the diagnostics plots for the 1.9x2.5_gx1v6 Pre-Industrial Control (see the CCSM4.0 diagnostics). Model output data for these runs will be available on the Earth System Grid (ESG) as well.

Optimizing Processor Layout

Load balancing refers to the optimization of the processor layout for a given model configuration (compset, grid, etc) such that the cost and throughput will be optimal. Optimal is a somewhat subjective thing. For a fixed total number of processors, it means achieving the maximum throughput. For a given configuration across varied processor counts, it means finding several "sweet spots" where the model is minimally idle, the cost is relatively low, and the throughput is relatively high. As with most models, increasing total processors normally results in both increased throughput and increased cost. If models scaled linearly, the cost would remain constant across different processor counts, but generally, models don't scale linearly and cost increases with increasing processor count. It is strongly recommended that a user perform a load-balancing exercise on their proposed model run before undertaking a long production run.

CIME experimental cases have significant flexibility with respect to the layout of components across different hardware processors. In general, there are eight unique models (atm, lnd, rof, ocn, ice, glc, wav, cpl) that are managed independently by the CIME driver, each with a unique MPI communicator. In addition, the driver runs on the union of all processors and controls the sequencing and hardware partitioning.

See now to cusomize the PE layout for a detailed discussion of how to set processor layouts.

Model timing data

In order to perform a load balancing exercise, you must first be aware of the different types of timing information produced by every model run.

A summary timing output file is produced after every model run. This file is placed in `$CASEROOT/timing/ccsm_timing.$CASE.$date`, where $date is a datestamp set by CIME at runtime, and contains a summary of various information. The following provides a description of the most important parts of a timing file.

The first section in the timing output, CCSM TIMING PROFILE, summarizes general timing information for the run. The total run time and cost is given in several metrics including pe-hrs per simulated year (cost), simulated years per wall day (thoughput), seconds, and seconds per model day. This provides general summary information quickly in several units for analysis and comparison with other runs. The total run time for each component is also provided, as is the time for initialization of the model. These times are the aggregate over the total run and do not take into account any temporal or processor load imbalances.

The second section in the timing output, "DRIVER TIMING FLOWCHART", provides timing information for the driver in sequential order and indicates which processors are involved in the cost. Finally, the timings for the coupler are broken out at the bottom of the timing output file.

Separately, there is another file in the timing directory, ccsm_timing_stats.$date that accompanies the above timing summary. This second file provides a summary of the minimum and maximum of all the model timers.

There is one other stream of useful timing information in the cpl.log.$date file that is produced for every run. The cpl.log file contains the run time for each model day during the model run. This diagnostic is output as the model runs. You can search for tStamp in the cpl.log file to see this information. This timing information is useful for tracking down temporal variability in model cost either due to inherent model variability cost (I/O, spin-up, seasonal, etc) or possibly due to variability due to hardware. The model daily cost is generally pretty constant unless I/O is written intermittently such as at the end of the month.

Using model timing data

In practice, load-balancing requires a number of considerations such as which components are run, their absolute and relative resolution; cost, scaling and processor count sweet-spots for each component; and internal load imbalance within a component. It is often best to load balance the system with all significant run-time I/O turned off because this occurs very infrequently, typically one timestep per month, and is best treated as a separate cost as it can bias interpretation of the overall model load balance. Also, the use of OpenMP threading in some or all of the components is dependent on the hardware/OS support as well as whether the system supports running all MPI and mixed MPI/OpenMP on overlapping processors for different components. A final point is deciding whether components should run sequentially, concurrently, or some combination of the two with each other. Typically, a series of short test runs is done with the desired production configuration to establish a reasonable load balance setup for the production job. The timing output can be used to compare test runs to help determine the optimal load balance.

Changing the pe layout of the model has NO IMPACT on the scientific results. The basic order of operations and calling sequence is hardwired into the driver and that doesn't change when the pe layout is changed. There are some constraints on the ability of either the CESM or ACME fully active configuraiton fully concurrent. In particular, the atmosphere model always run sequentially with the ice and land for scientific reasons. As a result, running the atmosphere concurrently with the ice and land will result in idle processors in these components at some point in the timestepping sequence.

For more information about how the driver is implemented, see (Craig, A.P., Vertenstein, M., Jacob, R., 2012: A new flexible coupler for earth system modeling developed for CCSM4 and CESM1.0. International Journal of High Performance Computing Applications, 26, 31-42, 10.1177/1094342011428141).

In general, we normally carry out 20-day model runs with restarts and history turned off in order to find the layout that has the best load balance for the targeted number of processors. This provides a reasonable performance estimate for the production run for most of the runtime. The end of month history and end of run restart I/O is treated as a separate cost from the load balance perspective. To set up this test configuration, create your production case, and then edit env_run.xml and set STOP_OPTION to ndays, STOP_N to 20, and RESTART_OPTION to never. Seasonal variation and spin-up costs can change performance over time, so even after a production run has started, it's worthwhile to occasionally review the timing output to see whether any changes might be made to the layout to improve throughput or decrease cost.

In determining an optimal load balance for a specific configuration, two pieces of information are useful.

- Determine which component or components are most expensive.
- Understand the scaling of the individual components, whether they run faster with all MPI or mixed MPI/OpenMP decomposition strategies, and their optimal decompositions at each processor count. If the cost and scaling of the components are unknown, several short tests can be carried out with arbitrary component pe counts just to establish component scaling and sweet spots.

One method for determining an optimal load balance is as follows

- start with the most expensive component and a fixed optimal processor count and decomposition for that component
- test the systems, varying the sequencing/concurrency of the components and the pe counts of the other components
- identify a few best potential load balance configurations and then run each a few times to establish run-to-run variability and to try to statistically establish the faster layout

In all cases, the component run times in the timing output file can be reviewed for both overall throughput and independent component timings. Using the timing output, idle processors can be identified by considering the component concurrency in conjunction with the component timing.

In general, there are only a few reasonable component layout options.

- fully sequential
- fully sequential except the ocean running concurrently
- fully concurrent except the atmosphere run sequentially with the ice, rof, and land components
- finally, it makes best sense for the coupler to run on a subset of the atmosphere processors and that can be sequentially or concurrently run with the land and ice

The concurrency is limited in part by the hardwired sequencing in the driver. This sequencing is set by scientific constraints, although there may be some addition flexibility with respect to concurrency when running with mixed active and data models.

There are some general rules for finding optimal configurations:

- Make sure you have set a processor layout where each hardware processor is assigned to at least one component. There is rarely a reason to have completely idle processors in your layout.
- Make sure your cheapest components keep up with your most expensive components. In other words, a component that runs on 1024 processors should not be waiting on a component running on 16 processors.
- Before running the job, make sure the batch queue settings in the $CASE.run script are set correctly for the specific run being targetted. The account numbers, queue names, time limits should be reviewed. The ideal time limit, queues, and run length are all dependent on each other and on the current model throughput.
- Make sure you are taking full advantage of the hardware resources. If you are charged by the 32-way node, you might as well target a total processor count that is a multiple of 32.
- If possible, keep a single component on a single node. That usually minimizes internal component communication cost. That's obviously not possible if running on more processors than the size of a node.
- And always assume the hardware performance could have variations due to contention on the interconnect, file systems, or other areas. If unsure of a timing result, run cases multiple times.

Setting the time limits

In looking at the ccsm_timing.$CASE.$datestamp files for "Model Throughput", output like the following will be found:

```
` Overall Metrics: Model Cost: 327.14 pe-hrs/simulated_year (scale= 0.50) Model Throughput: 4.70 simulated_years/day `
```

The model throughput is the estimated number of model years that you can run in a wallclock day. Based on this, you can maximize $CASE.run queue limit and change $STOP_OPTION and $STOP_N in `env_run.xml`. For example, say a model's throughput is 4.7 simulated_years/day. On yellowstone(??), the maximum runtime limit is 6 hours. 4.7 model years/24 hours * 6 hours = 1.17 years. On the massively parallel computers, there is always some variability in how long it will take a job to run. On some machines, you may need to leave as much as 20% buffer time in your run to guarantee that jobs finish reliably before the time limit. For that reason we will set our model to run only one model year/job. Continuing to assume that the run is on yellowstone, in `$CASE.yellowstone.run` set:

```
` #BSUB -W 6:00 `
```

and `xmlchange` should be invoked as follows in `CASEROOT`:

```
` ./xmlchange STOP_OPTION=nyears ./xmlchange STOP_N=1 ./xmlchange REST_OPTION=nyears ./xmlchange REST_N=1 `
```

Testing with create_test

Using create_test

manage_testlists

Adding tests

Fortran Unit Testing

Introduction

What is a unit test?

A unit test is a fast, self-verying test of a small piece of code. A single unit test typically covers 10s to 100s of lines of code (e.g., a single function or small module). It typically runs in just milliseconds, and produces a simple pass/fail result.

Unit tests:

- Ensure that code remains correct as it is modified (in this respect, they complement the CIME system tests; the remaining bullet points are unique to unit tests).
- Ensure that new code is correct.
- Can help guide development, via test-driven development (TDD).
- Provide executable documentation of the intended behavior of a piece of code.
- Support development on your desktop machine.

Overview of unit test support in CIME

CIME comes with a set of tools to support building and running unit tests. These consist of:

1. CMake tools to support building and running tests via CMake and CTest.
2. A python script that provides a simple front-end to the CMake-based tests.

The Fortran unit tests use pFUnit, which is a Fortran unit testing framework that follows conventions of other xUnit frameworks (JUnit, etc.).

Running CIME's Fortran unit tests

These instructions assume that you are using a machine that already has pFUnit installed, along with the necessary support in CIME. If that is not the case, then see the section How to add unit testing support on your machine.

From the top-level cime directory, you can run all of CIME's Fortran unit tests simply by running:

```
> tools/unit_testing/run_tests.py --build-dir MY_BUILD_DIR
```

where you can replace `MY_BUILD_DIR` with a path to the directory where you would like the unit test build files to be placed. Once you have built the unit tests once (whether the build was successful or not), you can reuse the same build directory later to speed up the rebuild. There are a number of useful arguments to `run_tests.py`; for full usage information, run:

```
> tools/unit_testing/run_tests.py -h
```

If the build is successful, then you should get a message that looks like this:

```
================================================
Running CTest tests for __command_line_test__/__command_line_test__.
================================================
```

This will be followed by a list of tests, with a Pass/Fail message for each, like this:

```
Test project /Users/sacks/cime/unit_tests.0XHUkfqL/__command_line_test__/__command_line_test
      Start  1: avect_wrapper
 1/17 Test  #1: avect_wrapper ...................   Passed    0.02 sec
      Start  2: seq_map
 2/17 Test  #2: seq_map .........................   Passed    0.01 sec
      Start  3: glc_elevclass
 3/17 Test  #3: glc_elevclass ...................   Passed    0.01 sec
```

You should then see a final message like this:

```
100% tests passed, 0 tests failed out of 17
```

For machines that have a serial build of pFUnit available for the default compiler, these unit tests are run automatically as part of `scripts_regression_tests`.

How to add unit testing support on your machine

The below instructions assume that you have already ported CIME to your machine, by following the instructions in Porting and Validating CIME on a new Platform. Once you have done that, you can add unit testing support by building pFUnit on your machine and then pointing to the build in `config_compilers.xml`.

At a minimum, you should do a serial build of pFUnit (without MPI or OpenMP), using the default compiler on your machine (according to `config_machines.xml`). That is the default used by `run_tests.py`, and is required for `scripts_regression_tests.py` to run the unit tests on your machine. Optionally, you can also provide pFUnit builds with other supported compilers on your machine. If you'd like, you can also provide additional pFUnit builds with other combinations of MPI and OpenMP on or off. However, at this time, no unit tests require parallel support, so there is no benefit gained by providing MPI-enabled builds.

Building pFUnit

To perform a serial build of pFUnit, follow these instructions:

1. Download pFUnit from https://sourceforge.net/projects/pfunit/
2. Set up your environment to be similar to the environment used in system builds of CIME. For example, load the appropriate compilers into your path. An easy way to achieve this is to run:

   ```
   > $CIMEROOT/tools/configure --mpilib mpi-serial
   ```

   (with an optional `--compiler` argument; you'll also want to change the `--mpilib` argument if you're doing an MPI-enabled build). Then source either `./.env_mach_specific.sh` or `./.env_mach_specific.csh`, depending on your shell.
3. For convenience, set the `PFUNIT` environment variable to point to the location where you want to install pFUnit. For example (in bash):

   ```
   > export PFUNIT=/glade/p/cesmdata/cseg/tools/pFUnit/pFUnit3.2.8_cheyenne_Intel17.0.1_noM
   ```
4. Configure and build pFUnit:

   ```
   > mkdir build
   > cd build
   > cmake -DMPI=NO -DOPENMP=NO -DCMAKE_INSTALL_PREFIX=$PFUNIT ..
   > make -j 4
   ```
5. Run pFUnit's self-tests:

   ```
   > make tests
   ```
6. Install pFUnit in the directory you specified earlier:

   ```
   > make install
   ```

If you'd like, you can then repeat this process with different compiler environments and/or different choices of `-DMPI` and `-DOPENMP` in the cmake step (each of these can have the value `NO` or `YES`). Make sure to choose a different installation directory for each of these, by setting the `PFUNIT` variable differently.

Adding to the xml file

You then need to tell CIME about your pFUnit build(s). To do this, specify the appropriate path(s) using the `PFUNIT_PATH` element in `config_compilers.xml`. For a serial build, this will look like:

```
<PFUNIT_PATH MPILIB="mpi-serial" compile_threaded="false">$ENV{CESMDATAROOT}/tools/pFUnit/pF
```

It is important that you provide the `MPILIB` and `compile_threaded` attributes. `MPILIB` should be `mpi-serial` for a pFUnit build with `-DMPI=NO`, or the name of the mpi library you used for a pFUnit build with `-DMPI=YES` (e.g., `mpich`; this should be one of this machine's MPI libraries specified by `MPILIBS` in `config_machines.xml`). `compile_threaded` should be either `true` or `false` depending on the value of `-DOPENMP`.

Once you have done this, you should be able to run the unit tests by following the instructions in Running CIME's Fortran unit tests.

How to write a new unit test

TODO: Need to write this section. This will draw on some of the information in sections 3 and 4 of https://github.com/NCAR/cesm_unit_test_tutorial (though without the clm and cam stuff).

General guidelines for writing unit tests

Unit tests typically test a small piece of code (e.g., order 10 - 100 lines, such as a single function or small-ish class).

Good unit tests are "FIRST" (https://pragprog.com/magazines/2012-01/unit-tests-are-first):

- Fast (order milliseconds or less)
    - This means that, generally, they should not do any file i/o. Also, if you are testing a complex function, test it with a simple set of inputs - not a 10,000-element array that will require a few seconds of runtime to process.
- Independent
    - This means that test Y shouldn't depend on some global variable that was created by test X. Dependencies like this cause problems if the tests run in a different order, if one test is dropped, etc.
- Repeatable
    - This means, for example, that you shouldn't generate random numbers in your tests.
- Self-verifying
    - This means that you shouldn't write a test that writes out its answers for manual comparison. Tests should generate an automatic pass/fail result.
- Timely
    - This means that the tests should be written *before* the production code (Test Driven Development), or immediately afterwards - not six months later when it's time to finally merge your changes onto the trunk, and have forgotten the details of what you have written. Much of the benefit of unit tests comes from developing them alongside the production code.

Good unit tests test a single, well-defined condition. This generally means that you make a single call to the function / subroutine that you're testing, with a single set of inputs. This means that you usually need multiple tests of the function / subroutine, in order to test all of its possible behaviors. The main reasons for testing a single condition in each test are:

- This makes it easier to pinpoint a problem when a test fails
- This makes it easier to read and understand the tests, allowing the tests to serve as useful documentation of how the code should operate

A good unit test has four distinct pieces:

1. **Setup**: e.g., create variables that will be needed for the routine you're testing. For simple tests, this piece may be empty.
2. **Exercise**: Call the routine you're testing
3. **Verify**: Call assertion methods to ensure that the results matched what you expected
4. **Teardown**: e.g., deallocate variables. For simple tests, this piece may be empty. **However, if this is needed, you should almost always do this teardown in the special tearDown routine, as discussed in the sections,** Defining a test class in order to define setUp and tearDown methods and More on test teardown.

If you have many tests of the same subroutine, then you'll often find quite a lot of duplication between the tests. It's good practice to extract major areas of duplication to their own subroutines in the .pf file, which can be called by your tests. This aids the understandability and maintainability of your tests. pFUnit knows which subroutines are tests and which are "helper" routines because of the `@Test` directives: You only add a `@Test` directive for your tests, not for your helper routines.

More details on writing pFUnit-based unit tests

Assertion methods

pFUnit provides many assertion methods that you can use in the Verify step. Some of the most useful are the following:

- `@assertEqual(expected, actual)`
  - Ensures that expected == actual
  - Accepts an optional `tolerance` argument giving the tolerance for real-valued comparisons
- `@assertLessThan(expected, actual)`
  - Ensures that expected < actual
- `@assertGreaterThan(expected, actual)`
  - Ensures that expected > actual
- `@assertLessThanOrEqual(expected, actual)`
- `@assertGreaterThanOrEqual(expected, actual)`
- `@assertTrue(condition)`
  - It's better to use the two-valued assertions above, if possible. For example, use `@assertEqual(foo, bar)` rather than `@assertTrue(foo == bar)`: the former gives more information if the test fails.
- `@assertFalse(condition)`
- `@assertIsFinite(value)`
  - Ensures that the result is not NaN or infinity
- `@assertIsNan(value)`
  - Can be useful for failure checking, e.g., if your function returns NaN to signal an error

Comparison assertions accept an optional `tolerance` argument, which gives the tolerance for real-valued comparisons.

In addition, all of the assertion methods accept an optional `message` argument, which gives a string that will be printed if the assertion fails. If no message is provided, you will be pointed to the file and line number of the failed assertion.

Defining a test class in order to define setUp and tearDown methods

As noted in the comments in `test_circle.pf`, the definition of a test class (here, `TestCircle`) is optional. However, it's convenient to define a minimal test class when you first write a new .pf file:

```
@TestCase
type, extends(TestCase) :: TestCircle
 contains
   procedure :: setUp
   procedure :: tearDown
end type TestCircle
```

Defining this test class allows you to take advantage of some useful pFUnit features like the setUp and tearDown methods.

If you define this test class, then you also need to:

- Define setUp and tearDown subroutines. These can start out empty:

```
subroutine setUp(this)
  class(TestCircle), intent(inout) :: this
end subroutine setUp
```

```
subroutine tearDown(this)
  class(TestCircle), intent(inout) :: this
end subroutine tearDown
```

- Add an argument to each test subroutine, of class `TestCircle` (or whatever you called your test class). By convention, this argument is named `this`.

Code in the setUp method will be executed before each test. This is convenient if you need to do some setup that is the same for every test.

Code in the tearDown method will be executed after each test. This is often used to deallocate memory. See the section, More on test teardown for details.

You can add any data or procedures to the test class. Adding data is particularly useful, as this can be a way for the setUp and tearDown methods to interact with your tests: The setUp method can fill a class variable with data, which can then be used by your tests (accessed via `this%somedata`). Conversely, if you want the tearDown method to deallocate a variable, that variable cannot be local to your test subroutine. Instead, you can make the variable a member of the class, so that the tearDown method can access it.

So, for example, if you have this variable in your test class (as in the example):

```
real(r8), pointer :: somedata(:)
```

Then `somedata` can be created in the setUp method (if it needs to be the same for every test). Alternatively, it can be created in each test routine that needs it (if it differs from test to test, or some tests don't need it at all). Its creation can look like:

```
allocate(this%somedata(5))
this%somedata(:) = [1,2,3,4,5]
```

Then your tearDown method can have code like this:

```
if (associated(this%somedata)) then
  deallocate(this%somedata)
end if
```

More on test teardown

All of the tests in a single test executable - which, for CIME, typically means all of the tests defined in all `.pf` files in a single test directory - will execute one after another in one run of the executable. This means that, if you don't clean up after yourself, tests can interact with each other. In the best case, this can mean you get a memory leak. In the worst case, it can mean that the pass / fail status of tests depends on what other tests have run before them, making your unit tests unrepeatable and unreliable. **As a general rule, you should deallocate any pointers that your test allocated, reset any global variables to some known, initial state, and do other, similar cleanup for resources that may be shared by multiple tests.**

As described in the section, Defining a test class in order to define setUp and tearDown methods, code in the tearDown method will be executed after each test. This is often used to do cleanup operations after each test. **Any teardown like this should generally happen in this tearDown method. This is because, if an assertion fails, the test aborts. So any teardown code in the test method (following the failed assert statement) is skipped, which can lead other tests to fail or give unexpected results. But this tearDown method is still called in this case, making it a safe place to put teardown that needs to be done regardless of whether the test passed or failed (which is the case for most teardown).** In order for this to work, you sometimes need to move variables that might otherwise be subroutine-local to the class - because the tearDown method can access class instance variables, but not subroutine-local variables.

