CCSM4.0 User's Guide

Mariana Vertenstein
NCAR

Tony Craig
NCAR

Adrianne Middleton
NCAR

Diane Feddema
NCAR

Chris Fischer
NCAR

CCSM4.0 User's Guide

by Mariana Vertenstein, Tony Craig, Adrianne Middleton, Diane Feddema, and Chris Fischer

Table of Contents

Chapter 1. Introduction

How To Use This Document

This guide instructs both novice and experienced users on building and running CCSM. If you are a new user, we recommend that the introductory sections be read before moving onto other sections or the Quick Start procedure. This document is written so, as much as possible, individual sections stand on their own and the user's guide can be scanned and sections read in a relatively ad hoc order. In addition, the web version provides clickable links that tie different sections together.

The chapters attempt to provide relatively detailed information about specific aspects of CCSM4 like setting up a case, building the model, running the model, porting, and testing. There is also a large section of use cases and a Frequently Asked Questions (FAQ) section.

```
Throughout the document, this presentation style indicates shell
commands and options, fragments of code, namelist variables, etc.
Where examples from an interactive shell session are presented,
lines starting with ">" indicate the shell prompt.
```

Please feel free to provide feedback to CCSM about how to improve the documentation.

CCSM Overview

The Community Climate System Model (CCSM) is a coupled climate model for simulating Earth's climate system. Composed of five separate models simultaneously simulating the Earth's atmosphere, ocean, land, land-ice, and sea-ice, plus one central coupler component, CCSM allows researchers to conduct fundamental research into the Earth's past, present, and future climate states.

The CCSM system can be configured a number of different ways from both a science and technical perspective. CCSM supports several different resolutions and component configurations. In addition, each model component has input options to configure specific model physics and parameterizations. CCSM can be run on a number of different hardware platforms and has a relatively flexible design with respect to processor layout of components. CCSM also supports both an internally developed set of component interfaces and the ESMF compliant component interfaces (See the Section called *Use of an ESMF library and ESMF interfaces* in Chapter 9)

The CCSM project is a cooperative effort among U.S. climate researchers. Primarily supported by the National Science Foundation (NSF) and centered at the National Center for Atmospheric Research (NCAR) in Boulder, Colorado, the CCSM project enjoys close collaborations with the U.S. Department of Energy and the National Aeronautics and Space Administration. Scientific development of the CCSM is guided by the CCSM working groups, which meet twice a year. The main CCSM workshop is held each year in June to showcase results from the various working groups and coordinate future CCSM developments among the working groups. The CCSM website[1] provides more information on the CCSM project, such as the management structure, the scientific working groups, downloadable source code, and online archives of data from previous CCSM experiments.

CCSM Components

CCSM consists of five geophysical models: atmosphere (atm), sea-ice (ice), land (lnd), ocean (ocn), and land-ice (glc), plus a coupler (cpl) that coordinates the models and passes information between them. Each model may have "active," "data," "dead," or "stub" component version allowing for a variety of "plug and play" combinations. The

land-ice component (glc) will not be released as part of CCSM4.0; to satisfy interface requirements, only the stub version will accompany the release.

During the course of a CCSM run, the model components integrate forward in time, periodically stopping to exchange information with the coupler. The coupler meanwhile receives fields from the component models, computes, maps, and merges this information, then sends the fields back to the component models. The coupler brokers this sequence of communication interchanges and manages the overall time progression of the coupled system. A CCSM component set is comprised of six components: one component from each model (atm, lnd, ocn, ice, and glc) plus the coupler. Model components are written primarily in Fortran 90.

The active (dynamical) components are generally fully prognostic, and they are state-of-the-art climate prediction and analysis tools. Because the active models are relatively expensive to run, data models that cycle input data are included for testing, spin-up, and model parameterization development. The dead components generate scientifically invalid data and exist only to support technical system testing. The dead components must all be run together and should never be combined with any active or data versions of models. Stub components exist only to satisfy interface requirements when the component is not needed for the model configuration (e.g., the active land component forced with atmospheric data does not need ice, ocn, or glc components, so ice, ocn, and glc stubs are used).

The CCSM components can be summarized as follows:

Model Type	Model Name	Component Name	Type	Description
atmosphere	atm	cam	active	The Community Atmosphere Model (CAM) is a global atmospheric general circulation model developed from the NCAR CCM3.
atmosphere	atm	datm	data	The data atmosphere component is a pure data component that reads in atmospheric forcing data
atmosphere	atm	xatm	dead	
atmosphere	atm	satm	stub	

Model Type	Model Name	Component Name	Type	Description
land	lnd	clm	active	The Community Land Model (CLM) is the result of a collaborative project between scientists in the Terrestrial Sciences Section of the Climate and Global Dynamics Division (CGD) at NCAR and the CCSM Land Model Working Group. Other principal working groups that also contribute to the CLM are Biogeochemistry, Paleoclimate, and Climate Change and Assessment.
land	lnd	dlnd	data	The data land component differs from the other data models in that it can run as a purely data-runoff model (reading in runoff data) or as a purely data-land model (reading in coupler history data for atm/land fluxes and land albedos produced by a previous run) or both.
land	lnd	xlnd	dead	

Model Type	Model Name	Component Name	Type	Description
land	lnd	slnd	stub	
ocean	ocn	pop	active	The ocean model is an extension of the Parallel Ocean Program (POP) Version 2 from Los Alamos National Laboratory (LANL).
ocean	ocn	docn	data	The data ocean component has two distinct modes of operation. It can run as a pure data model, reading ocean SSTs (normally climatological) from input datasets, interpolating in space and time, and then passing these to the coupler. Alternatively, docn can compute updated SSTs based on a slab ocean model where bottom ocean heat flux convergence and boundary layer depths are read in and used with the atmo-sphere/ocean and ice/ocean fluxes obtained from the coupler.
ocean	ocn	xocn	dead	
ocean	ocn	socn	stub	

Model Type	Model Name	Component Name	Type	Description
sea-ice	ice	cice	active	The sea-ice component (CICE) is an extension of the Los Alamos National Laboratory (LANL) sea-ice model and was developed though collaboration within the CCSM Polar Climate Working Group (PCWG). In CCSM, CICE can run as a fully prognostic component or in prescribed mode where ice coverage (normally climatological) is read in.
sea-ice	ice	dice	data	The data ice component is a partially prognostic model. The model reads in ice coverage and receives atmospheric forcing from the coupler, and then it calculates the