Note that, in Fortran2003, allocatable variables are automatically deallocated when they go out of scope, but pointers are not. So you need to explicitly deallocate any pointers that have been allocated, either in test setup or in the execution of the routine you're testing.

CIME makes extensive use of global variables: variables declared in some module, which may be used (directly or indirectly) by the routine you're testing. If your test has allocated or modified any global variables, it is important to reset them to their initial state in the teardown portion of the test. (Incidentally, this is just one of many reasons to prefer explicit argument-passing over the use of global variables.)

pFUnit documentation and examples

Some pFUnit documentation is available here: http://pfunit.sourceforge.net/

If you download pFUnit (from http://sourceforge.net/projects/pfunit/), you can find more extensive documentation and examples in the following places. Among other things, this can show you other assertion methods that are available:

- documentation/pFUnit3-ReferenceManual.pdf
- Examples/
- tests/
  - These are tests of the pFUnit code itself, written in pFUnit. You can see many uses of pFUnit features in these tests.

Finding more documentation and examples in CIME

Documentation of the unit test build system

The CMake build infrastructure is in `$CIMEROOT/src/externals/CMake`.

The infrastructure for building and running tests with `run_tests.py` is in `$CIMEROOT/tools/unit_testing`. That directory also contains some general documentation about how to use the CIME unit test infrastructure (in the `README` file), and examples (in the `Examples` directory).

Finding more detailed examples

At this point, there are many examples of unit tests in CIME, some simple and some quite complex. You can find these by looking for files with the '.pf' extension:

```
> find . -name '*.pf'
```

You can also see examples of the unit test build scripts by viewing the CMakeLists.txt files throughout the source tree.

Multi-instance component functionality

The CIME coupling infrastructure has the capability to run multiple component instances under one model executable. The only caveat to this usage is that if N multiple instances of any one active component is used, then N multiple instances of ALL active components are required. More details are discussed below. The primary motivation for this development was to be able to run an ensemble Kalman-Filter for data assimilation and parameter estimation (e.g. UQ). However, it also provides you with the ability to run a set of experiments within a single model executable where each instance can have a different namelist, and have all the output go to one directory.

In the following an F compset will be used as an illustration. Utilizing the multiple instance code involves the following steps:

1. create the case

```
> create_newcase -case Fmulti -compset F -res ne30_g16
> cd Fmulti
```

2. Lets assume the following out of the box pe-layout

```
NTASKS(ATM)=128, NTHRDS(ATM)=1, ROOTPE(ATM)=0, NINST(ATM)=1
NTASKS(LND)=128, NTHRDS(LND)=1, ROOTPE(LND)=0, NINST(LND)=1
NTASKS(ICE)=128, NTHRDS(ICE)=1, ROOTPE(ICE)=0, NINST(ICE)=1
NTASKS(OCN)=128, NTHRDS(OCN)=1, ROOTPE(OCN)=0, NINST(OCN)=1
NTASKS(GLC)=128, NTHRDS(GLC)=1, ROOTPE(GLC)=0, NINST(GLC)=1
NTASKS(WAV)=128, NTHRDS(WAV)=1, ROOTPE(WAV)=0, NINST(WAV)=1
NTASKS(CPL)=128, NTHRDS(CPL)=1, ROOTPE(CPL)=0
```

In this F compset, the atm, lnd, rof are active components, the ocn is a prescribed data component, cice is a mixed prescribed/active component (ice-coverage is prescribed) and glc and wav are stub components. Lets say we want to run 2 instances of CAM in this experiment. The current implementation of multi-instances will also require you to run 2 instances of CLM, CICE and RTM. However, you have the flexibility to run either 1 or 2 instances of DOCN (we can ignore glc and wav since they do not do anything in this compset). To run 2 instances of CAM, CLM, CICE, RTM and DOCN, all you need to do is to invoke the following command in your `$CASEROOT`:

```
./xmlchange NINST_ATM=2
./xmlchange NINST_LND=2
./xmlchange NINST_ICE=2
./xmlchange NINST_ROF=2
./xmlchange NINST_OCN=2
```

As a result of this, you will have 2 instances of CAM, CLM and CICE (prescribed), RTM, and DOCN, each running concurrently on 64 MPI tasks **TODO: put in reference to xmlchange".**

3. Setup the case

```
> ./case.setup
```

New user_nl_xxx_NNNN file (where NNNN is the number of the component instances) will be generated when **case.setup** is called. In particular, calling **case.setup** with the above `env_mach_pes.xml` file will result in the following `user_nl_*` files in `$CASEROOT`

```
user_nl_cam_0001,  user_nl_cam_0002
user_nl_cice_0001, user_nl_cice_0002
user_nl_clm_0001,  user_nl_clm_0002
user_nl_rtm_0001,  user_nl_rtm_0002
user_nl_docn_0001, user_nl_docn_0002
user_nl_cpl
```

and the following `*_in_*` files and `*txt*` files in $CASEROOT/CaseDocs:

```
atm_in_0001, atm_in_0002
docn.streams.txt.prescribed_0001, docn.streams.txt.prescribed_0002
docn_in_0001, docn_in_0002
docn_ocn_in_0001, docn_ocn_in_0002
drv_flds_in, drv_in
ice_in_0001, ice_in_0002
lnd_in_0001, lnd_in_0002
rof_in_0001, rof_in_0002
```

The namelist for each component instance can be modified by changing the corresponding user_nl_xxx_NNNN file for that component instance. Modifying the user_nl_cam_0002 will result in the namelist changes you put in to be active ONLY for instance 2 of CAM. To change the DOCN stream txt file instance 0002, you should place a copy of `docn.streams.txt.prescribed_0002` in `$CASEROOT` with the name `user_docn.streams.txt.prescribed_0002` and modify it accordlingly.

It is also important to stress the following points:

1. **Different component instances can ONLY differ by differences in namelist settings - they are ALL using the same model executable.**
2. Only 1 coupler component is supported currently in multiple instance implementation.
3. `user_nl_xxx_NN` files once they are created by **case.setup** *ARE NOT* removed by calling **case.setup -clean**.
4. In general, you should run multiple instances concurrently (the default setting in `env_mach_pes.xml`). The serial setting is only for EXPERT USERS in upcoming development code implementations.

Adding new cases

Adding compsets

Adding grids

CIME is accompanied by support for numerous out-of-the box model resolutions. To see the supported grids call **manange_case --query-grids**. In general, CIME model grids are associated with a specific combination of atmosphere, land, land-ice, river-runoff and ocean/ice grids. The naming convention for these grids still only involves atmosphere, land, and ocean/ice grid specifications.

The most common resolutions have the atmosphere and land components on one grid and the ocean and ice on a second grid. The following overview assumes that this is the case. The naming convention looks like *f19_g16*, where the f19 indicates that the atmosphere and land are on the 1.9x2.5 (finite volume dycore) grid while the g16 means the ocean and ice are on the gx1v6 one-degree displaced pole grid.

CIME provides support for you to add your own specific component grid combinations. To achieve this, CIME has a `$CIMEROOT/tools/mapping/`. A brief list of the steps needed to add a new component grid to the model system follows. Again, this process can be simplified if the atmosphere and land are running on the same grid.

1. You must have or generate SCRIP grid files for the atmosphere, land, ocean, land-ice, river and wave component grids that will comprise your model grid. If your new model grid just introduces one new grid, then you can leverage SCRIP grid files that are already in place for the other components. At present there is no supported functionality for creating the SCRIP format file.

2. Build the `check_map` utility. When you add new user-defined grid files, you will also need to generate a set of mapping files so the coupler can send data from a component on one grid to a component on another grid. There is an ESMF tool that tests the mapping file by comparing a mapping of a smooth function to its true value on the destination grid. We have tweaked this utility to test a suite of of smooth functions, as well as ensure conservation (when the map is conservative). Before generating mapping functions it is *highly recommended* that you build this utility.

   To build this tool, follow the instructions in `$CCSMROOT/mapping/check_maps/INSTALL`. As with many of the steps in this document, you will need to have the ESMF toolkit installed. It is installed by default on most NCAR computers.

3. Generate `atm <-> ocn`, `atm <-> wav`, `lnd <-> rof`, `lnd <-> glc`, `ocn <-> wav` and `rof -> ocn` mapping files.

   Using the SCRIP grid files from step one, you must generate a set of conservative (area-averaged) and non-conservative (patch and bilinear) mapping files. You can do this by calling **gen_cesm_maps.sh** in `$CCSMROOT/tools/mapping/gen_mapping_files/`. This script generates all the mapping files needed except `rof -> ocn`, which is discussed below. This script uses the ESMF offline weight generation utility, which you must build *prior* to running **gen_cesm_maps.sh**.

   The `README file` in the `gen_mapping_files/` directory contains details on how to run **gen_cesm_maps.sh**. The basic usage is

```
> cd $CCSMROOT/mapping/gen_mapping_files
> ./gen_cesm_maps.sh \
   --fileocn  <input SCRIP ocn_grid full pathname>  \
   --fileatm  <input SCRIP atm grid full pathname>  \
   --filelnd  <input SCRIP lnd grid full pathname>  \
   --filertm  <input SCRIP rtm grid full pathname>  \
   --nameocn  <ocnname in output mapping file> \
   --nameatm  <atmname in output mapping file> \
   --namelnd  <lndname in output mapping file> \
   --namertm  <rtmname in output mapping file>
```

   This command will generate the following mapping files:

```
map_atmname_TO_ocnname_aave.yymmdd.nc
map_atmname_TO_ocnname_blin.yymmdd.nc
map_atmname_TO_ocnname_patc.yymmdd.nc
map_ocnname_TO_atmname_aave.yymmdd.nc
map_ocnname_TO_atmname_blin.yymmdd.nc
map_atmname_TO_lndname_aave.yymmdd.nc
map_atmname_TO_lndname_blin.yymmdd.nc
map_lndname_TO_atmname_aave.yymmdd.nc
map_ocnname_TO_lndname_aave.yymmdd.nc
map_lndname_TO_rtmname_aave.yymmdd.nc
map_rtmname_TO_lndname_aave.yymmdd.nc
```

Note

You do not need to specify all four grids. For example, if you are running with the atmosphere and land on the same grid, then you do not need to specify the land grid (and atm<->rtm maps will be generated). If you also omit the runoff grid, then only the 5 atm<->ocn maps will be generated.

Note

ESMF_RegridWeightGen runs in parallel, and the `gen_cesm_maps.sh` script has been written to run on yellowstone. To run on any other machine, you may need to add some environment variables to `$CCSMROOT/mapping/gen_mapping_files/gen_ESMF_mapping_file/create_ESMF_map.sh` -- search for hostname to see where to edit the file.

4. Generate atmosphere, land and ocean / ice domain files.

   Using the conservative ocean to land and ocean to atmosphere mapping files created in the previous step, you can create domain files for the atmosphere, land, and ocean; these are basically grid files with consistent masks and fractions. You make these files by calling **gen_domain** in `$CCSMROOT/mapping/gen_domain_files`. The `INSTALL` file in the `gen_domain_files/` directory contains details on how to build the **gen_domain** executable. After you have built it, the README in the same directory contains details on how to use the tool. The basic usage is:

```
> ./gen_domain -m ../gen_mapping_files/map_ocnname_TO_lndname_aave.yymmdd.nc -o ocnname
> ./gen_domain -m ../gen_mapping_files/map_ocnname_TO_atmname_aave.yymmdd.nc -o ocnname
```

   These commands will generate the following domain files:

```
domain.lnd.lndname_ocnname.yymmdd.nc
domain.ocn.lndname_ocnname.yymmdd.nc
domain.lnd.atmname_ocnname.yymmdd.nc
domain.ocn.atmname_ocnname.yymmdd.nc
domain.ocn.ocnname.yymmdd.nc
```

Note

The input atmosphere grid is assumed to be unmasked (global). Land cells whose fraction is zero will have land mask = 0.

Note

If the ocean and land grids *are identical* then the mapping file will simply be unity and the land fraction will be one minus the ocean fraction.

5. If you are adding a new ocn or rtm grid, create a new rtm->ocn mapping file. (Otherwise you can skip this step.) The process for mapping from the runoff grid to the ocean grid is currently undergoing many changes. At this time, if you are running with a new ocean or runoff grid, please contact Michael Levy (mlevy_AT_ucar_DOT_edu) for assistance. If you are running with standard ocean and runoff grids, the mapping file should already exist and you do not need to generate it.
6. CESM specific: If you are adding a new atmosphere grid, this means you are also generating a new land grid, and you will need to create a new CLM surface dataset. (Otherwise you can skip this step). You need to first generate mapping files for CLM surface dataset (since this is a non-standard grid).

```
  > cd $CCSMROOT/models/lnd/clm/tools/mkmapdata
  > ./mkmapdata.sh --gridfile <lnd SCRIP grid file> --res <atm resolution name> --gridty

These mapping files are then used to generate CLM surface dataset. Below is an example f

::

   > cd  $CCSMROOT/models/lnd/clm/tools/mksurfdata_map
   > ./mksurfdata.pl -res usrspec -usr_gname <atm resolution name> -usr_gdate yymmdd -y
```

7. Create grid file needed for create_newcase. The next step is to add the necessary new entries in the appropriate `config_grids.xml` file. You will need to modify `$CIMEROOT/cime_config/cesm/config_grids.xml` or `$CIMEROOT/cime_config/acme/config_grids.xml` depending on the value of `$CIME_MODEL`. You will need to:

   - add a single `<model_grid>` entry
   - add possibly multiple `<domain>` entries for every new component grid that you have added
   - add possibly multiple `<gridmap>` entries for all the new component combinations that require new mapping files

8. Test new grid.

   Below assume that the new grid is an atmosphere grid.

```
Test the new grid with all data components.
(write an example)
Test the new grid with CAM(newgrid), CLM(newgrid), DOCN(gx1v6), DICE(gx1v6)
(write an example)
```

Adding components

FAQ

The following section contains examples for a variety of use cases, spanning a beginner to expert-only range.

A Basic Example

This specifies all the steps necessary to create, set up, build, and run a case. The following assumes that `$CIMEROOT` is the root directory of CIME.

1. Create a new case named EXAMPLE_CASE in the ~/cesm directory. Use an 1850 control compset at 1-degree resolution on yellowstone.

```
> cd $CIME/scripts
> ./create_newcase -case ~/EXAMPLE_CASE -compset B1850_CN -res f09_g16
```

2. Go to the $CASEROOT directory. Edit env_mach_pes.xml if a different pe-layout is desired first. Then set up and build the case.

```
> cd ~/EXAMPLE_CASE
> ./case.setup
> ./case.build
```

3. Go back to the case directory, set the job to run 12 model months and increase the job wallclock time

```
> cd ~/EXAMPLE_CASE
> xmlchange STOP_OPTION=nmonths
> xmlchange STOP_N=12
> xmlchange JOB_WALLCLOCK_TIME=06:00
> ./case.submit
```

4. Make sure the run succeeded. Look for the following line at the end of the cpl.log file in your run directory.:

```
(seq_mct_drv): ===============       SUCCESSFUL TERMINATION OF CPL7-CESM ===============
```

5. Set it to resubmit itself 10 times so that it will run a total of 11 years (including the initial year), and resubmit the case. (Note that a resubmit will automatically change the run to be a continuation run).:

```
> xmlchange RESUBMIT=10
> case.submit
```

Setting up a branch or hybrid run

A branch or hybrid run uses initialization data from a previous run. First you need to create a new case.

```
> cd $CIMEROOT/scripts
> create_newcase -case ~/EXAMPLE_CASEp -compset B1850 -res f09_g16
> cd ~/EXAMPLE_CASEp
```

For a branch run, modify `env_run.xml` to branch from EXAMPLE_CASE at year 0001-02-01.

```
> xmlchange RUN_TYPE=branch
> xmlchange RUN_REFCASE=EXAMPLE_CASE
> xmlchange RUN_REFDATE=0001-02-01
```

For a hybrid run, modify `env_run.xml` to start up from EXAMPLE_CASE at year 0001-02-01.

```
> xmlchange RUN_TYPE=hybrid
> xmlchange RUN_REFCASE=EXAMPLE_CASE
> xmlchange RUN_REFDATE=0001-02-01
```

For a branch run, `env_run.xml` for EXAMPLE_CASEp should be identical to EXAMPLE_CASE, except for the `$RUN_TYPE` setting. In addition, any modifications introduced into any of the `~/EXAMPLE_CASE/user_nl_*` files, should be re-introduced into the corresponding files in EXAMPLE_CASEp.

Set up and build the case executable.

```
> ./case.setup
> ./case.build
```

Prestage the necessary restart/initial data in `$RUNDIR`. Assume that the data was created in `/user/archive/EXAMPLE_CASE/rest/0001-02-01-00000`.

```
> cd $RUNDIR
> cp /user/archive/EXAMPLE_CASE/rest/0001-02-01-00000/* .
```

It is assumed that you already have a valid load-balanced scenario. Go back to the case directory, set the job to run 12 model months.

```
> cd ~/EXAMPLE_CASEp
> xmlchange STOP_OPTION=nmonths
> xmlchange STOP_N=12
> xmlchange JOB_WALLCLOCK_TIME=06:00
> case.submit
```

Make sure the run succeeded. Look for the following line at the end of the cpl.log file in your run directory.

```
(seq_mct_drv): ===============       SUCCESSFUL TERMINATION OF CPL7-CCSM ===============
```

Change the run to a continuation run. Set it to resubmit itself 10 times so that it will run a total of 11 years (including the initial year), then resubmit the case.

```
> xmlchange CONTINUE_RUN=TRUE
> xmlchange RESUMIT=10
> case.submit
```

Indices and tables

- **genindex**

- **modindex**
- **search**

CIME Data Models

Introduction

Overview

The CIME data models perform the basic function of reading external data files, modifying those data, and then sending the data to the driver via the CIME coupling interfaces. The fields sent to the driver are the same as those that would be sent by an active component. This takes advantage of the fact that the driver and other models have no fundamental knowledge of whether another component is fully active or just a data model. So, for example, the data atmosphere model (datm) sends the same fields as the prognostic Community Atmosphere Model (CAM). However, rather than determining these fields prognostically, most data models simply read prescribed data.

The data models typically read gridded data from observations or reanalysis products. Out of the box, they often provide a few possible data sources and/or time periods that you can choose from when setting up a case. However, data models can also be configured to read output from a previous coupled run. For example, you can perform a fully-coupled run in which you ask for particular extra output streams; you can then use these saved "driver history" files as inputs to datm to run a later land-only spinup.

In some cases, data models have prognostic functionality, that is, they also receive and use data sent by the driver. However, in most cases, the data models are not running prognostically and have no need to receive any data from the driver.

The CIME data models have parallel capability and share significant amounts of source code. Methods for reading and interpolating data have been established and can easily be reused: The data model calls strdata ("stream data") methods which then call stream methods. The stream methods are responsible for managing lists of input data files and their time axis. The information is then passed up to the strdata methods where the data is read and interpolated in space and time. The interpolated data is passed up to the data model where final fields are derived, packed, and returned to the driver.

Design

Data models function by reading in different streams of input data and interpolating those data both spatially and temporally to the appropriate final model grid and model time.

- **Each data model**
  - communicates with the driver with fields on only the data model model grid.
  - can be associated with multiple streams
  - is associated with only one datamode value (specified in the `shr_strdata_nml` namelist group)
  - has an xml variable in `env_run.xml` that specifies its mode. These are: `DATM_MODE`, `DICE_MODE`, `DLND_MODE`, `DOCN_MODE`, `DROF_MODE`, `DWAV_MODE`.
- **Each ``DXXX_MODE`` xml variable variable specfies 2 things:**
  - the list of streams that are associated with the data model.
  - a `datamode` namelist variable that is associated with each data model and that determines if additional operations need to be performed on on the input streams before returning to the driver.

    at a minimum, all data models support `datamode` values of `NULL` and `COPYALL`.
    - `NULL` - turns off the data model as a provider of data to the coupler.
    - `COPYALL` - copies all fields directly from the input data streams. Any required fields not found on an input stream will be set to zero.
- **Each data model stream**
  - can be associated with multiple stream input files (specified in the `shr_strdata_nml` namelist group).

- **Each stream input file**
  - can contain data on a unique grid and unique temporal time stamps.
  - is interpolated to a single model grid and the present model time.

More details of the data model design are covered in design details.

Namelist Input

Each data model has two namelist groups in its input namelist file: a **stream-dependent** and a **stream-independent** namelist group.

The stream-dependent namelist group (`shr_strdata_nml`) specifies the data model mode, stream description text files, and interpolation options. The stream description files will be provided as separate input files and contain the files and fields that need to be read. The stream-independent namelist group (one of `[datm_nml, dice_nml, dlnd_nml, docn_nml, drof_nml, dwav_nml]`) contains namelist input such as the data model decomposition, etc.

From a user perspective, for any data model, it is important to know what modes are supported and the internal field names in the data model. That information will be used in the strdata namelist and stream input files.

Users will primarily setup different data model configurations through namelist settings. **The strdata and stream input options and format are identical for all data models**. The data model specific namelist has significant overlap between data models, but each data model has a slightly different set of input namelist variables and each model reads that namelist from a unique filename. The detailed namelist options for each data model will be described later, but each model will specify a filename or filenames for strdata namelist input and each strdata namelist will specify a set of stream input files.

The following example illustrates the basic set of namelist inputs:

```
&dlnd_nml
   decomp = '1d'
/
&shr_strdata_nml
  dataMode   = 'CPLHIST'
  domainFile = 'grid.nc'
  streams    = 'streama', 'streamb', 'streamc'
  mapalgo    = 'interpa', 'interpb', 'interpc'
/
```

As mentioned above, the `dataMode` namelist variable that is associated with each data model specifies if there is any additional operations that need to be performed on that data model's input streams before return to the driver. At a minimum, all data models support `datamode` values of `NULL` and `COPYALL`.

- `NULL` - turns off the data model as a provider of data to the coupler.
- `COPYALL` - copies all fields directly from the input data streams. Any required fields not found on an input stream will be set to zero.

Three stream description files are then expected to be available, `streama`, `streamb` and `streamc`. Those files specify the input data filenames, input data grids, and input fields that are expected, among other things. The stream files are **not** Fortran namelist format. Their format and options will be described later. As an example, one of the stream description files might look like

```
<stream>
   <dataSource>
      GENERIC
   </dataSource>
   <fieldInfo>
      <variableNames>
         dn10  dens
         slp_  pslv
         q_10  shum
         t_10  tbot
         u_10  u
         v_10  v
```

```
      </variableNames>
      <filePath>
         /glade/proj3/cseg/inputdata/atm/datm7/NYF
      </filePath>
      <offset>
         0
      </offset>
      <fileNames>
         nyf.ncep.T62.050923.nc
      </fileNames>
   </fieldInfo>
   <domainInfo>
      <variableNames>
         time   time
         lon    lon
         lat    lat
         area   area
         mask   mask
      </variableNames>
      <filePath>
         /glade/proj3/cseg/inputdata/atm/datm7/NYF
      </filePath>
      <fileNames>
         nyf.ncep.T62.050923.nc
      </fileNames>
   </domainInfo>
</stream>
```

In general, these examples of input files are not complete, but they do show the general hierarchy and feel of the data model input.

Next Sections

In the next sections, more details will be presented including a full description of the science modes and namelist settings for the data atmosphere, data land, data runoff, data ocean, and data ice models; namelist settings for the strdata namelist input; a description of the format and options for the stream description input files; and a list of internal field names for each of the data components. The internal data model field names are important because they are used to setup the stream description files and to map the input data fields to the internal data model field names.

Input Streams

Overview

An *input data stream* is a time-series of input data files where all the fields in the stream are located in the same data file and all share the same spatial and temporal coordinates (ie. are all on the same grid and share the same time axis). Normally a time axis has a uniform dt, but this is not a requirement.

The data models can have multiple input streams.

The data for one stream may be all in one file or may be spread over several files. For example, 50 years of monthly average data might be contained all in one data file or it might be spread over 50 files, each containing one year of data.

The data models can *loop* over stream data -- repeatedly cycle over some subset of an input stream's time axis. When looping, the models can only loop over whole years. For example, an input stream might have SST data for years 1950 through 2000, but a model could loop over the data for years 1960 through 1980. A model *cannot* loop over partial years, for example, from 1950-Feb-10 through 1980-Mar-15.

The input data must be in a netcdf file and the time axis in that file must be CF-1.0 compliant.

There are two main categories of information that the data models need to know about a stream:

- data that describes what a user wants -- what streams to use and how to use them -- things that can be changed by a user.
- data that describes the stream data -- meta-data about the inherent properties of the data itself -- things that cannot be changed by a user.

Generally, information about what streams a user wants to use and how to use them is input via the strdata ("stream data") Fortran namelist, while meta-data that describes the stream data itself is found in an xml-like text file called a "stream description file."

Stream Data and shr_strdata_nml namelists

The stream data (referred to as *strdata*) input is set via a fortran namelist called `shr_strdata_nml`. That namelist, the associated strdata datatype, and the methods are contained in the share source code file, `shr_strdata_mod.F90`. In general, strdata input defines an array of input streams and operations to perform on those streams. Therefore, many namelist inputs are arrays of character strings. Different variables of the same index are associated. For instance, mapalgo(1) spatial interpolation will be performed between streams(1) and the target domain.

Each data model as an associated input namelist file, `xxx_in`, where `xxx=[datm,dlnd,dice,docn,drof,dwav]`.

The input namelist file for each data model has a stream dependent namelist group, `shr_strdata_nml`, and a stream independent namelist group. The `shr_strdata_nml` namelist variables **are the same for all data models**.

File	Namelist Groups
datm_in	datm_nml, shr_strdata_nml
dice_in	dice_nml, shr_strdata_nml
dlnd_in	dlnd_nml, shr_strdata_nml
docn_in	docn_nml, shr_strdata_nml
drof_in	drof_nml, shr_strdata_nml
dwav_in	dwav_nml, shr_strdata_nml

The following table summaries the `shr_strdata_nml` entries.

Namelist	Description
dataMode	component specific mode. Each CIME data model has its own datamode values as described below: datm dataMode dice dataMode dlnd dataMode docn dataMode drof dataMode dwav dataMode
domainFile	component domain (all streams will be mapped to this domain). Spatial gridfile associated with the strdata. grid information will be read from this file and that grid will serve as the target grid for all input data for this strdata input. If the value is **null** then the domain of the first stream will be used as the component domain default="null"

streams	character array (up to 30 elemnets) of input stream filenames and associated years of data. Each array entry consists of a stream_input_filename year_align year_first year_last. The stream_input_filename is a stream text input file and the format and options are described elsewhere. The year_align, year_first, and year_last provide information about the time axis of the file and how to relate the input time axis to the model time axis. default="null".
fillalgo	array (up to 30 elements) of fill algorithms associated with the array of streams. Valid options are just copy (ie. no fill), special value, nearest neighbor, nearest neighbor in "i" direction, or nearest neighbor in "j" direction. valid values: 'copy','spval','nn','nnoni','nnonj' default value='nn'
fillmask	array (up to 30 elements) of fill masks. valid values: "nomask,srcmask,dstmask,bothmask" default="nomask"
fillread	array (up to 30 elements) fill mapping files to read. Secifies the weights file to read in instead of computing the weights on the fly for the fill operation. If this is set, fillalgo and fillmask are ignored. default='NOT_SET'
fillwrite	array of fill mapping file to write default='NOT_SET'
mapalgo	array of spatial interpolation algorithms default="bilinear"
mapmask	array of spatial interpolation mask default='NOT_SET'
mapread	array of spatial interpolation mapping files to read (optional) default='NOT_SET'
mapwrite	array (up to 30 elements) of spatial interpolation mapping files to write (optional). Specifies the weights file to generate after weights are computed on the fly for the mapping (interpolation) operation, thereby allowing users to save and reuse a set of weights later. default='NOT_SET'
tintalgo	array (up to 30 elements) of time interpolation algorithm options associated with the array of streams. valid values: lower,upper,nearest,linear,coszen lower = Use lower time-value upper = Use upper time-value nearest = Use the nearest time-value linear = Linearly interpolate between the two time-values coszen = Scale according to the cosine of the solar zenith angle (for solar) default="linear"
taxMode	array (up to 30 elements) of time interpolation modes. Time axis interpolation modes are associated with the array of streams for handling data outside the specified stream time axis. Valid options are to cycle the data based on the first, last, and align settings associated with the stream dataset, to extend the first and last valid value indefinitely, or to limit the interpolated data to fall only between the least and greatest valid value of the time array. valid values: cycle,extend,limit extend = extrapolate before and after the period by using the first or last value. cycle = cycle between the range of data limit = restrict to the period for which the data is valid default="cycle"

dtlimit	array (up to 30 elements) of setting delta time axis limit. Specifies delta time ratio limits placed on the time interpolation associated with the array of streams. Causes the model to stop if the ratio of the running maximum delta time divided by the minimum delta time is greater than the dtlimit for that stream. For instance, with daily data, the delta time should be exactly one day throughout the dataset and the computed maximum divided by minimum delta time should always be 1.0. For monthly data, the delta time should be between 28 and 31 days and the maximum ratio should be about 1.1. The running value of the delta time is computed as data is read and any wraparound or cycling is also included. this input helps trap missing data or errors in cycling. to turn off trapping, set the value to 1.0e30 or something similar. default=1.5
vectors	paired vector field names

`shr_strdata_nml` contains a namelist variable, `streams`, that specifies a list of input stream description files and for each file what years of data to use, and how to align the input stream time axis with the model run time axis.

The general input format for the `streams` namelist variable is:

```
&shr_strdata_nml
  streams = 'stream1.txt year_align year_first year_last ',
            'stream2.txt year_align year_first year_last ',
             ...
            'streamN.txt year_align year_first year_last '
/
```

where:

```
streamN.txt
   the stream description file, a plain text file containing details about the input stream
year_first
   the first year of data that will be used
year_last
   the last year of data that will be used
year_align
   a model year that will be aligned with data for year_first
```

Customizing shr_strdata_nml values

The contents of `shr_strdata_nml` are automatically generated by that data model's **cime_config/buildnml** script. These contents are easily customizable for your target experiment. As an example we refer to the following ``datm_in contents (that would appear in both `$CASEROOT/CaseDocs` and `$RUNDIR`):

```
\&shr_strdata_nml
   datamode   = 'CLMNCEP'
   domainfile = '/glade/proj3/cseg/inputdata/share/domains/domain.lnd.fv1.9x2.5_gx1v6.090206
   dtlimit    = 1.5,1.5,1.5,1.5
   fillalgo   = 'nn','nn','nn','nn'
   fillmask   = 'nomask','nomask','nomask','nomask'
   mapalgo    = 'bilinear','bilinear','bilinear','bilinear'
   mapmask    = 'nomask','nomask','nomask','nomask'
   streams    = "datm.streams.txt.CLM_QIAN.Solar  1895 1948 1972  ",
                "datm.streams.txt.CLM_QIAN.Precip 1895 1948 1972  ",
                "datm.streams.txt.CLM_QIAN.TPQW   1895 1948 1972  ",
                "datm.streams.txt.presaero.trans_1850-2000 1849 1849 2006"
   taxmode    = 'cycle','cycle','cycle','cycle'
   tintalgo   = 'coszen','nearest','linear','linear'
```

```
   vectors    = 'null'
/
```

As is discussed in the CIME User's Guide, to change the contents of `datm_in`, you must edit `$CASEROOT/user_nl_datm`. In the above example, you can to this to change any of the above settings **except for the names**

::

datm.streams.txt.CLM_QIAN.Solar datm.streams.txt.CLM_QIAN.Precip datm.streams.txt.CLM_QIAN.TPQW datm.streams.txt.presaero.trans_1850-2000

Other than these names, any namelist variable from `shr_strdata_nml` can be modified by adding the appropriate keyword/value pairs to `user_nl_datm`.

As an example, the following could be the contents of `$CASEROOT/user_nl_datm`:

```
!-----------------------------------------------------------------------
! Users should ONLY USE user_nl_datm to change namelists variables
! Users should add all user specific namelist changes below in the form of
! namelist_var = new_namelist_value
! Note that any namelist variable from shr_strdata_nml and datm_nml can
! be modified below using the above syntax
! User preview_namelists to view (not modify) the output namelist in the
! directory $CASEROOT/CaseDocs
! To modify the contents of a stream txt file, first use preview_namelists
! to obtain the contents of the stream txt files in CaseDocs, and then
! place a copy of the  modified stream txt file in $CASEROOT with the string
! user_ prepended.
!-----------------------------------------------------------------------
streams    = "datm.streams.txt.CLM_QIAN.Solar  1895 1948 1900  ",
             "datm.streams.txt.CLM_QIAN.Precip 1895 1948 1900  ",
             "datm.streams.txt.CLM_QIAN.TPQW   1895 1948 1900  ",
             "datm.streams.txt.presaero.trans_1850-2000 1849 1849 2006"
```

and the contents of `shr_strdata_nml` (in both `$CASEROOT/CaseDocs` and `$RUNDIR`) would be

```
datamode   = 'CLMNCEP'
domainfile = '/glade/proj3/cseg/inputdata/share/domains/domain.lnd.fv1.9x2.5_gx1v6.090206.nc
dtlimit    = 1.5,1.5,1.5,1.5
fillalgo   = 'nn','nn','nn','nn'
fillmask   = 'nomask','nomask','nomask','nomask'
mapalgo    = 'bilinear','bilinear','bilinear','bilinear'
mapmask    = 'nomask','nomask','nomask','nomask'
streams    = "datm.streams.txt.CLM_QIAN.Solar  1895 1948 1900  ",
             "datm.streams.txt.CLM_QIAN.Precip 1895 1948 1900  ",
             "datm.streams.txt.CLM_QIAN.TPQW   1895 1948 1900  ",
             "datm.streams.txt.presaero.trans_1850-2000 1849 1849 2006"
taxmode    = 'cycle','cycle','cycle','cycle'
tintalgo   = 'coszen','nearest','linear','linear'
vectors    = 'null'
```

As is discussed in the CIME User's Guide, you should use **preview_namelists** to view (not modify) the output namelist in `CaseDocs`.

Stream Description File

The *stream description file* is not a Fortran namelist, but a locally built xml-like parsing implementation. Sometimes it is called a "stream dot-text file" because it has a ".txt." in the filename. Stream description files contain data that specifies the names of the fields in the stream, the names of the input data files, and the file system directory where the data files are located.

The data elements found in the stream description file are:

`dataSource`

A comment about the source of the data -- always set to GENERIC and is there only for backwards compatibility.

`domainInfo`

Information about the domain data for this stream specified by the following 3 sub elements.

`variableNames`

A list of the domain variable names. This is a paired list with the name of the variable in the netCDF file on the left and the name of the corresponding model variable on the right. This data models require five variables in this list. The names of model's variables (names on the right) must be: "time," "lon," "lat," "area," and "mask."

`filePath`

The file system directory where the domain data file is located.

`fileNames`

The name of the domain data file. Often the domain data is located in the same file as the field data (above), in which case the name of the domain file could simply be the name of the first field data file. Sometimes the field data files don't contain the domain data required by the data models, in this case, one new file can be created that contains the required data.

`fieldInfo`

Information about the stream data for this stream specified by the following 3 required sub elements and optional offset element.

`variableNames`

A list of the field variable names. This is a paired list with the name of the variable in the netCDF file on the left and the name of the corresponding model variable on the right. This is the list of fields to read in from the data file, there may be other fields in the file which are not read in (ie. they won't be used).

`filePath`

The file system directory where the data files are located.

`fileNames`

The list of data files to use. If there is more than one file, the files must be in chronological order, that is, the dates in time axis of the first file are before the dates in the time axis of the second file.

`offset`

The offset allows a user to shift the time axis of a data stream by a fixed and constant number of seconds. For instance, if a data set contains daily average data with timestamps for the data at the end of the day, it might be appropriate to shift the time axis by 12 hours so the data is taken to be at the middle of the day instead of the end of the day. This feature supports only simple shifts in seconds as a way of correcting input data time axes without having to modify the input data time axis manually. This feature does not support more complex shifts such as end of month to mid-month. But in conjunction with the time interpolation methods in the strdata input, hopefully most user needs can be accommodated with the two settings. Note that a positive offset advances the input data time axis forward by that number of seconds.

The data models advance in time discretely. At a given time, they read/derive fields from input files. Those input files have data on a discrete time axis as well. Each data point in the input files are associated with a discrete time (as opposed to a time interval). Depending whether you pick lower, upper, nearest, linear, or coszen; the data in the input file will be "interpolated" to the time in the model.

The offset shifts the time axis of the input data the given number of seconds. So if the input data is at 0, 3600, 7200, 10800 seconds (hourly) and you set an offset of 1800, then the input data will be set at times 1800, 5400, 9000, and 12600. So a model at time 3600 using linear interpolation would have data at "n=2" with offset of 0 will have data at "n=(2+3)/2" with an offset of 1800. n=2 is the 2nd data in the time list 0, 3600, 7200, 10800 in this example. n=(2+3)/2 is the average of the 2nd and 3rd data in the time list 0, 3600, 7200, 10800. offset can be positive or negative.

Actual example:

```
<stream>
<domainInfo>
   <variableNames>
      time    time
```

```
      lon    lon
      lat    lat
      area   area
      mask   mask
   </variableNames>
   <filePath>
     /glade/proj3/cseg/inputdata/atm/datm7/NYF
   </filePath>
   <fileNames>
     nyf.ncep.T62.050923.nc
   </fileNames>
</domainInfo>
 <fieldInfo>
   <variableNames>
      dn10  dens
      slp_  pslv
      q10   shnum
      t_10  tbot
      u_10  u
      v_10  v
   </variableNames>
   <filePath>
      /glade/proj3/cseg/inputdata/atm/datm7/NYF
   </filePath>
   <offset>
      0
   </offset>
   <fileNames>
      nyf.ncep.T62.050923.nc
   </fileNames>
 </fieldInfo>
</stream>
```

Customizing stream description files

Each data model's **cime-config/buildnml** utility automatically generates the required stream description files for the case. The directory contents of each data model will look like the following (using DATM as an example)

```
$CIMEROOT/components/data_comps/datm/cime_config/buildnml
$CIMEROOT/components/data_comps/datm/cime_config/namelist_definition_datm.xml
```

The `namelist_definition_datm.xml` file defines and sets default values for all the namelist variables and associated groups and also provides out-of-the box settings for the target data model and target stream. **buildnml** utilizes this two files to construct the stream files for the given compset settings. You can modify the generated stream files for your particular needs by doing the following:

1. Copy the relevant description file from `$CASEROOT/CaseDocs` to `$CASEROOT` and pre-pend a "user_"string to the filename. Change the permission of the file to write. For example, assuming you are in **$CASEROOT**

   ```
   cp $CASEROOT/CaseDocs/datm.streams.txt.CLM_QIAN.Solar  $CASEROOT/user_datm.streams.txt.C
   chmod u+w $CASEROOT/user_datm.streams.txt.CLM_QIAN.Solar
   ```

2.
   - Edit `$CASEROOT/user_datm.streams.txt.CLM_QIAN.Solar` with your desired changes.
   - *Be sure not to put any tab characters in the file: use spaces instead*.
   - In contrast to other user_nl_xxx files, be sure to set all relevant data model settings in the xml files, issue the **preview_namelist** command and THEN edit the `user_datm.streams.txt.CLM_QIAN.Solar` file.
   - **Once you have created a user_xxx.streams.txt.* file, further modifications to the relevant data model settings in the xml files will be ignored.**
   - If you later realize that you need to change some settings in an xml file, you should remove the user_xxx.streams.txt.* file(s), make the modifications in the xml file, rerun **preview_namelists**, and then reintroduce your modifications into a new user_xxx.streams.txt.* stream file(s).
3. Call **preview_namelists** and verify that your changes do indeed appear in the resultant stream description file appear in `CaseDocs/datm.streams.txt.CLM_QIAN.Solar`. These changes will also appear in `$RUNDIR/datm.streams.txt.CLM_QIAN.Solar`.

Design Details

The data model functionality is executed via set of specific operations associated with reading and interpolating data in space and time. The strdata implementation does the following:

1. determines nearest lower and upper bound data from the input dataset
2. if that is new data then read lower and upper bound data
3. fill lower and upper bound data
4. spatially map lower and upper bound data to model grid
5. time interpolate lower and upper bound data to model time
6. return fields to data model

IO Through Data Models

Namlist variables referenced below are discussed in detail in stream data namelist section.

The two timestamps of input data that bracket the present model time are read first. These are called the lower and upper bounds of data and will change as the model advances. Those two sets of inputdata are first filled based on the user setting of the namelist variables `str_fillalgo` and `str_fillmask`. That operation occurs on the input data grid. The lower and upper bound data are then spatially mapped to the model grid based upon the user setting of the namelist variables `str_mapalgo` and `str_mapmask`. Spatial interpolation only occurs if the input data grid and model grid are not the identical, and this is determined in the strdata module automatically. Time interpolation is the final step and is done using a time interpolation method specified by the user in namelist (via the shr_strdata_nml namelist variable "tintalgo"). A final set of fields is then available to the data model on the model grid and for the current model time.

There are two primary costs associated with strdata, reading data and spatially mapping data. Time interpolation is relatively cheap in the current implementation. As much as possible, redundant operations are minimized. Fill and mapping weights are generated at initialization and saved. The upper and lower bound mapped input data is saved between time steps to reduce mapping costs in cases where data is time interpolated more often than new data is read. If the input data timestep is relatively small (for example, hourly data as opposed to daily or monthly data) the cost of reading input data can be quite large. Also, there can be significant variation in cost of the data model over the coarse of the run, for instance, when new inputdata must be read and interpolated, although it's relatively predictable. The present implementation doesn't support changing the order of operations, for instance, time interpolating the data before spatial mapping. Because the present computations are always linear, changing the order of operations will not fundamentally change the results. The present order of operations generally minimizes the mapping cost for typical data model use cases.

There are several limitations in both options and usage within the data models at the present time. Spatial interpolation can only be performed from a two-dimensional latitude-longitude input grid. The target grid can be arbitrary but the source grid must be able to be described by simple one-dimensional lists of longitudes and latitudes, although they don't have to have equally spaced.

At the present time, data models can only read netcdf data, and IO is handled through either standard netcdf interfaces or through the PIO library using either netcdf or pnetcdf. If standard netcdf is used, global fields are read

and then scattered one field at a time. If PIO is used, then data will be read either serially or in parallel in chunks that are approximately the global field size divided by the number of io tasks. If pnetcdf is used through PIO, then the pnetcdf library must be included during the build of the model. The pnetcdf path and option is hardwired into the `Macros.make` file for the specific machine. To turn on `pnetcdf` in the build, make sure the `Macros.make` variables PNETCDF_PATH, INC_PNETCDF, and LIB_PNETCDF are set and that the PIO CONFIG_ARGS sets the PNETCDF_PATH argument.

Beyond just the option of selecting IO with PIO, several namelist are available to help optimize PIO IO performance. Those are **TODO** - list these. The total mpi tasks that can be used for IO is limited to the total number of tasks used by the data model. Often though, fewer io tasks result in improved performance. In general, [io_root + (num_iotasks-1)*io_stride + 1] has to be less than the total number of data model tasks. In practice, PIO seems to perform optimally somewhere between the extremes of 1 task and all tasks, and is highly machine and problem dependent. .. _restart-files:

Restart Files

Restart files are generated automatically by the data models based upon a flag sent from the driver. The restart files must meet the CIME naming convention and an `rpointer` file is generated at the same time. An `rpointer` file is a *restart pointer* file which contains the name of the most recently created restart file. Normally, if restart files are read, the restart filenames are specified in the rpointer file. Optionally though, there are namelist variables such as *restfilm*` to specify the restart filenames via namelist. If those namelist are set, the `rpointer` file will be ignored. The default method is to use the `rpointer` files to specify the restart filenames. In most cases, no model restart is required for the data models to restart exactly. This is because there is no memory between timesteps in many of the data model science modes. If a model restart is required, it will be written automatically and then must be used to continue the previous run.

There are separate stream restart files that only exist for performance reasons. A stream restart file contains information about the time axis of the input streams. This information helps reduce the start costs associated with reading the input dataset time axis information. If a stream restart file is missing, the code will restart without it but may need to reread data from the input data files that would have been stored in the stream restart file. This will take extra time but will not impact the results.

Data Structures

The data models all use three fundamental routines.

- $CIMEROOT/src/utils/shr_dmodel_mod.F90
- $CIMEROOT/src/utils/shr_stream_mod.F90
- $CIMEROOT/src/utils/shr_strdata.F90

These routines contain three data structures that are leveraged by all the data model code.

The most basic type, `shr_stream_fileType` is contained in `shr_stream_mod.F90` and specifies basic information related to a given stream file.

```
type shr_stream_fileType
   character(SHR_KIND_CL) :: name = shr_stream_file_null     ! the file name
   logical                :: haveData = .false.              ! has t-coord data been read in
   integer  (SHR_KIND_IN) :: nt = 0                          ! size of time dimension
   integer  (SHR_KIND_IN),allocatable :: date(:)             ! t-coord date: yyyymmdd
   integer  (SHR_KIND_IN),allocatable :: secs(:)             ! t-coord secs: elapsed on date
end type shr_stream_fileType
```

The following type, `shr_stream_streamType` contains information that encapsulates the information related to all files specific to a target stream. These are the list of files found in the `domainInfo` and `fieldInfo` blocks of the target stream description file (see the overview of the Stream Description File).

```
type shr_stream_streamType
   !private                                  ! no public access to internal components
   !--- input data file names and data ---
   logical                  :: init          ! has stream been initialized?
   integer  (SHR_KIND_IN),pointer :: initarr(:) => null()! surrogate for init flag
   integer  (SHR_KIND_IN)    :: nFiles       ! number of data files
```

```
   character(SHR_KIND_CS)    :: dataSource     ! meta data identifying data source
   character(SHR_KIND_CL)    :: filePath       ! remote location of data files
   type(shr_stream_fileType), allocatable :: file(:) ! data specific to each file

   !--- specifies how model dates align with data dates ---
   integer(SHR_KIND_IN)      :: yearFirst      ! first year to use in t-axis (yyyymmdd)
   integer(SHR_KIND_IN)      :: yearLast       ! last  year to use in t-axis (yyyymmdd)
   integer(SHR_KIND_IN)      :: yearAlign      ! align yearFirst with this model year
   integer(SHR_KIND_IN)      :: offset         ! offset in seconds of stream data
   character(SHR_KIND_CS)    :: taxMode        ! cycling option for time axis

   !--- useful for quicker searching ---
   integer(SHR_KIND_IN) :: k_lvd,n_lvd         ! file/sample of least valid date
   logical              :: found_lvd           ! T <=> k_lvd,n_lvd have been set
   integer(SHR_KIND_IN) :: k_gvd,n_gvd         ! file/sample of greatest valid date
   logical              :: found_gvd           ! T <=> k_gvd,n_gvd have been set

   !---- for keeping files open
   logical                :: fileopen          ! is current file open
   character(SHR_KIND_CL) :: currfile          ! current filename
   type(file_desc_t)      :: currpioid         ! current pio file desc

   !--- stream data not used by stream module itself ---
   character(SHR_KIND_CXX):: fldListFile       ! field list: file's  field names
   character(SHR_KIND_CXX):: fldListModel      ! field list: model's field names
   character(SHR_KIND_CL) :: domFilePath       ! domain file: file path of domain file
   character(SHR_KIND_CL) :: domFileName       ! domain file: name
   character(SHR_KIND_CS) :: domTvarName       ! domain file: time-dim var name
   character(SHR_KIND_CS) :: domXvarName       ! domain file: x-dim var name
   character(SHR_KIND_CS) :: domYvarName       ! domain file: y-dim var name
   character(SHR_KIND_CS) :: domZvarName       ! domain file: z-dim var name
   character(SHR_KIND_CS) :: domAreaName       ! domain file: area  var name
   character(SHR_KIND_CS) :: domMaskName       ! domain file: mask  var name

   character(SHR_KIND_CS) :: tInterpAlgo       ! Algorithm to use for time interpolation
   character(SHR_KIND_CL) :: calendar          ! stream calendar
end type shr_stream_streamType
```

and finally, the `shr_strdata_type` is the heart of the CIME data model implemenentation and contains information for all the streams that are active for the target data model. The first part of the shr_strdata_type is filled in by the namelist values read in from the namelist group (see the stream data namelist section).

```
 type shr_strdata_type
   ! --- set by input namelist ---
  character(CL)  :: dataMode          ! flags physics options wrt input data
  character(CL)  :: domainFile        ! file   containing domain info
  character(CL)  :: streams (nStrMax) ! stream description file names
  character(CL)  :: taxMode (nStrMax) ! time axis cycling mode
  real(R8)       :: dtlimit (nStrMax) ! dt max/min limit
  character(CL)  :: vectors (nVecMax) ! define vectors to vector map
  character(CL)  :: fillalgo(nStrMax) ! fill algorithm
  character(CL)  :: fillmask(nStrMax) ! fill mask
  character(CL)  :: fillread(nStrMax) ! fill mapping file to read
  character(CL)  :: fillwrit(nStrMax) ! fill mapping file to write
  character(CL)  :: mapalgo (nStrMax) ! scalar map algorithm
  character(CL)  :: mapmask (nStrMax) ! scalar map mask
  character(CL)  :: mapread (nStrMax) ! regrid mapping file to read
  character(CL)  :: mapwrit (nStrMax) ! regrid mapping file to write
  character(CL)  :: tintalgo(nStrMax) ! time interpolation algorithm
  integer(IN)    :: io_type           ! io type, currently pnetcdf or netcdf
```

```
  !--- data required by cosz t-interp method, ---
  real(R8)     :: eccen   ! orbital eccentricity
  real(R8)     :: mvelpp  ! moving vernal equinox long
  real(R8)     :: lambm0  ! mean long of perihelion at vernal equinox (radians)
  real(R8)     :: obliqr  ! obliquity in degrees
  integer(IN)  :: modeldt ! data model dt in seconds (set to the coupling frequency)

  ! --- data model grid, public ---
  integer(IN)     :: nxg          ! data model grid lon size
  integer(IN)     :: nyg          ! data model grid lat size
  integer(IN)     :: nzg          ! data model grid vertical size
  integer(IN)     :: lsize        ! data model grid local size
  type(mct_gsmap) :: gsmap        ! data model grid global seg map
  type(mct_ggrid) :: grid         ! data model grid ggrid
  type(mct_avect) :: avs(nStrMax) ! data model stream attribute vectors

  ! --- stream specific arrays, stream grid ---
  type(shr_stream_streamType)    :: stream(nStrMax)
  type(iosystem_desc_t), pointer :: pio_subsystem => null()
  type(io_desc_t)    :: pio_iodesc(nStrMax)
  integer(IN)        :: nstreams          ! actual number of streams
  integer(IN)        :: strnxg(nStrMax)   ! stream grid lon sizes
  integer(IN)        :: strnyg(nStrMax)   ! stream grid lat sizes
  integer(IN)        :: strnzg(nStrMax)   ! tream grid global sizes
  logical            :: dofill(nStrMax)   ! true if stream grid is different from data model
  logical            :: domaps(nStrMax)   ! true if stream grid is different from data model
  integer(IN)        :: lsizeR(nStrMax)   ! stream local size of gsmapR on processor
  type(mct_gsmap)    :: gsmapR(nStrMax)   ! stream global seg map
  type(mct_rearr)    :: rearrR(nStrMax)   ! rearranger
  type(mct_ggrid)    :: gridR(nStrMax)    ! local stream grid on processor
  type(mct_avect)    :: avRLB(nStrMax)    ! Read attrvect
  type(mct_avect)    :: avRUB(nStrMax)    ! Read attrvect
  type(mct_avect)    :: avFUB(nStrMax)    ! Final attrvect
  type(mct_avect)    :: avFLB(nStrMax)    ! Final attrvect
  type(mct_avect)    :: avCoszen(nStrMax) ! data assocaited with coszen time interp
  type(mct_sMatP)    :: sMatPf(nStrMax)   ! sparse matrix map for fill on stream grid
  type(mct_sMatP)    :: sMatPs(nStrMax)   ! sparse matrix map for mapping from stream to dat
  integer(IN)        :: ymdLB(nStrMax)    ! lower bound time for stream
  integer(IN)        :: todLB(nStrMax)    ! lower bound time for stream
  integer(IN)        :: ymdUB(nStrMax)    ! upper bound time for stream
  integer(IN)        :: todUB(nStrMax)    ! upper bound time for stream
  real(R8)           :: dtmin(nStrMax)
  real(R8)           :: dtmax(nStrMax)

  ! --- internal ---
  integer(IN)        :: ymd  ,tod
  character(CL)      :: calendar          ! model calendar for ymd,tod
  integer(IN)        :: nvectors          ! number of vectors
  integer(IN)        :: ustrm (nVecMax)
  integer(IN)        :: vstrm (nVecMax)
  character(CL)      :: allocstring
end type shr_strdata_type
```

Data Model Science

When a given data models run, the user must specify which *science mode* it will run in. Each data model has a fixed set of fields that it must send to the coupler, but it is the choice of mode that specifies how that set of fields is to be computed. Each mode activates various assumptions about what input fields are available from the input data streams, what input fields are available from the the coupler, and how to use this input data to compute the output fields sent to the coupler.

In general, a mode might specify...

- that fields be set to a time invariant constant (so that no input data is needed)
- that fields be taken directly from a input data files (the input streams)
- that fields be computed using data read in from input files
- that fields be computed using from data received from the coupler
- some combination of the above.

If a science mode is chosen that is not consistent with the input data provided, the model may abort (perhaps with a "missing data" error message), or the model may send erroneous data to the coupler (for example, if a mode assumes an input stream has temperature in Kelvin on it, but it really has temperature in Celsius). Such an error is unlikely unless a user has edited the run scripts to specify either non-standard input data or a non-standard science mode. When editing the run scripts to use non-standard stream data or modes, users must be careful that the input data is consistent with the science mode and should verify that the data model is providing data to the coupler as expected.

The data model mode is a character string that is set in the namelist variable `datamode` in the namelist group `shr_strdata_nml`. Although each data model, `datm`, `dlnd`, `drof`, `docn`, `dice` and `dwav` has its own set of valid datamode values, two modes are common to all data models: `COPYALL` and `NULL`.

`dataMode = "COPYALL"`

The default mode is COPYALL -- the model will assume *all* the data that must be sent to the coupler will be found in the input data streams, and that this data can be sent to the coupler, unaltered, except for spatial and temporal interpolation.

`dataMode = "NULL"`

NULL mode turns off the data model as a provider of data to the coupler. The model_present flag (eg. atm_present) will be set to false and the coupler will assume no exchange of data to or from the data model.

Data Atmosphere (DATM)

DATM is normally used to provide observational forcing data (or forcing data produced by a previous run using active components) to drive prognostic components. In the case of CESM, these would be: CLM (I compset), POP2 (C compset), and POP2/CICE (G compset). As a result, DATM variable settings are specific to the compset that will be targeted. As examples, CORE2_NYF (CORE2 normal year forcing) is the DATM mode used in C and G compsets. CLM_QIAN, CLMCRUNCEP, CLMGSWP3 and CLM1PT are DATM modes using observational data for forcing CLM in I compsets.

xml variables

The following are `$CASEROOT` xml variables that CIME supports for DATM. These variables are defined in `$CIMEROOT/src/components/data_comps/datm/cime_config/config_component.xml`. These variables will appear in `env_run.xml` and the resulting values are compset dependent.

> ***Note***
>
> These xml variables are used by the the datm's **cime_config/buildnml** script in conjunction with datm's **cime_config/namelist_definition_datm.xml** file to generate the namelist file `datm_in`.

"DATM xml variables"

xml variable	description
DATM_MODE	Mode for atmospheric component
	Valid values are: CORE2_NYF,CORE2_IAF,CLM_QIAN,CLM_QIAN_WISO,CLM1PT,CLMCRUNCEP,
	CLMCRUNCEP_V5,CLMGSWP3,WW3,CPLHISTForcing

DATM_PRESAERO	Prescribed aerosol forcing, if any
DATM_TOPO	Surface topography
DATM_CO2_TSERIES	CO2 time series type
DATM_CPLHIST_CASE	Coupler history data mode case name
DATM_CPLHIST_DIR	Coupler history data mode directory containing coupler history data
DATM_CPLHIST_YR_ALIGN	Coupler history data model simulation year corresponding to data starting year
DATM_CPLHIST_YR_START	Coupler history data model starting year to loop data over
DATM_CPLHIST_YR_END	Coupler history data model ending year to loop data over
DATM_CLMNCEP_YR_ALIGN	I compsets only - simulation year corresponding to data starting year
DATM_CPLHIST_YR_START	I compsets only - data model starting year to loop data over
DATM_CPLHIST_YR_END	I compsets only - data model ending year to loop data over

datamode values

The xml variable `DATM_MODE` sets the streams that are associated with DATM and also sets the namelist variable `datamode` that specifies what additional operations need to be done by DATM on the streams before returning to the driver. One of the variables in `shr_strdata_nml` is `datamode`, whose value is a character string. Each data model has a unique set of `datamode` values that it supports. The valid values for `datamode` are set in the file `namelist_definition_datm.xml` using the xml variable `DATM_MODE` in the `config_component.xml` file for DATM. CIME will generate a value `datamode` that is compset dependent.

The following are the supported DATM datamode values and their relationship to the `$DATM_MODE` xml variable value.

"Valid values for datamode namelist variable"

datamode variable	description
NULL	This mode turns off the data model as a provider of data to the coupler. The `atm_present` flag will be set to `false` and the coupler assumes no exchange of data to or from the data model.
COPYALL	The default science mode of the data model is the COPYALL mode. This mode will examine the fields found in all input data streams, if any input field names match the field names used internally, they are copied into the export array and passed directly to the coupler without any special user code. Any required fields not found on an input stream will be set to zero except for aerosol deposition fields which will be set to a special value.
CPLHIST	Uilize user-generated coupler history data to spin up prognostic component. Works exactly like COPYALL.
CLMNCEP	In conjunction with NCEP climatological atmosphere data, provides the atmosphere forcing favored by the Land Model Working Group when coupling an active land model with observed atmospheric forcing. This mode replicates code previously found in CLM (circa 2005), before the LMWG started using the CCSM flux coupler and data models to do active-land-only simulations.
CORE2_NYF	Coordinated Ocean-ice Reference Experiments (CORE) Version 2 Normal Year Forcing.

CORE2_IAF	In conjunction with with CORE Version 2 atmospheric forcing data, provides the atmosphere forcing favored by the Ocean Model Working Group when coupling an active ocean model with observed atmospheric forcing. This mode and associated data sets implement the CORE-IAF Version 2 forcing data, as developed by Large and Yeager (2008) at NCAR. Note that CORE2_NYF and CORE2_IAF work exactly the same way.

DATM_MODE, datamode and streams

The following tabe describes the valid values of `DATM_MODE`, and how it relates to the associated input streams and the `datamode` namelist variable.

"Relationship between DATM_MODE, datamode and streams"

DATM_MODE	description-streams-datamode
NULL	null mode
	streams: none
	datamode: NULL
CORE2_NYF	CORE2 normal year forcing (C ang G compsets)
	streams: CORE2_NYF.GISS,CORE2_NYF.GXGXS,CORE2_NYF.NCEP
	datamode: CORE2_NYF
CORE2_IAF	CORE2 interannual year forcing (C ang G compsets)
	streams: CORE2_IAF.GCGCS.PREC,CORE2_IAF.GISS.LWDN,CORE2_IAF.GISS.SWDN,CORE2_IAF.GISS.SWUP,
	CORE2_IAF.NCEP.DN10,CORE2_IAF.NCEP.Q_10,CORE2_IAF.NCEP.SLP_,CORE2_IAF.NCEP.T_10,CORE2_IAF.NCEP.U_10,
	CORE2_IAF.NCEP.V_10,CORE2_IAF.CORE2.ArcFactor
	datamode: CORE2_IAF
CLM_QIAN_WISO	QIAN atm input data with water isotopes (I compsets)
	streams: CLM_QIAN_WISO.Solar,CLM_QIAN_WISO.Precip,CLM_QIAN_WISO.TPQW
	datamode: CLMNCEP
CLM_QIAN	QIAN atm input data (I compsets)
	streams: CLM_QIAN.Solar,CLM_QIAN.Precip,CLM_QIAN.TPQW
	datamode: CLMNCEP
CLMCRUNCEP	CRUNCEP atm input data (I compsets)
	streams: CLMCRUNCEP.Solar,CLMCRUNCEP.Precip,CLMCRUNCEP.TPQW
	datamode: CLMNCEP
CLMCRUNCEP_V5	CRUNCEP atm input data (I compsets)
	streams: CLMCRUNCEP_V5.Solar,CLMCRUNCEP_V5.Precip,CLMCRUNCEP_V5.TPQW
	datamode: CLMNCEP
CLMGSWP3	GSWP3 atm input data (I compsets)
	streams: CLMGSWP3.Solar,CLMGSWP3.Precip,CLMGSWP3.TPQW
	datamode: CLMNCEP
CLM1PT	single point tower site atm input data

	streams: CLM1PT.$ATM_GRID
	datamode: CLMNCEP
CPLHISTForcing	user generated forcing data to spinup for I and G compsets
	streams: CPLHISTForcingForOcnIce.Solar,CPLHISTForcingForOcnIce.nonSolarFlux,
	CPLHISTForcingForOcnIce.State3hr,CPLHISTForcingForOcnIce.State1hr
	datamode: CPLHIST
WW3	WW3 wave watch data from a short period of hi WW3 wave watch data from a short period of hi temporal frequency COREv2 data
	streams: WW3
	datamode: COPYALL

Namelists

The DATM namelist file is `datm_in` (or `datm_in_NNN` for multiple instances). DATM namelists can be separated into two groups: *stream-independent* namelist variables that are specific to the DATM model and *stream-specific* namelist variables whose names are common to all the data models.

Stream dependent input is in the namelist group `"shr_strdata_nml` which is discussed in input streams and is the same for all data models.

The stream-independent group is `datm_nml` and the DATM stream-independent namelist variables are:

datm_nml vars	description
decomp	decomposition strategy (1d, root) 1d => vector decomposition, root => run on master task
restfilm	master restart filename
restfils	stream restart filename
force_prognostic_true	TRUE => force prognostic behavior
bias_correct	if set, include bias correction streams in namelist
anomaly_forcing	if set, includ anomaly forcing streams in namelist
factorfn	filename containing correction factors for use in CORE2 modes (CORE2_IAF and CORE2_NYF)
presaero	if true, prescribed aerosols are sent from datm
iradsw	frequency to update radiation in number of time steps (of hours if negative)
wiso_datm	if true, turn on water isotopes

Streams independent of DATM_MODE value

In general, each `DATM_MODE` xml variable is identified with a unique set of streams. However, there are several streams in DATM that can accompany any `DATM_MODE` setting. Currently, these are streams associated with prescribed aerosols, co2 time series, topography, anomoly forcing and bias correction. These mode-independent streams are activated different, depending on the stream.

- `prescribed aerosol stream:` To add this stream, set `$DATM_PRESAERO` to a supported value other than `none`.
- `co2 time series stream`: To add this stream, set `$DATM_CO2_TSERIES` to a supported value other than `none`.
- `topo stream`: To add this stream, set `$DATM_TOPO` to a supported value other than `none`.

- `anomaly forcing stream:` To add this stream, you need to add any of the following keywword/value pair to the end of `user_nl_datm`:

```
Anomaly.Forcing.Precip = <filename>
Anomaly.Forcing.Temperature = <filename>
Anomaly.Forcing.Pressure = <filename>
Anomaly.Forcing.Humidity = <filename>
Anomaly.Forcing.Uwind = <filename>
Anomaly.Forcing.Vwind = <filename>
Anomaly.Forcing.Shortwave = <filename>
Anomaly.Forcing.Longwave = <filename>
```

- `bias_correct stream:` To add this stream, you need to add any of the following keywword/value pair to the end of `user_nl_datm`:

```
BC.QIAN.CMAP.Precip = <filename>
BC.QIAN.GPCP.Precip = <filename>
BC.CRUNCEP.CMAP.Precip = <filename>
BC.CRUNCEP.GPCP.Precip = <filename>
```

DATM Field names

DATM defines a set of pre-defined internal field names as well as mappings for how those field names map to the fields sent to the coupler. In general, the stream input file should translate the stream input variable names into the `datm_fld` names for use within the data atmosphere model.

"DATM internal field names"

datm_fld (avifld)	driver_fld (avofld)
z	Sa_z
topo	Sa_topo
u	Sa_u
v	Sa_v
tbot	Sa_tbot
ptem	Sa_ptem
shum	Sa_shum
dens	Sa_dens
pbot	Sa_pbot
pslv	Sa_pslv
lwdn	Faxa_lwdn
rainc	Faxa_rainc
rainl	Faxa_rainl
snowc	Faxa_snowc
snowl	Faxa_snowl
swndr	Faxa_swndr
swvdr	Faxa_swvdr
swndf	Faxa_swndf
swvdf	Faxa_swvdf
swnet	Faxa_swnet
co2prog	Sa_co2prog

co2diag	Sa_co2diag
bcphidry	Faxa_bcphidry
bcphodry	Faxa_bcphodry
bcphiwet	Faxa_bcphiwet
ocphidry	Faxa_ocphidry
ocphodry	Faxa_ocphodry
ocphiwet	Faxa_ocphiwet
dstwet1	Faxa_dstwet1
dstwet2	Faxa_dstwet2
dstwet3	Faxa_dstwet3
dstwet4	Faxa_dstwet4
dstdry1	Faxa_dstdry1
dstdry2	Faxa_dstdry2
dstdry3	Faxa_dstdry3
dstdry4	Faxa_dstdry4
tref	Sx_tref
qref	Sx_qref
avsdr	Sx_avsdr
anidr	Sx_anidr
avsdf	Sx_avsdf
anidf	Sx_anidf
ts	Sx_t
to	So_t
snowhl	Sl_snowh
lfrac	Sf_lfrac
ifrac	Sf_ifrac
ofrac	Sf_ofrac
taux	Faxx_taux
tauy	Faxx_tauy
lat	Faxx_lat
sen	Faxx_sen
lwup	Faxx_lwup
evap	Faxx_evap
co2lnd	Fall_fco2_lnd
co2ocn	Faoo_fco2_ocn
dms	Faoo_fdms_ocn
precsf	Sa_precsf
prec_af	Sa_prec_af
u_af	Sa_u_af
v_af	Sa_v_af

tbot_af	Sa_tbot_af
pbot_af	Sa_pbot_af
shum_af	Sa_shum_af
swdn_af	Sa_swdn_af
lwdn_af	Sa_lwdn_af
rainc_18O	Faxa_rainc_18O
rainc_HDO	Faxa_rainc_HDO
rainl_18O	Faxa_rainl_18O
rainl_HDO	Faxa_rainl_HDO
snowc_18O	Faxa_snowc_18O
snowc_HDO	Faxa_snowc_HDO
snowl_18O	Faxa_snowl_18O
snowl_HDO	Faxa_snowl_HDO
shum_16O	Sa_shum_16O
shum_18O	Sa_shum_18O

Data Land (DLND)

The land model is unique because it supports land data and snow data (*lnd and sno*) almost as if they were two separate components, but they are in fact running in one component model through one interface. The lnd (land) data consist of fields sent to the atmosphere. This set of data is used when running DLND with an active atmosphere. In general this is not a mode that is used or supported. The sno (snow) data consist of fields sent to the glacier model. This set of data is used when running dlnd with an active glacier model (TG compsets). Both sets of data are assumed to be on the same grid.

xml variables

The following are xml variables that CIME supports for DLND. These variables are defined in `$CIMEROOT/src/components/data_comps/dlnd/cime_config/config_component.xml`. These variables will appear in `env_run.xml` and are used by the DLND `cime_config/buildnml` script to generate the DLND namelist file `dlnd_in` and the required associated stream files for the case.

Note

These xml variables are used by the the dlnd's **cime_config/buildnml** script in conjunction with dlnd's **cime_config/namelist_definition_dlnd.xml** file to generate the namelist file `dlnd_in`.

"DLND xml variables"

xml variable	description
DLND_MODE	Mode for data land component
	Valid values are: NULL, CPLHIST, GLC_CPLHIST
DLND_CPLHIST_CASE	Coupler history data mode case name
DLND_CPLHIST_DIR	Coupler history data mode directory containing coupler history data

DLND_CPLHIST_YR_ALIGN	Coupler history data model simulation year corresponding to data starting year
DLND_CPLHIST_YR_START	Coupler history data model starting year to loop data over
DLND_CPLHIST_YR_END	Coupler history data model ending year to loop data over

datamode values

The xml variable `DLND_MODE` sets the streams that are associated with DLND and also sets the namelist variable `datamode` that specifies what additional operations need to be done by DLND on the streams before returning to the driver. One of the variables in `shr_strdata_nml` is `datamode`, whose value is a character string. Each data model has a unique set of `datamode` values that it supports. The valid values for `datamode` are set in the file `namelist_definition_dlnd.xml` using the xml variable `DLND_MODE` in the `config_component.xml` file for DLND. CIME will generate a value `datamode` that is compset dependent.

The following are the supported DATM datamode values and their relationship to the `$DATM_MODE` xml variable value.

"Valid values for datamode namelist variable"

datamode variable	description
NULL	Turns off the data model as a provider of data to the coupler. The ice_present flag will be set to false and the coupler will assume no exchange of data to or from the data model.
COPYALL	The default science mode of the data model is the COPYALL mode. This mode will examine the fields found in all input data streams, if any input field names match the field names used internally, they are copied into the export array and passed directly to the coupler without any special user code. Any required fields not found on an input stream will be set to zero.

DLND_MODE, datamode and streams

The following tabe describes the valid values of `DLND_MODE`, and how it relates to the associated input streams and the `datamode` namelist variable.

"Relationship between DLND_MODE, datamode and streams"

DLND_MODE	description-streams-datamode
NULL	null mode
	streams: none
	datamode: null
CPLHIST	land forcing data (e.g. produced by CESM/CLM) from a previous model run is output in coupler history files and read in by the data land model.
	streams: lnd.cplhist
	COPYALL
GLC_CPLHIST	glc coupling fields (e.g. produced by CESM/CLM) from a previous model run are read in from a coupler history file.
	streams: glc.cplhist
	COPYALL

Namelists

The namelist file for DLND is `dlnd_in` (or `dlnd_in_NNN` for multiple instances).

As is the case for all data models, DLND namelists can be separated into two groups, stream-independent and stream-dependent.

The stream dependent group is shr_strdata_nml.

The stream-independent group is `dlnd_nml` and the DLND stream-independent namelist variables are:

decomp	decomposition strategy (1d, root) 1d => vector decomposition, root => run on master task
restfilm	master restart filename
restfils	stream restart filename
force_prognostic_true	TRUE => force prognostic behavior

To change the namelist settings in dlnd_in, edit the file user_nl_dlnd.

Streams independent of DLND_MODE value

There are no datamode independent streams for DLND.

Field names

DLND defines a set of pre-defined internal field names as well as mappings for how those field names map to the fields sent to the coupler. In general, the stream input file should translate the stream input variable names into the `dlnd_fld` names below for use within the data land model.

"DLND internal field names"

dlnd_fld (avifld)	driver_fld (avofld)
t	Sl_t
tref	Sl_tref
qref	Sl_qref
avsdr	Sl_avsdr
anidr	Sl_anidr
avsdf	Sl_avsdf
anidf	Sl_anidf
snowh	Sl_snowh
taux	Fall_taux
tauy	Fall_tauy
lat	Fall_lat
sen	Fall_sen
lwup	Fall_lwup
evap	Fall_evap
swnet	Fall_swnet
lfrac	Sl_landfrac
fv	Sl_fv
ram1	Sl_ram1
flddst1	Fall_flxdst1

flxdst2	Fall_flxdst2
flxdst3	Fall_flxdst3
flxdst4	Fall_flxdst4
tsrfNN	Sl_tsrf
topoNN	Sl_topo
qiceNN	Flgl_qice

where NN = (01,02,...,``nflds_snow * glc_nec)``, and `nflds_snow` is the number of snow fields in each elevation class and `glc_nec` is the number of elevation classes.

Data Ice (DICE)

DICE is a combination of a data model and a prognostic model. The data functionality reads in ice coverage. The prognostic functionality calculates the ice/atmosphere and ice/ocean fluxes. DICE receives the same atmospheric input from the coupler as the active CICE model (i.e., atmospheric states, shortwave fluxes, and ocean ice melt flux) and acts very similarly to CICE running in prescribed mode. Currently, this component is only used to drive POP in "C" compsets.

xml variables

The following are xml variables that CIME supports for DICE. These variables are defined in `$CIMEROOT/src/components/data_comps/dice/cime_config/config_component.xml`. These variables will appear in `env_run.xml` and are used by the DICE `cime_config/buildnml` script to generate the DICE namelist file `dice_in` and the required associated stream files for the case.

Note

These xml variables are used by the the dice's **cime_config/buildnml** script in conjunction with dice's **cime_config/namelist_definition_dice.xml** file to generate the namelist file `dice_in`.

"DICE xml variables"

xml variable	description
DICE_MODE	Mode for sea-ice component
	Valid values are: null, prescribed, ssmi, ssmi_iaf, ww3

datamode values

The xml variable `DICE_MODE` sets the streams that are associated with DICE and also sets the namelist variable `datamode` that specifies what additional operations need to be done by DICE on the streams before returning to the driver. One of the variables in `shr_strdata_nml` is `datamode`, whose value is a character string. Each data model has a unique set of `datamode` values that it supports. The valid values for `datamode` are set in the file `namelist_definition_dice.xml` using the xml variable `DICE_MODE` in the `config_component.xml` file for DICE. CIME will generate a value `datamode` that is compset dependent.

The following are the supported DICE datamode values and their relationship to the `DICE_MODE` xml variable value.

"Valid values for datamode namelist variable"

datamode variable	description
NULL	Turns off the data model as a provider of data to the coupler. The ice_present flag will be set to false and the coupler will assume no exchange of data to or from the data model.

COPYALL	The default science mode of the data model is the COPYALL mode. This mode will examine the fields found in all input data streams, if any input field names match the field names used internally, they are copied into the export array and passed directly to the coupler without any special user code. Any required fields not found on an input stream will be set to zero.
SSTDATA	Is a prognostic mode. It requires data be sent to the ice model. Ice fraction (extent) data is read from an input stream, atmosphere state variables are received from the coupler, and then an atmosphere-ice surface flux is computed and sent to the coupler. It is called SSTDATA mode because normally the ice fraction data is found in the same data files that provide SST data to the data ocean model. They are normally found in the same file because the SST and ice fraction data are derived from the same observational data sets and are consistent with each other.

DICE_MODE, datamode and streams

The following tabe describes the valid values of `DICE_MODE`, and how it relates to the associated input streams and the `datamode` namelist variable.

"Relationship between DICE_MODE, datamode and streams"

DICE_MODE, description-streams-datamode"	
null	null mode
	streams: none
	datamode: null
prescribed	prognostic mode - requires data to be sent to DICE
	streams: prescribed
	datamode: SSTDATA
ssmi	Special Sensor Microwave Imager climatological data
	streams: SSMI
	datamode: SSTDATA
ssmi	Special Sensor Microwave Imager inter-annual forcing data
	streams: SSMI_IAF
	datamode: SSTDATA
ww3	ww3 mode
	streams: ww3
	datamode: COPYALL

NIf DICE_MODE is set to `ssmi`, `ssmi_iaf` or `prescribed`, it is a prognostic mode and requires data be sent to the ice model. Ice fraction (extent) data is read from an input stream, atmosphere state variables are received from the coupler, and then an atmosphere-ice surface flux is computed and sent to the coupler. Normally the ice fraction data is found in the same data files that provide SST data to the data ocean model. They are normally found in the same file because the SST and ice fraction data are derived from the same observational data sets and are consistent with each other.

Namelists

The namelist file for DICE is `dice_in` (or `dice_in_NNN` for multiple instances).

As is the case for all data models, DICE namelists can be separated into two groups, stream-independent and stream-dependent.

The stream dependent group is shr_strdata_nml.

The stream-independent group is `dice_nml` and the DICE stream-independent namelist variables are:

decomp	decomposition strategy (1d, root) 1d => vector decomposition, root => run on master task
flux_qacc	activates water accumulation/melt wrt Q
flux_qacc0	initial water accumulation value
flux_qmin	bound on melt rate
flux_swpf	short-wave penetration factor
restfilm	master restart filename
restfils	stream restart filename
force_prognostic_true	TRUE => force prognostic behavior

To change the namelist settings in `dice_in`, edit the file `user_nl_dice`.

Streams independent of DICE_MODE value

There are no datamode independent streams for DICE.

Field names

DICE defines a set of pre-defined internal field names as well as mappings for how those field names map to the fields sent to the coupler. In general, the stream input file should translate the stream input variable names into the `dice_fld` names below for use within the data ice model.

"DICE internal field names"

dice_fld (avifld)	driver_fld (avofld)
to	So_t
s	So_s
uo	So_u
vo	So_v
dhdx	So_dhdx
dhdy	So_dhdy
q	Fioo_q
z	Sa_z
ua	Sa_u
va	Sa_v
ptem	Sa_ptem
tbot	Sa_tbot
shum	Sa_shum
dens	Sa_dens
swndr	Faxa_swndr
swvdr	Faxa_swvdr
swndf	Faxa_swndf
swvdf	Faxa_swvdf
lwdn	Faxa_lwdn

rain	Faxa_rain
snow	Faxa_snow
t	Si_t
tref	Si_tref
qref	Si_qref
ifrac	Si_ifrac
avsdr	Si_avsdr
anidr	Si_anidr
avsdf	Si_avsdf
anidf	Si_anidf
tauxa	Faii_taux
tauya	Faii_tauy
lat	Faii_lat
sen	Faii_sen
lwup	Faii_lwup
evap	Faii_evap
swnet	Faii_swnet
swpen	Fioi_swpen
melth	Fioi_melth
meltw	Fioi_meltw
salt	Fioi_salt
tauxo	Fioi_taux
tauyo	Fioi_tauy

Data Ocean (DOCN)

Data ocean can be run both as a prescribed component, simply reading in SST data from a stream, or as a prognostic slab ocean model component.

The data ocean component (DOCN) always returns SSTs to the driver. The atmosphere/ocean fluxes are computed in the coupler. Therefore, the data ocean model does not compute fluxes like the data ice (DICE) model. DOCN has two distinct modes of operation. DOCN can run as a pure data model, reading in ocean SSTs (normally climatological) from input datasets, performing time/spatial interpolations, and passing these to the coupler. Alternatively, DOCN can compute updated SSTs by running as a slab ocean model where bottom ocean heat flux convergence and boundary layer depths are read in and used with the atmosphere/ocean and ice/ocean fluxes obtained from the driver.

DOCN running in prescribed mode assumes that the only field in the input stream is SST and also that SST is in Celsius and must be converted to Kelvin. All other fields are set to zero except for ocean salinity, which is set to a constant reference salinity value. Normally the ice fraction data (used for prescribed CICE) is found in the same data files that provide SST data to the data ocean model since SST and ice fraction data are derived from the same observational data sets and are consistent with each other. For DOCN prescribed mode, default yearly climatological datasets are provided for various model resolutions.

DOCN running as a slab ocean model is used in conjunction with active ice mode running in full prognostic mode (e.g. CICE for CESM). This mode computes a prognostic sea surface temperature and a freeze/melt potential (surface Q-flux) used by the sea ice model. This calculation requires an external SOM forcing data file that includes ocean mixed layer depths and bottom-of-the-slab Q-fluxes. Scientifically appropriate bottom-of-the-slab Q-fluxes are normally ocean resolution dependent and are derived from the ocean model output of a fully coupled CCSM run. Note that this mode no longer runs out of the box, the default testing SOM forcing file is not scientifically appropriate

and is provided for testing and development purposes only. Users must create scientifically appropriate data for their particular application or use one of the standard SOM forcing files from the full prognostic control runs. For CESM, some of these are available in the inputdata repository. The user then modifies the `$DOCN_SOM_FILENAME` variable in env_run.xml to point to the appropriate SOM forcing dataset.

Note

A tool is available to derive valid SOM forcing and more information on creating the SOM forcing is also available.

xml variables

The following are xml variables that CIME supports for DOCN. These variables are defined in `$CIMEROOT/src/components/data_comps/docn/cime_config/config_component.xml`. These variables will appear in `env_run.xml` and are used by the DOCN `cime_config/buildnml` script to generate the DOCN namelist file `docn_in` and the required associated stream files for the case.

Note

These xml variables are used by the the docn's **cime_config/buildnml** script in conjunction with docn's **cime_config/namelist_definition_docn.xml** file to generate the namelist file `docn_in`.

"DOCN xml variables"

xml variable	description
DOCN_MODE	Data mode
	Valid values are: null, prescribed, som, interannual, ww3
DOCN_SOM_FILENAME	Sets SOM forcing data filename for pres runs, only used in D and E compset
SSTICE_STREAM	Prescribed SST and ice coverage stream name.
	Sets SST and ice coverage stream name for prescribed runs.
SSTICE_DATA_FILENAME	Prescribed SST and ice coverage data file name.
	Sets SST and ice coverage data file name for DOCN prescribed runs.
SSTICE_YEAR_ALIGN	The model year that corresponds to SSTICE_YEAR_START on the data file.
	Prescribed SST and ice coverage data will be aligned so that the first year of
	data corresponds to SSTICE_YEAR_ALIGN in the model. For instance, if the first
	year of prescribed data is the same as the first year of the model run, this
	should be set to the year given in RUN_STARTDATE.
	If SSTICE_YEAR_ALIGN is later than the model's starting year, or if the model is
	run after the prescribed data ends (as determined by SSTICE_YEAR_END), the
	default behavior is to assume that the data from SSTICE_YEAR_START to SSTICE_YEAR_END
	cyclically repeats. This behavior is controlled by the *taxmode* stream option
SSTICE_YEAR_START	The first year of data to use from SSTICE_DATA_FILENAME.

	This is the first year of prescribed SST and ice coverage data to use. For
	example, if a data file has data for years 0-99, and SSTICE_YEAR_START is 10,
	years 0-9 in the file will not be used.
SSTICE_YEAR_END	The last year of data to use from SSTICE_DATA_FILENAME.
	This is the last year of prescribed SST and ice coverage data to use. For
	example, if a data file has data for years 0-99, and value is 49,
	years 50-99 in the file will not be used.

Note

For multi-year runs requiring AMIP datasets of sst/ice_cov fields, you need to set the xml variables for `DOCN_SSTDATA_FILENAME`, `DOCN_SSTDATA_YEAR_START`, and `DOCN_SSTDATA_YEAR_END`. CICE in prescribed mode also uses these values.

datamode values

The xml variable `DOCN_MODE` sets the streams that are associated with DOCN and also sets the namelist variable `datamode` that specifies what additional operations need to be done by DOCN on the streams before returning to the driver. One of the variables in `shr_strdata_nml` is `datamode`, whose value is a character string. Each data model has a unique set of `datamode` values that it supports. The valid values for `datamode` are set in the file `namelist_definition_docn.xml` using the xml variable `DOCN_MODE` in the `config_component.xml` file for DOCN. CIME will generate a value `datamode` that is compset dependent.

The following are the supported DOCN datamode values and their relationship to the `DOCN_MODE` xml variable value.

"Valid values for datamode namelist variable"

datamode variable	description
NULL	Turns off the data model as a provider of data to the coupler. The ocn_present flag will be set to false and the coupler will assume no exchange of data to or from the data model.
COPYALL	The default science mode of the data model is the COPYALL mode. This mode will examine the fields found in all input data streams, if any input field names match the field names used internally, they are copied into the export array and passed directly to the coupler without any special user code. Any required fields not found on an input stream will be set to zero.
SSTDATA	assumes the only field in the input stream is SST. It also assumes the SST is in Celsius and must be converted to Kelvin. All other fields are set to zero except for ocean salinity, which is set to a constant reference salinity value. Normally the ice fraction data is found in the same data files that provide SST data to the data ocean model. They are normally found in the same file because the SST and ice fraction data are derived from the same observational data sets and are consistent with each other. They are normally found in the same file because the SST and ice fraction data are derived from the same observational data sets and are consistent with each other.
IAF	is the interannually varying version of SSTDATA

SOM	(slab ocean model) mode is a prognostic mode. This mode computes a prognostic sea surface temperature and a freeze/melt potential (surface Q-flux) used by the sea ice model. This calculation requires an external SOM forcing data file that includes ocean mixed layer depths and bottom-of-the-slab Q-fluxes. Scientifically appropriate bottom-of-the-slab Q-fluxes are normally ocean resolution dependent and are derived from the ocean model output of a fully coupled CCSM run. Note that while this mode runs out of the box, the default SOM forcing file is not scientifically appropriate and is provided for testing and development purposes only. Users must create scientifically appropriate data for their particular application. A tool is available to derive valid SOM forcing.

DOCN_MODE, datamode and streams

The following tabe describes the valid values of `DOCN_MODE`, and how it relates to the associated input streams and the `datamode` namelist variable.

"Relationship between DOCN_MODE, datamode and streams"

DOCN_MODE, description-streams-datamode"	
null	null mode
	streams: none
	datamode: null
prescribed	run with prescribed climatological SST and ice-coverage
	streams: prescribed
	datamode: SSTDATA
interannual	run with interannual SST and ice-coverage
	streams: prescribed
	datamode: SSTDATA
som	run in slab ocean mode
	streams: som
	datamode: SOM
ww3	ww3 mode
	streams: ww3
	datamode: COPYALL

Namelists

As is the case for all data models, DOCN namelists can be separated into two groups, stream-independent and stream-dependent.

The namelist file for DOCN is `docn_in` (or `docn_in_NNN` for multiple instances).

The stream dependent group is shr_strdata_nml .

As part of the stream dependent namelist input, DOCN supports two science modes, `SSTDATA` (prescribed mode) and `SOM` (slab ocean mode).

The stream-independent group is `docn_nml` and the DOCN stream-independent namelist variables are:

decomp	decomposition strategy (1d, root) 1d => vector decomposition, root => run on master task
restfilm	master restart filename

restfils	stream restart filename
force_prognostic_true	TRUE => force prognostic behavior

To change the namelist settings in docn_in, edit the file user_nl_docn.

Datamode independent streams

There are no datamode independent streams for DOCN.

Field names

DOCN defines a set of pre-defined internal field names as well as mappings for how those field names map to the fields sent to the coupler. In general, the stream input file should translate the stream input variable names into the `docn_fld` names below for use within the data ocn model.

"DOCN internal field names"

docn_fld (avifld)	driver_fld (avofld)
ifrac	Si_ifrac
pslv	Sa_pslv
duu10n	So_duu10n
taux	Foxx_taux
tauy	Foxx_tauy
swnet	Foxx_swnet
lat	Foxx_lat
sen	Foxx_sen
lwup	Foxx_lwup
lwdn	Faxa_lwdn
melth	Fioi_melth
salt	Fioi_salt
prec	Faxa_prec
snow	Faxa_snow
rain	Faxa_rain
evap	Foxx_evap
meltw	Fioi_meltw
rofl	Foxx_rofl
rofi	Foxx_rofi
t	So_t
u	So_u
v	So_v
dhdx	So_dhdx
dhdy	So_dhdy
s	So_s
q	Fioo_q
h	strm_h

qbot	strm_qbot
fswpen	So_fswpen

Creating SSTDATA mode input from a fully prognostic run (CESM only)

The following outlines the steps you would take to create monthly averages of SST and ice coverage from a previous fully prognostic run that can then be read as as stream data by DOCN.

As an example, the following uses an f09_g16 CESM B-configuration simulation using CAM5 physics and with cosp enabled. The procedure to create the SST/ICE file is as follows:

1. Save monthly averaged 'aice' information from cice code (this is the default).
2. Save monthly averaged SST information from pop2. To do this, copy $SRCROOT/pop2/input_templates/gx1v6_tavg_contents to $CASEROOT/SourceMods/src.pop2 and change the 2 in front of SST to 1 for monthly frequency.
3. Extract (using ncrcat) SST from monthly pop2 history files and form a single netcdf file containing just SST; change SST to SST_cpl.

```
> ncrcat -v SST case.pop.h.*.nc temp.nc
> ncrename -v SST,SST_cpl temp.nc sst_cpl.nc
```

4. Extract aice from monthly cice history files and form a single netcdf file containing aice; change aice to ice_cov; divide values by 100 (to convert from percent to fraction).

```
> ncrcat -v aice case.cice.h.*.nc temp.nc
> ncrename -v aice,ice_cov temp.nc temp2.nc
> ncap2 -s 'ice_cov=ice_cov/100.' temp2.nc ice_cov.nc
```

5. Modify fill values in the sst_cpl file (which are over land points) to have value -1.8 and remove fill and missing value designators; change coordinate lengths and names: to accomplish this, first run ncdump, then replace _ with -1.8 in SST_cpl, then remove lines with _FillValue and missing_value. (Note: although it might be possible to merely change the fill value to -1.8, this is conforming to other SST/ICE files, which have SST_cpl explicitly set to -1.8 over land.) To change coordinate lengths and names, replace nlon by lon, nlat by lat, TLONG by lon, TLAT by lat. The last step is to run ncgen. Note: when using ncdump followed by ncgen, precision will be lost; however, one can specify -d 9,17 to maximize precision - as in the following example:

```
> ncdump -d 9,17 old.nc > old
> ncgen -o new.nc new
```

6. Modify fill values in the ice_cov file (which are over land points) to have value 1 and remove fill and missing value designators; change coordinate lengths and names; patch longitude and latitude to replace missing values. To accomplish this, first run ncdump, then replace _ with 1 in ice_cov, then remove lines with _FillValue and missing_value. To change coordinate lengths and names, replace ni by lon, nj by lat, TLON by lon, TLAT by lat. To patch longitude and latitude arrays, replace values of those arrays with those in sst_cpl file. The last step is to run ncgen. (Note: the replacement of longitude and latitude missing values by actual values should not be necessary but is safer.)
7. Combine (using ncks) the two netcdf files.

```
> ncks -v ice_cov ice_cov.nc sst_cpl.nc
```

   Rename the file to ssticetemp.nc. The time variable will refer to the number of days at the end of each month, counting from year 0, whereas the actual simulation began at year 1. However, we want time values to be in the middle of each month, referenced to the first year of the simulation (first time value equals 15.5). Extract (using ncks) time variable from existing amip sst file (for correct number of months - 132 in this example) into working netcdf file.

```
> ncks -d time,0,131 -v time amipsst.nc ssticetemp.nc
```

   Add date variable: ncdump date variable from existing amip sst file; modify first year to be year 0 instead of 1949 (do not including leading zeroes or it will interpret as octal) and use correct number of months; ncgen to new netcdf file; extract date (using ncks) and place in working netcdf file.

```
> ncks -v date datefile.nc ssticetemp.nc
```

Add datesec variable: extract (using ncks) datesec (correct number of months) from existing amip sst file and place in working netcdf file.

```
> ncks -d time,0,131 -v datesec amipsst.nc ssticetemp.nc
```

8. At this point, you have an SST/ICE file in the correct format.
9. Due to CAM's linear interpolation between mid-month values, you need to apply a procedure to assure that the computed monthly means are consistent with the input data. To do this, invoke `$SRCROOT/components/cam/tools/icesst/bcgen` and following the following steps:
   a. Rename SST_cpl to SST, and ice_cov to ICEFRAC in the current SST/ICE file:
   ```
   > ncrename -v SST_cpl,SST -v ice_cov,ICEFRAC ssticetemp.nc
   ```
   b. In driver.f90, sufficiently expand the lengths of variables prev_history and history (16384 should be sufficient); also comment out the test that the climate year be between 1982 and 2001 (lines 152-158).
   c. In bcgen.f90 and setup_outfile.f90, change the dimensions of xlon and ???TODO xlat to (nlon,nlat); this is to accommodate use of non-cartesian ocean grid.
   d. In setup_outfile.f90, modify the 4th and 5th ???TODO arguments in the calls to wrap_nf_def_var for *lon* and *lat* to be *2* and *dimids*; this is to accommodate use of non-cartesian ocean grid.
   e. Adjust Makefile to have proper path for LIB_NETCDF and INC_NETCDF.
   f. Modify namelist accordingly.
   g. Make bcgen and execute per instructions. The resulting sstice_ts.nc file is the desired ICE/SST file.
9. Place the new SST/ICE file in desired location and modify `env_run.xml` to have :
   a. `SSTICE_DATA_FILENAME` point to the complete path of your SST/ICE file.
   b. `SSTICE_GRID_FILENAME` correspond to full path of (in this case) gx1v6 grid file.
   c. `SSTICE_YEAR_START` set to 0
   d. `SSTICE_YEAR_END` to one less than the total number of years
   e. `SSTICE_YEAR_ALIGN` to 1 (for CESM, since CESM starts counting at year 1).

Data River (DROF)

The data river model (DROF) provides river runoff data primarily to be used by the prognostic ocean component. This data can either be observational (climatological or interannual river data) or data from a previous model run that is output to coupler history files and then read back in by DROF.

xml variables

The following are xml variables that CIME supports for DROF. These variables are defined in `$CIMEROOT/src/components/data_comps/drof/cime_config/config_component.xml`. These variables will appear in `env_run.xml` and are used by the DROF `cime_config/buildnml` script to generate the DROF namelist file `drof_in` and the required associated stream files for the case.

> **Note**
>
> These xml variables are used by the the drof's **cime_config/buildnml** script in conjunction with drof's **cime_config/namelist_definition_drof.xml** file to generate the namelist file `drof_in`.

"DROF xml variables"

xml variable	description
DROF_MODE	Data mode
	Valid values are: NULL,CPLHIST,DIATREN_ANN_RX1,DIATREN_IAF_RX1

DROF_CPLHIST_CASE	Coupler history data mode case name
DROF_CPLHIST_DIR	Coupler history data mode directory containing coupler history data
DROF_CPLHIST_YR_ALIGN	Coupler history data model simulation year corresponding to data starting year
DROF_CPLHIST_YR_START	Coupler history data model starting year to loop data over
DROF_CPLHIST_YR_END	Coupler history data model ending year to loop data over

datamode values

The xml variable `DROF_MODE` sets the streams that are associated with DROF and also sets the namelist variable `datamode` that specifies what additional operations need to be done by DROF on the streams before returning to the driver. One of the variables in `shr_strdata_nml` is `datamode`, whose value is a character string. Each data model has a unique set of `datamode` values that it supports. The valid values for `datamode` are set in the file `namelist_definition_drof.xml` using the xml variable `DROF_MODE` in the `config_component.xml` file for DROF. CIME will generate a value `datamode` that is compset dependent.

The following are the supported DROF datamode values and their relationship to the `DROF_MODE` xml variable value.

"Valid values for datamode namelist variable"

datamode variable	description
NULL	Turns off the data model as a provider of data to the coupler. The rof_present flag will be set to false and the coupler will assume no exchange of data to or from the data model.
COPYALL	Copies all fields directly from the input data streams Any required fields not found on an input stream will be set to zero.

Namelists

The data river runoff model (DROF) provides data river input to prognostic components such as the ocean.

The namelist file for DROF is `drof_in`.

As is the case for all data models, DROF namelists can be separated into two groups, stream-independent and stream-dependent. The stream dependent group is shr_strdata_nml. The stream-independent group is `drof_nml` and the DROF stream-independent namelist variables are:

decomp	decomposition strategy (1d, root) 1d => vector decomposition, root => run on master task
restfilm	master restart filename
restfils	stream restart filename
force_prognostic_true	TRUE => force prognostic behavior

To change the namelist settings in `drof_in`, edit the file `user_nl_drof` in your case directory.

DROF_MODE, datamode and streams

The following tabe describes the valid values of `DROF_MODE`, and how it relates to the associated input streams and the `datamode` namelist variable.

"Relationship between DROF_MODE, datamode and streams"

DROF_MODE	description-streams-datamode
NULL	null mode
	streams: none
	datamode: NULL
DIATREN_ANN_RX1	Reads in annual forcing river data used for CORE2 forcing runs.
	streams: rof.diatren_ann_rx1
	datamode: COPYALL
DIATREN_IAF_RX1	Reads in intra-annual forcing river data used for CORE2 forcing runs.
	streams: rof.diatren_iaf_rx1
	datamode: COPYALL
CPLHIST	Reads in data from coupler history files generated by a previous run.
	streams: rof.cplhist
	datamode: COPYALL

Streams independent of DROF_MODE value

There are no datamode independent streams for DROF.

DROF Field names

DROF defines a set of pre-defined internal field names as well as mappings for how those field names map to the fields sent to the coupler. In general, the stream input file should translate the stream input variable names into the `drof_fld` names for use within the data rofosphere model.

"DROF internal field names"

drof_fld (avifld)	driver_fld (avofld)
roff	Forr_rofl
ioff	Forr_rofi

Data Wave (DWAV)

The data wave model (DWAV) provides data wave forcing primarily to be used by the prognostic ocean component. Currently, this data is climatological.

xml variables

The following are XML variables that CIME supports for DWAV. These variables will appear in `env_run.xml` and are used by the DWAV `cime_config/buildnml` script to generate the DWAV namelist file `dwav_in` and the required associated stream files for the case.

> ***Note***
>
> These XML variables are used by the the DWAV **cime_config/buildnml** script in conjunction with the DWAV **cime_config/namelist_definition_dwav.xml** file to generate the namelist file `dwav_in`.

"DROF xml variables"

xml variable	description
DWAV_MODE	Data mode
	Valid values are: NULL, CLIMO

DWAV datamode values

One of the variables in `shr_strdata_nml` is `datamode`, whose value is a character string. Each data model has a unique set of `datamode` values that it supports.

The valid values for `datamode` are set by the XML variable `DWAV_MODE` in the `config_component.xml` file for DWAV. CIME will generate a value `datamode` that is compset dependent.

The following are the supported DWAV datamode values and their relationship to the `DWAV_MODE` xml variable value.

Relationship between `DWAV_MODE` *xml variables and* `datamode` *namelist variables*

DWAV_MODE (xml)	datamode (namelist)
NULL	NULL
	This mode turns off the data model as a provider of data to the coupler.
	The `wav_present` flag will be set to `false` and the coupler assumes no exchange of data to or from the data model.
CLIMO	COPYALL
	Examines the fields found in all input data streams and if any input field names match the field names used internally,
	they are copied into the export array and passed directly to the coupler without any special user code.

Namelists

The data wave model (DWAV) provides data wave input to prognostic components such as the ocean.

The namelist file for DWAV is `dwav_in`.

As is the case for all data models, DWAV namelists can be separated into two groups, stream-independent and stream-dependent. The stream dependent group is shr_strdata_nml. The stream-independent group is `dwav_nml` and the DWAV stream-independent namelist variables are:

decomp	decomposition strategy (1d, root) 1d => vector decomposition, root => run on master task
restfilm	master restart filename
restfils	stream restart filename
force_prognostic_true	TRUE => force prognostic behavior

To change the namelist settings in `dwav_in`, edit the file `user_nl_dwav` in your case directory.

DROF_MODE, datamode and streams

The following tabe describes the valid values of `DWAV_MODE`, and how it relates to the associated input streams and the `datamode` namelist variable.

"Relationship between DWAV_MODE, datamode and streams"

DWAV_MODE	description-streams-datamode
NULL	null mode
	streams: none
	datamode: NULL

Streams independent of DWAV_MODE value

There are no datamode independent streams for DWAV.

Field names

DWAV defines a set of pre-defined internal field names as well as mappings for how those field names map to the fields sent to the coupler. In general, the stream input file should translate the stream input variable names into the `dwav_fld` names below for use within the data wave model.

"DWAV internal field names"

dwav_fld (avifld)	driver_fld (avofld)
lamult	Sw_lamult
ustokes	Sw_ustokes
vstokes	Sw_vstokes

Indices and tables

- **`genindex`**
- **`modindex`**
- **`search`**

CIME Driver/Coupler

Introduction

The following provides an overview of the CIME driver/coupler. We will cover the top level driver implementation as well as the coupler component within the system. The driver runs on all hardware processors, runs the top level instructions, and, executes the driver time loop. The coupler is a component of the CIME infrastructure that is run from within the driver. It can be run on a subset of the total processors, and carries out mapping (interpolation), merging, diagnostics, and other calculations. The name cpl7 refers to the source code associated with both the driver and the coupler parts of the model. cpl7 code is located in the CIME source tree under driver_cpl/ and the main program of `driver_cpl/driver/cesm_driver.F90`.

We also provide a general overview of the cpl7 design. Specific implementation issues are then discussed individually. Finally, there is a section summarizing all of the cpl7 namelist input. This document is written primarily to help users understand the inputs and controls within the cpl7 system, but to also provide some background about the associated implementation. Coupler flow diagrams are provided in a separate document. Some additional documentation on how the coupler works can be found in Craig et al, "A New Flexible Coupler for Earth System Modeling Developed for CCSM4 and CESM1", International Journal of High Performance Computing Applications 2012 26: 31 DOI: 10.1177/1094342011428141.

Design

Overview

cpl7 is built as a single executable with a single high-level driver. The driver runs on all processors and handles coupler sequencing, model concurrency, and communication of data between components. The driver calls all model components via common and standard interfaces. The driver also directly calls coupler methods for mapping (interpolation), rearranging, merging, an atmosphere/ocean flux calculation, and diagnostics. The model components and the coupler methods can run on subsets of all the processors. In other words, cpl7 consists of a driver that controls the top level sequencing, the processor decomposition, and communication between components and the coupler while coupler operations such as mapping and merging are running under the driver on a subset of processors as if there were a unique coupler model component.

In general, an active component both needs data from and provides data to the coupler while data models generally read data from I/O and then just provide data to the coupler. Currently, the atmosphere, land, river, and sea ice models are always tightly coupled to better resolve the diurnal cycle. This coupling is typically half-hourly, although at higher resolutions, can be more frequent. The ocean model coupling is typically once or a few times per day. The diurnal cycle of ocean surface albedo is computed in the coupler for use by the atmosphere model. The looser ocean coupling frequency means the ocean forcing and response is lagged in the system. There is an option in cpl7 to run the ocean tightly coupled without any lags, but this is more often used only when running with data ocean components.

Sequencing and Concurrency

The component processor layouts and MPI communicators are derived from namelist input. At the present time, there are eight (10) basic processor groups in cpl7. These are associated with the atmosphere, land, river, ocean, sea ice, land ice, wave, external-system-process, coupler, and global groups, although others could be easily added later. Each of the processor groups can be distinct, but that is not a requirement of the system. A user can overlap processor groups relatively arbitrarily. If all processors sets overlap each other in at least one processor, then the model runs sequentially. If all processor sets are distinct, the model runs as concurrently as science allows. The processor sets for each component group are described via 3 basic scalar parameters at the present time; the number of mpi tasks, the number of openmp threads per mpi task, and the global mpi task rank of the root mpi task for that group. For example, a layout where the number of mpi tasks is 8, the number of threads per mpi task is 4, and the root mpi task is 16 would create a processor group that consisted of 32 hardware processors, starting on global mpi task number 16 and it would contain 8 mpi tasks. The global group would have at least 24 tasks and at least 48 hardware processors. The driver derives all MPI communicators at initialization and passes them to the component models for use. More information on the coupler concurrency can be found in the Craig et al IJHPCA 2012 reference mentioned in the top section of this document.

As mentioned above, there are two issues related to whether the component models run concurrently. The first is whether unique chunks of work are running on distinct processor sets. The second is the sequencing of this work in the driver. As much as possible, the driver sequencing has been implemented to maximize the potential amount of concurrency of work between different components. Ideally, in a single coupling step, the forcing for all models would be computed first, the models could then all run concurrently, and then the driver would advance. However, scientific requirements such as the coordination of surface albedo and atmosphere radiation computations as well as general computational stability issues prevents this ideal implementation in cpl7. Figure 1 shows the maximum amount of concurrency supported for a fully active system. In practice, the scientific constraints mean the active atmosphere model cannot run concurrently with the land, runoff, and sea-ice models. Again, figure 1 does not necessarily represent the optimum processor layout for performance for any configuration, but it provides a practical limit to the amount of concurrency in the system due to scientific constraints. Results are bit-for-bit identical regardless of the component sequencing because the scientific lags are fixed by the implementation, not the processor layout.

image:: cplug-02.1-figx1.jpg

Figure 1: Maximum potential processor concurrency designed to support scientific requirements and stability.

Component Interfaces

The standard cpl7 component model interfaces are based upon the ESMF design. Each component provides an init, run, and finalize method with consistent arguments. The component interface arguments currently consist of Fortran and MCT datatypes. The physical coupling fields are passed through the interfaces in the init, run, and finalize phases. As part of initialization, an MPI communicator is passed from the driver to the component, and grid and decomposition information is passed from the component back to the driver. The driver/coupler acquires all information about resolution, configurations, and processor layout at run-time from either namelist or from communication with components.

Initialization of the system is relatively straight-forward. First, the MPI communicators are computed in the driver. Then the component model initialization methods are called on the appropriate processor sets, and an mpi communicator is sent, and the grid and decomposition information are passed back to the driver. Once the driver has all the grid and decomposition information from the components, various rearrangers and mappers are initialized that will move data between processors, decompositions, and grids as needed at the driver level. No distinction is made in the coupler implementation for sequential versus concurrent execution. In general, even for cases where two components have identical grids and processor layouts, often their decomposition is different for performance reasons. In cases where the grid, decomposition, and processor layout are identical between components, the mapping or rearranging operation will degenerate to a local data copy.

The interface to the components' run method consists of two distinct bundles of fields. One is the data sent to force the model. The second is data received from the model for coupling to other components. The run interface also contains a clock that specifies the current time and the run length for the model and a data type that encapsulates grid, decomposition, and scalar coupling information. These interfaces generally follow the ESMF design principles.

MCT, The Model Coupling Toolkit

In cpl7, the MCT attribute_vector, global_segmap, and general_grid datatypes have been adopted at the highest levels of the driver, and they are used directly in the component init, run, and finalize interfaces. In addition, MCT is used for all data rearranging and mapping (interpolation). The clock used by cpl7 at the driver level is based on the ESMF specification. Mapping weights are still generated off-line using the SCRIP or ESMF packages as a preprocessing step. They are read using a subroutine that reads and distributes the mapping weights in reasonably small chunks to minimize the memory footprint. Development of the cpl7 coupler not only relies on MCT, but MCT developers contributed significantly to the design and implementation of the cpl7 driver. Development of cpl7 coupler resulted from a particularly strong and close collaboration between NCAR and the Department of Energy Argonne National Lab.

Memory, Parallel IO, and Performance

Scaling to tens-of-thousands of processors requires reasonable performance scaling of the models, and all components have worked at improving scaling via changes to algorithms, infrastructure, or decompositions. In particular, decompositions using shared memory blocking, space filling curves, and all three spatial dimensions have been implemented to varying degrees in all components to increase parallelization and improve scalability. The Craig et al IJHPCA 2012 reference mentioned in the first section of this document provides a summary of scaling performance of cpl7 for several coupler kernals.

In practice, performance, load balance, and scalability are limited as a result of the size, complexity, and multiple model character of the system. Within the system, each component has its own scaling characteristics. In particular, each may have processor count "sweet-spots" where the individual component model performs particularly well. This might occur within a component because of internal load balance, decomposition capabilities, communication patterns, or cache usage. Second, component performance can vary over the length of the model run. This occurs because of seasonal variability of the cost of physics in models, changes in performance during an adjustment (spin-up) phase, and temporal variability in calling certain model operations like radiation, dynamics, or I/O. Third, the hardware or batch queueing system might have some constraints on the total number of processors that are available. For instance, on 16 or 32 way shared memory node, a user is typically charged based on node usage, not processor usage. So there is no cost savings running on 40 processors versus 64 processors on a 32-way node system. As a result of all of these issues, perfect load-balancing is generally not possible. But to a large degree, if one accepts the limitations, a load balance configuration with acceptable idle-time and reasonably good throughput is nearly always possible to configure.

Load-balancing requires a number of considerations such as which components are run, their absolute resolution, and their relative resolution; cost, scaling and processor count sweet-spots for each component; and internal load imbalance within a component. It is often best to load balance the system with all significant run-time I/O turned off because this generally occurs very infrequently (typically one timestep per month), is best treated as a separate cost, and can bias interpretation of the overall model load balance. Also, the use of OpenMP threading in some or all of the system is dependent on the hardware/OS support as well as whether the system supports running all MPI and mixed MPI/OpenMP on overlapping processors for different components. Finally, should the components run sequentially, concurrently, or some combination of the two. Typically, a series of short test runs is done with the desired production configuration to establish a reasonable load balance setup for the production job.

Implementation

Time Management

Driver Clocks

The driver manages the main clock in the system. The main clock advances at the shortest coupling period and uses alarms to trigger component coupling and other events. In addition, the driver maintains a clock that is associated with each component. The driver's component clocks have a timestep associated with the coupling period of that component. The main driver clock and the component clocks in the driver advance in a coordinated manor and are always synchronized. The advancement of time is managed as follows in the main run loop. First, the main driver clock advances one timestep and the component clocks are advanced in a synchronous fashion. The clock time represents the time at the end of the next model timestep. Alarms may be triggered at that timestep to call the the atmosphere, land, runoff, sea ice, land ice, or ocean run methods. If a component run alarm is triggered, the run method is called and the driver passes that component's clock to that component. The component clock contains information about the length of the next component integration and the expected time of the component at the end of the integration period.

Generally, the component models have indepedent time management software. When a component run method is called, the component must advance the proper period and also check that their internal clock is consistent with the coupling clock before returning to the driver. The clock passed to the component by the driver contains this information. Component models are also responsible for making sure the coupling period is consistent with their internal timestep. History files are managed independently by each component, but restart files are coordinated by the driver.

The driver clocks are based on ESMF clock datatype are are supported in software by either an official ESMF library or by software included in CIME called `esmf_wrf_timemgr`, which is a much simplified Fortran implementation of a subset of the ESMF time manager interfaces.

The Driver Time Loop

The driver time loop is hardwired to sequence the component models in a specific way to meet scientific requirements and to otherwise provide the maximum amount of potential concurrency of work. The results of the model integration are not dependent on the processor layout of the components. See the Craig et al IJHPCA 2012 reference for further details.

In addition, the driver is currently configured to couple the atmosphere, land, and sea ice models using the same coupling frequency while the runoff, land ice, and ocean model can be coupled at the same or at a lower frequency. To support this feature, the driver does temporal averaging of coupling inputs to the ocean and runoff, and the driver also computes the surface ocean albedo at the higher coupling frequency. There is no averaging of coupling fields for other component coupling interactions and the land and sea ice models' surface albedos are computed inside those components. Averaging functionality could be added to the driver to support alternative relative coupling schemes in the future if desired with the additional caveat that the interaction between the surface albedo computation in each component and the atmospheric radiation calculation have to be carefully considered. In addition, some other features may need to be extended to support other coupling schemes and still allow model concurrency.

The coupler processors (pes) handle the interaction of data between components, so there are separate tasks associated with deriving fields on the coupler pes, transfering data to and from the coupler pes and other components, and then running the component models on their processors. The driver time loop is basically sequenced as follows,

```
- The driver clock is advanced first and alarms set.
- Input data for ocean, land, sea ice, and runoff is computed.
- Ocean data is rearranged from the coupler to the ocean pes.
- Land data is rearranged from the coupler to the land pes.
- Ice data is rearranged from the coupler to the ice pes.
- Runoff data is rearranged from the coupler to the ice pes.
- The ice model is run.
- The land model is run.
- The runoff model is run.
- The ocean model is run.
- The ocean inputs are accumulated, and the atmosphere/ocean fluxes are
  computed on the coupler pes based on the results from the previous
  atmosphere and ocean coupled timestep.
```

```
- Land data is rearranged from the land pes to the coupler pes.
- Land ice input is computed.
- Land ice data is rearranged from the coupler to the land ice pes.
- River output (runoff) data is rearranged from the runoff pes to the coupler pes.
- Ice data is rearranged from the ice pes to the coupler pes.
- Coupler fractions are updated.
- Atmospheric forcing data is computed on the coupler pes.
- Atmospheric data is rearranged from the coupler pes to the atmosphere pes.
- The atmosphere model is run.
- The land ice model is run.
- Land ice data is rearranged from the land ice pes to the coupler pes.
- Atmospheric data is rearranged from the atmosphere pes to the coupler pes.
- Ocean data is rearranged from the ocean pes to the coupler pes.
- The loop returns

 Within this loop, as much as possible, coupler work associated
 with mapping data, merging fields, diagnosing, applying area corrections,
 and computing fluxes is overlapped with component work.
```

The land ice model interaction is slightly different.

```
- The land ice model is run on the land grid
- Land model output is passed to the land ice model every land coupling period.
- The driver accumluates this data, interpolates the data to the land ice grid,
  and the land ice model advances the land ice model about once a year.
```

The runoff coupling should be coupled at a frequency between the land coupling and ocean coupling frequencies. The runoff model runs at the same time as the land and sea ice models when it runs.

The current driver sequencing has been developed over nearly two decades, and it plays a critical role in conserving mass and heat, minimizing lags, and providing stability in the system. The above description is consistent with the concurrency limitations described here. Just to reiterate, the land, runoff, and sea ice models will always run before the atmospheric model, and the coupler and ocean models are able to run concurrently with all other components. The coupling between the atmosphere, land, sea ice, and atmosphere/ocean flux computation incurs no lags but the coupling to the ocean state is lagged by one ocean coupling period in the system. Mass and heat are conserved in the system with more description here.

It is possible to reduce the ocean lag in the system. A driver namelist variable, `ocean_tight_coupling`, moves the step where ocean data is rearranged from the ocean pes to the coupler pes from the end of the loop to before the atmosphere/ocean flux computation. If ocean_tight_coupling is set to true, then the ocean lag is reduced by one atmosphere coupling period, but the ability of the ocean model to run concurrently with the atmosphere model is also reduced or eliminated. This flag is most useful when the ocean coupling frequency matches the other components.

Coupling Frequency

In the current implementation, the coupling period must be identical for the atmosphere, sea ice, and land components. The ocean coupling period can be the same or greater. The runoff coupling period should be between or the same as the land and ocean coupling period. All coupling periods must be multiple integers of the smallest coupling period and will evenly divide the NCPL_BASE_PERIOD, typically one day, set in env_run.xml. The coupling periods are set using the NCPL env variables in env_run.xml.

The coupling periods are set in the driver namelist for each component via variables called something like atm_cpl_dt and atm_cpl_offset. The units of these inputs are seconds. The coupler template file derives these values from CIME script variable names like ATM_NCPL which is the coupling frequency per day. The *_cpl_dt input specifies the coupling period in seconds and the *_cpl_offset input specifies the temporal offset of the coupling time relative to initial time. An example of an offset might be a component that couples every six hours. That would normally be on the 6th, 12th, 18th, and 24th hour of every day. An offset of 3600 seconds would change the coupling to the 1st, 7th, 13th, and 19th hour of every day. The offsets cannot be larger than the coupling period and the sign of the offsets is such that a positive offset shifts the alarm time forward by that number of seconds. The offsets are of limited use right now because of the limitations of the relative coupling frequencies.

Offsets play an important role in supporting concurrency. There is an offset of the smallest coupling period automatically introduced in every coupling run alarm for each component clock. This is only mentioned because it is

an important but subtle point of the implementation and changing the coupling offset could have an impact on concurrency performance. Without this explicit automatic offset, the component run alarms would trigger at the end of the coupling period. This is fine for components that are running at the shortest coupling period, but will limit the ability of models to run concurrently for models that couple at longer periods. What is really required for concurrency is that the run alarm be triggered as early as possible and that the data not be copied from that component to the coupler pes until the coupling period has ended. The detailed implementation of this feature is documented in the seq_timemgr_mod. 90 file and the impact of it for the ocean coupling is implemented in the ccsm_driver.F90 code via use of the ocnrun_alarm and ocnnext_alarm variables.

Grids

Standard Grid Configurations

The standard implementation for grids in CIME has been that the atmosphere and land models are run on identical grids and the ocean and sea ice model are run on identical grids. The ocean model mask is used to derive a complementary mask for the land grid such that for any given combination of atmosphere/land and ocean/ice grids, there is a unique land mask. This approach for dealing with grids is still used a majority of the time. But there is a new capability, called `trigrid` that allows the atmosphere and land grids to be unique. A typical grid is the finite volume "1 degree" atmosphere/land grid matched with the "1 degree" ocean/ice grid. The runoff grid is generally unique to runoff and the land ice grid is coupled on the land grid with interpolation carried out to a unique land ice grid inside that component.

Historically, the ocean grid has been the higher resolution grid in CIME model configurations. While that is no longer always the case, the current driver implementation largely reflects that presumption. The atmosphere/ocean fluxes in the coupler are computed on the ocean grid. The driver namelist variable `aoflux_grid` allows users to specify the atmosphere/ocean flux computation grid in the coupler in the future. In addition, the default mapping approach used also reflects the presumption that the ocean is generally higher resolution. Fluxes are always mapped using a locally conservative area average methods to preserve conservation. However, states are mapped using bilinear interpolation from the atmosphere grid to the ocean grid to better preserve gradients, while they are mapped using a locally conservative area average approach from the ocean grid to the atmosphere grid. These choices are based on the presumption that the ocean grid is higher resolution.

There has always been an option that all grids (atmosphere, land, ocean, and ice) could be identical, and this is still supported. There are a couple of namelist variables, `samegrid_ao`, `samegrid_al`, and `samegrid_ro` that tell the coupler whether to expect that the following grids; atmosphere/ocean, atmosphere/land, and runoff/ocean respectively are identical. These are set automaticaly in the driver namelist depending on the grid chosen and impact mapping as well as domain checking.

Trigrid Configurations

Grid configurations are allowed where the atmosphere and land grids are unique.

The trigrid implementation introduces an ambiguity in the definition of the mask. This ambiguity is associated with an inability to define an absolutely consistent ocean/land mask across all grids in the system. A summary of trigrid support follows: - The land mask is defined on the atmosphere grid as the complement of the ocean mask mapped conservatively to the atmosphere grid. - Then the land and ocean masks are exactly complementary on the atmosphere grid where conservative merging are critical. - No precise land fraction needs to be defined in the land grid. - The only requirement is that the land model compute data on a masked grid such that when mapped to the atmosphere grid, all atmosphere grid points that contain some fraction of land have valid values computed in the land model. - There are an infinite number of land fraction masks that can accomplish this including a fraction field that is exactly one at every grid cell. - In the land model, all land fraction masks produce internally conservative results. - The only place where the land fraction becomes important is mapping the land model output to the runoff model. - In that case, the land fraction on the land grid is applied to the land to runoff mapping.

Fractions

The component grid fractions in the coupler are defined and computed in `$CIMEROOT/driver_cpl/driver/seq_frac_mct`. A slightly modified version of the notes from this file is pasted below. Just to clarify some of the terms. - fractions_a, fractions_l, fractions_i, and fractions_o are the fractions on the atmosphere, land, ice, and ocean grids. - afrac, lfrac, ifrac, and ofrac are the atmosphere, land, ice, and ocean fractions on those grids. So fractions_a(lfrac) is the land fraction on the atmosphere grid. lfrin in the land fraction defined in the land model. This can be different from lfrac because of the trigrid implementation. lfrac is the land

fraction consistent with the ocean mask and lfrin is the land fraction in the land model. ifrad and ofrad are fractions at the last radiation timestep. These fractions preserve conservation of heat in the net shortwave calculation because the net shortwave calculation is one timestep behind the ice fraction evolution in the system. When the variable "dom" is mentioned below, that refers to a field sent from a component at initialization.

```
!  the fractions fields are now afrac, ifrac, ofrac, lfrac, and lfrin.
!    afrac = fraction of atm on a grid
!    lfrac = fraction of lnd on a grid
!    ifrac = fraction of ice on a grid
!    ofrac = fraction of ocn on a grid
!    lfrin = land fraction defined by the land model
!    ifrad = fraction of ocn on a grid at last radiation time
!    ofrad = fraction of ice on a grid at last radiation time
!      afrac, lfrac, ifrac, and ofrac are the self-consistent values in the
!      system.  lfrin is the fraction on the land grid and is allowed to
!      vary from the self-consistent value as descibed below.  ifrad
!      and ofrad are needed for the swnet calculation.
!  the fractions fields are defined for each grid in the fraction bundles as
!    needed as follows.
!    character(*),parameter :: fraclist_a = 'afrac:ifrac:ofrac:lfrac:lfrin'
!    character(*),parameter :: fraclist_o = 'afrac:ifrac:ofrac:ifrad:ofrad'
!    character(*),parameter :: fraclist_i = 'afrac:ifrac:ofrac'
!    character(*),parameter :: fraclist_l = 'afrac:lfrac:lfrin'
!    character(*),parameter :: fraclist_g = 'gfrac'
!
!  we assume ocean and ice are on the same grids, same masks
!  we assume ocn2atm and ice2atm are masked maps
!  we assume lnd2atm is a global map
!  we assume that the ice fraction evolves in time but that
!    the land model fraction does not.  the ocean fraction then
!    is just the complement of the ice fraction over the region
!    of the ocean/ice mask.
!  we assume that component domains are filled with the total
!    potential mask/fraction on that grid, but that the fractions
!    sent at run time are always the relative fraction covered.
!    for example, if an atm cell can be up to 50% covered in
!    ice and 50% land, then the ice domain should have a fraction
!    value of 0.5 at that grid cell.  at run time though, the ice
!    fraction will be between 0.0 and 1.0 meaning that grid cells
!    is covered with between 0.0 and 0.5 by ice.  the "relative" fractions
!    sent at run-time are corrected by the model to be total fractions
!    such that
!  in general, on every grid,
!              fractions_*(afrac) = 1.0
!              fractions_*(ifrac) + fractions_*(ofrac) + fractions_*(lfrac) = 1.0
!  where fractions_* are a bundle of fractions on a particular grid and
!    *frac (ie afrac) is the fraction of a particular component in the bundle.
!
!  the fractions are computed fundamentally as follows (although the
!    detailed implementation might be slightly different)
!  initialization (frac_init):
!    afrac is set on all grids
!      fractions_a(afrac) = 1.0
!      fractions_o(afrac) = mapa2o(fractions_a(afrac))
!      fractions_i(afrac) = mapa2i(fractions_a(afrac))
!      fractions_l(afrac) = mapa2l(fractions_a(afrac))
!    initially assume ifrac on all grids is zero
!      fractions_*(ifrac) = 0.0
!    fractions/masks provided by surface components
!      fractions_o(ofrac) = dom_o(frac)  ! ocean "mask"
!      fractions_l(lfrin) = dom_l(frac)  ! land model fraction
```

```
!    then mapped to the atm model
!      fractions_a(ofrac) = mapo2a(fractions_o(ofrac))
!      fractions_a(lfrin) = mapl2a(fractions_l(lfrin))
!    and a few things are then derived
!      fractions_a(lfrac) = 1.0 - fractions_a(ofrac)
!        this is truncated to zero for very small values (< 0.001)
!        to attempt to preserve non-land gridcells.
!      fractions_l(lfrac) = mapa2l(fractions_a(lfrac))
!    one final term is computed
!      dom_a(ascale) = fractions_a(lfrac)/fractions_a(lfrin)
!      dom_l(ascale) = mapa2l(dom_a(ascale))
!        these are used to correct land fluxes in budgets and lnd2rtm coupling
!        and are particularly important when the land model is running on
!        a different grid than the atm model.  in the old system, this term
!        was treated as effectively 1.0 since there was always a check that
!        fractions_a(lfrac) ~ fractions_a(lfrin), namely that the land model
!        provided a land frac that complemented the ocean grid.  this is
!        no longer a requirement in this new system and as a result, the
!        ascale term can be thought of as a rescaling of the land fractions
!        in the land model to be exactly complementary to the ocean model
!        on whatever grid it may be running.
!  run-time (frac_set):
!    update fractions on ice grid
!      fractions_i(ifrac) = i2x_i(Si_ifrac)  ! ice frac from ice model
!      fractions_i(ofrac) = 1.0 - fractions_i(ifrac)
!        note: the relative fractions are corrected to total fractions
!      fractions_o(ifrac) = mapi2o(fractions_i(ifrac))
!      fractions_o(ofrac) = mapi2o(fractions_i(ofrac))
!      fractions_a(ifrac) = mapi2a(fractions_i(ifrac))
!      fractions_a(ofrac) = mapi2a(fractions_i(ofrac))
!
!  fractions used in merging are as follows
!    mrg_x2a uses fractions_a(lfrac,ofrac,ifrac)
!    mrg_x2o needs to use fractions_o(ofrac,ifrac) normalized to one
!      normalization happens in mrg routine
!
!  fraction corrections in mapping are as follows
!    mapo2a uses *fractions_o(ofrac) and /fractions_a(ofrac)
!    mapi2a uses *fractions_i(ifrac) and /fractions_a(ifrac)
!    mapl2a uses *fractions_l(lfrin) and /fractions_a(lfrin)
!    mapa2* should use *fractions_a(afrac) and /fractions_*(afrac) but this
!      has been defered since the ratio always close to 1.0
!
!  budgets use the standard afrac, ofrac, ifrac, and lfrac to compute
!    quantities except in the land budget which uses lfrin multiplied
!    by the scale factor, dom_l(ascale) to compute budgets.
!
!  fraction and domain checks
!    initialization:
!      dom_i = mapo2i(dom_o)  ! lat, lon, mask, area
!      where fractions_a(lfrac) > 0.0, fractions_a(lfrin) is also > 0.0
!         this ensures the land will provide data everywhere the atm needs it
!         and allows the land frac to be subtlely different from the
!         land fraction specified in the atm.
!      dom_a = mapl2a(dom_l)  ! if atm/lnd same grids
!      dom_a = mapo2a(dom_o)  ! if atm/ocn same grids
!      dom_a = mapi2a(dom_i)  ! if atm/ocn same grids
!      0.0-eps < fractions_*(*) < 1.0+eps
!      fractions_l(lfrin) = fractions_l(lfrac)
!        only if atm/lnd same grids (but this is not formally required)
```

```
!        this is needed until dom_l(ascale) is sent to the land model
!        as an additional field for use in l2r mapping.
!    run time:
!      fractions_a(lfrac) + fractions_a(ofrac) + fractions_a(ifrac) ~ 1.0
!      0.0-eps < fractions_*(*) < 1.0+eps
```

Domain Checking

Domain checking is a very important initialization step in the system. Domain checking verifies that the longitudes, latitudes, areas, masks, and fractions of different grids are consistent with each other. The subroutine that carries out domain checking is in `$CIMEROOT/driver_cpl/driver/seq_domain_mct`. Tolerances for checking the domains can be set in the drv_in driver namelist via the namelist variables, `eps_frac`, `eps_amask`, `eps_agrid`, `eps_aarea`, `eps_omask`, `eps_ogrid`, and `eps_oarea`. These values are derived in the coupler namelist from the script env variables, EPS_FRAC, EPS_AMASK, EPS_AGRID, EPS_AAREA, EPS_OMASK, EPS_OGRID, and EPS_OAREA in the env_run.xml. If an error is detected in the domain checking, the model will write an error message and abort.

The domain checking is dependent on the grids and in particular, the samegrid input namelist settings. But it basically does the following,

```
ocean/ice grid comparison:
   verifies the grids are the same size
   verifies the difference in longitudes and latitudes is less than eps_ogrid.
   verifies the difference in masks is less than eps_omask
   verifies the difference in areas is less than eps_oarea

atmosphere/land grid comparison (if samegrid_al):
   verifies the grids are the same size
   verifies the difference in longitudes and latitudes is less than eps_agrid.
   verifies the difference in masks is less than eps_amask
   verifies the difference in areas is less than eps_aarea

atmosphere/ocean grid comparison (if samegrid_ao):
   verifies the grids are the same size
   verifies the difference in longitudes and latitudes is less than eps_agrid.
   verifies the difference in masks is less than eps_amask
   verifies the difference in areas is less than eps_aarea

fractions
   verifies that the land fraction on the atmosphere grid and the ocean fraction
   on the atmosphere grid add to one within a tolerance of eps_frac.
```

There are a number of subtle aspects in the domain checking like whether to check over masked grid cells, but these issues are less important than recognizing that errors in the domain checking should be treated seriously. It is easy to make the errors go away by changing the tolerances, but by doing so, critical grid errors that can impact conservation and consistency in a simulation might be overlooked.

Mapping (Interpolation)

Mapping files to support interpolation of fields between grids are computed offline. This is done using the ESMF offline regridding utility. First, note that historically, the ocean grid has been the higher resolution grid. While that is no longer always the case, the current implementation largely reflects that presumption. In general, mapping of fluxes is done using a locally conservative area average approach to preserve conservation. State fields are generally mapped using bilinear interpolation from the atmosphere grid to the ocean grid to better preserve gradients, but state fields are generally mapped using the conservative area average approach from the ocean grid to the atmosphere grid. But this is not a requirement of the system. The individual state and flux mapping files are specified at runtime using the `seq_maps.rc` input file, and any valid mapping file using any mapping approach can be specified in that input file.

The `seq_maps.c` file contains information about the mapping files as well as the mapping type. There are currently two types of mapping implementations, "X" and "Y".

- "X" mapping rearranges the source data to the destination grid decomposition and then a local mapping is done from the source to the destination grid on the destination decomposition. In the "X" type, the source grid is rearranged.
- "Y" mapping does a local mapping from the source grid to the destination grid on the source grid decomposition. That generates a partial sum of the destination values which are then rearranged to the destination decomposition and summed. Both options produce reasonable results, although they may differ in value by "roundoff" due to differences in order or operations. The type chosen impacts performance. In both implementations, the number of flops is basically identical. The difference is the communication. In the "Y" type, the destination grid is rearranged.

Since historically, the ocean grid is higher resolution than the atmosphere grid, "X" mapping is used for atmosphere to ocean/ice mapping and "Y" mapping is used from ocean/ice to atmosphere mapping to optimize mapping performance.

Mapping corrections are made in some cases in the polar region. In particular, the current bilinear and area conservative mapping approaches introduce relatively large errors in mapping vector fields around the pole. The current coupler can correct the interpolated surface wind velocity near the pole when mapping from the atmosphere to the ocean and ice grids. There are several options that correct the vector mapping and these are set in the env variable VECT_MAP. The npfix option only affects ocean and ice grid cells that are northward of the last latitude line of the atmospheric grid. The algorithm is contained in the file models/drv/driver/map_atmocn_mct.F90 and is only valid when the atmosphere grid is a longitude/latitude grid. This feature is generally on by default. The other alternative is the cart3d option which converts the surface u and v velocity to 3d x,y,z vectors then maps those three vectors before coverting back to u and v east and north directions on the surface. Both vector mapping methods introduce errors of different degrees but are generally much better than just mapping vector fields as if they were individual scalars. The `vect_map` namelist input is set in the `drv_in` file.

The input mapping files are assumed to be valid for grids with masks of value zero or one where grid points with a mask of zero are never considered in the mapping. Well defined, locally conservative area mapping files as well as bilinear mapping files can be generated using this masked approach. However, there is another issue which is that a grid fraction in an active cell might actually change over time. This is not the case for land fraction, but it is the case for relative ice and ocean fractions. The ice fraction is constantly evolving in the system in general. To improve the accuracy of the ice and ocean mapping, the ocean/ice fields are scaled by the local fraction before mapping and unscaled by the mapped fraction after mapping. The easiest way to demonstate this is via an example. Consider a case where two ice cells of equal area underlie a single atmosphere cell completely. The mapping weight of each ice cell generated offline would be 0.5 in this case and if ice temperatures of -1.0 and -2.0 in the two cells respectively were mapped to the atmosphere grid, a resulting ice temperature on the atmosphere grid of -1.5 would result. Consider the case where one cell has an ice fraction of 0.3 and the other has a fraction of 0.5. Mapping the ice fraction to the atmospheric cell results in a value of 0.4. If the same temperatures are mapped in the same way, a temperature of -1.5 results which is reasonable, but not entirely accurate. Because of the relative ice fractions, the weight of the second cell should be greater than the weight of the first cell. Taking this into account properly results in a fraction weighted ice temperature of -1.625 in this example. This is the fraction correction that is carried out whenever ocean and ice fields are mapped to the atmosphere grid. Time varying fraction corrections are not required in other mappings to improve accuracy because their relative fractions remain static.

Area Correction of Fluxes

To improve conservation in the system, all fluxes sent to and received from components are corrected for the area differences between the components. There are many reasonable ways to compute an area of a grid cell, but they are not generally consistent. One assumption with respect to conservation of fluxes is that the area acting upon the flux is well defined. Differences in area calculations can result in differences of areas up to a few percent and if these are not corrected, will impact overall mass and heat conservation. Areas are extracted for each grid from the mapping files. In this implementation, it is assumed that the areas in all mapping files are computed reasonably and consistently for each grid and on different grids. Those mapping areas are used to correct the fluxes for each component by scaling the fluxes sent to and received by the component by the ratio of the mapping area and the component area. The areas from the components are provided to the coupler by the component at initialization. The minimum and maximum value of each area corrections is written to the coupler log file at initialization. One critical point is that if mapping files are generated by different tools offline and used in the driver, an error could be introduced that is related to inconsistent areas provided by different mapping files.

Initialization and Restart

The initialization has been developed over the last two decades to meet the scientific goals, minimize the communication required, and ensure a consistent and well defined climate system. The order of operations is critical. The initialization is basically as follows:

- The `ccsm_pes` namelist is read and mpi communicators are initialized.
- The `seq_infodata` namelist is read and configuration settings are established.
- The `prof_inparm` namelist is read and the timing tool is initialized.
- The `pio_inparm` namelist is read and the driver IO is initialized.
- The `seq_timemgr` namelist is read and the driver time manager and clocks are initialized.
- The atmosphere init routine is called, the mpi communicator and clock are sent, and the atmosphere grid is returned.
- The land init routine is called, the mpi communicator and clock are sent, and the land grid is returned.
- The runoff init routine is called, the mpi communicator and clock are sent, and the runoff grid is returned.
- The ocean init routine is called, the mpi communicator and clock are sent, and the ocean grid is returned.
- The ice init routine is called, the mpi communicator and clock are sent, and the ice grid is returned.
- The land ice init routine is called, the mpi communicator and clock are sent, and the land ice grid is returned.
- The infodata buffer is synchronized across all processors. This buffer contains many model configuration settings set by the driver but also sent from the components.
- The atmosphere, land, runoff, ice, land ice, and ocean rearrangers are initialized. - These rearrangers move component data between the component pes and the coupler pes.
- The Remaining attribute datatypes associated are initialized
- The mapping weights and areas are read.
- The Component grids are checked using the domain checking method.
- The flux area corrections are initialized on the component pes and applied to the initial fields sent by each component on the component pes. Those initial fields are then rearranged to the coupler pes.
- The fractions are initialized on the coupler pes.
- The atmosphere/ocean flux computation is initialized and initial ocean albedos are computed on the coupler pes.
- The land, ocean, and ice initial albedos are mapped to the atmosphere grid and merged to generate initial surface albedos.
- The initial atmosphere forcing data (albedos) is rearranged from the coupler pes to the atmosphere pes, and the area corrections are applied.
- The second phase of the atmosphere init method is to initialize the atmosphere radiation from the surface albedos.
- The new atmosphere initial data is area corrected and rearranged to the coupler pes.
- The budget diagnostics are zeroed out.
- The coupler restart file is read.
- Initialization is complete.

Driver Threading Control

OpenMP thread counts are controlled at three levels.

- The coarsest level is prior to launching the model. The environment variable OMP_NUM_THREADS is usually set to the largest value any mpi task will use. At a minimum, this will ensure threading is turned on to the maximum desired value in the run.
- The next level is during the driver initialization phase. When the mpi communicators are initialized, the maximum number of threads per mpi task can be computed based on the ccsm_pes namelist input. At that

point, there is an initial fortran call to the intrinsic, omp_set_num_threads. When that happens and if that call is successful, the number of threads will be set to the maximum needed in the system on an mpi task by task basis.

- Finally, there is the ability of CESM to change the thread count per task as each component is individually called and as the model integrates through the driver run loop. In other words, for components that share the same hardware processor but have different threads per task, this feature allows those components to run with the exact value set by the user in the ccsm_pes namelist. This final level of thread control is turned off by default, but it can be turned on using the driver namelist variable `drv_threading`.

This fine control of threading is likely of limited use at this point given the current driver implementation.

Bit-for-bit flag

The driver namelist variable `bfbflag` provides the option of preserving bit-for-bit results on different coupler processor counts. This flag has no impact on other components and their ability to generate bit-for-bit results on different pe counts. When this flag is set, all mappings become "X" types where the source data is rearranged to the destination processor and then local mapping is carried out. The order of operations of this mapping is independent of the pe count or decomposition of the grids.

The other feature that is changed by the `bfbflag` is the global sum diagnostics.

- When `bfbflag` is set to *.false.*, a partial sum is done on each processors and those partial sums are added together to form a global sum. This is generally not order of operations independent for different pe counts or decompositions.
- When `bfbflag` is set to *.true.*, the global sums are computed by gathering the global field on the root processor and doing an ordered sum there.

History and Restarts

In addition to log files, component models also produce history and restart files. History files are generally netcdf format and contain fields associated with the state of the model. History files are implemented and controlled independently in the component models, although support for monthly average history files is a standard output of most production runs. The driver has a file naming standard for history files which includes the case names, component name, and model date.

All CIME-compliant component models must be able to stop in the middle of a run and then subsequently restart in a bit-for-bit fashion. For most models, this requires the writing of a restart file. The restart file can be any format, although netcdf has become relatively standard, and it should contain any scalars, fields, or information that is required to restart the component model in exactly the same state as when the restart was written and the model was stopped. The expectation in CIME is that a restart of a model run will be bit-for-bit identical and this is regularly tested as part of component model development by running the model 10 days, writing a restart at the end of 5 days, and then restarting at day 5 and comparing the result with the 10 day run. Unlike history files, restart files must be coordinated across different components. The restart frequency is set in the driver time manager namelist by driver namelist variables `restart_option`, `restart_n`, and `restart_ymd`. The driver will trigger a restart alarm in clocks when a coordinated restart is requested. The components are required to check this alarm whenever they are called and to write a restart file at the end of the current coupling period. This method ensures all components are writing restart files at a consistent timestamp. The restart filenames are normally set in a generic rpointer file. The rpointer file evolves over the integration and keeps track of the current restart filenames. When a model is restarted, both the rpointer file and the actual restart file are generally required.

Many models are also able to restart accumulating history files in the middle of an accumulation period, but this is not a current requirement for CIME compliant components. In production, the model is usually started and stopped on monthly boundaries so monthly average history files are produced cleanly. The run length of a CESM1 production run is usually specified using the nmonths or nyears option and restart files are normally written only at the end of the run.

Mass and Heat Budgets

Mass and heat are conserved in the coupler to several digits over centuries. Several steps have been taken to ensure this level of conservation, and these are described in other sections of the document. In addition, efforts have been made to make sure each component is internally conservative with respect to mass and heat.

The budgets can be turned on and off using the namelist variable `do_budgets`. The value of that namelist is set by the `$CASEROOT/env_run.xml` variable, `BUDGETS`.

The driver coupler can diagnose heat and mast budgets at several levels and over different periods. The periods are *instantenous*, *daily average*, *monthly average*, *annual average*, or since the start of the run. The budget output for each of these periods is controlled by the driver namelist variables `budget_inst`, `budget_daily`, `budget_month`, `budget_ann`, `budget_ltann`, and `budget_ltend`. `budget_ltann` and `budget_ltend` are used to write the long term budget at either the end of every year or the end of every run. Other budgets are written at their period interval.

The namelist input is an integer specifying what to write. The budget flags are controlled by `$CASEROOT/env_run.xml` variables `BUDGET_INST`, `BUDGET_DAILY`, `BUDGET_MONTHLY`, `BUDGET_ANNUAL`, `BUDGET_LONGTER_EOY`, and `BUDGET_LONGTERM_STOP` respectively. Valid values are 0, 1, 2, or 3. If 0 is set, no budget data is written. The value 1 generates a net heat and water budget for each component, 2 adds a detailed heat and water budget for each component, and 3 adds a detailed heat and water budget of the different conmponents on the atmosphere grid. Normally values of 0 or 1 are specified. Values of 2 or 3 are generally used only when debugging problems involving conservation.

Multi-instance Functionality

The multi-instance feature allows multiple instances of a given component to run in a single CESM run. This might be useful for data assimilation or to average results from multiple instances to force another model.

The multi-instance implementation is fairly basic at this point. It does not do any averaging or other statistics between multiple instances, and it requires that all prognostic components must run the same multiple instances to ensure correct coupling. The multi-instance feature is set via the `$CASEROOT/env_mach_pes.xml` variables that have an `NINST_` prefix. The tasks and threads that are specified in multi-instance cases are distributed evenly between the multiple instances. In other words, if 16 tasks are requested for each of two atmosphere instances, each instance will run on 8 of those tasks. The `NINST_*_LAYOUT` value should always be set to *concurrent* at this time. Sequential running on multiple instances is still not a robust feature. Multiple instances is a build time setting in env_mach_pes.xml. Multiple instance capabilities are expected to be extended in the future.

More on Driver Namelists

There are a series of driver/coupler namelist input files created by the driver namelist generator `$CIMEROOT/driver_cpl/cime_config/buildnml`. These are

- drv_in
- drv_flds_in
- cpl_modelio.nml, atm_modelio.nml, esp_modelio.nml, glc_modelio.nml, ice_modelio.nml, lnd_modelio.nmlo, ocn_modelio.nml, rof_modelio.nml, wav_modelio.nml
- seq_maps.rc

The `*_modelio.nml` files set the filename for the primary standard output file and also provide settings for the parallel IO library, PIO. The drv_in namelist file contains several different namelist groups associated with general options, time manager options, pe layout, timing output, and parallel IO settings. The seq_maps.rc file specifies the mapping files for the configuration. Note that seq_maps.rc is NOT a Fortran namelist file but the format should be relatively clear from the default settings.

Indices and tables

- **genindex**
- **modindex**
- **search**

Building a Coupled Model with CIME

Introduction

Content to go here:

How to replace an existing cime model with another one,

How to integrate your model in to the cime build/configure system and coupler.

How to work with the CIME-supplied models.

What to do if you want to add another component to the long name.

Indices and tables

- **genindex**
- **modindex**
- **search**

Miscellaneous Tools

In addition to basic infrastructure for a coupled model, CIME contains in its distribution several stand-alone tools that are necessary and/or useful when building a climate model. Guides for using them will be here.

Statistical Ensemble Test

Mapping Tools

cprnc

load-balancing-tool

Indices and tables

- **genindex**
- **modindex**
- **search**

Indices and tables

- **genindex**
- **modindex**
- **search**

CIME is developed by the ACME and CESM projects.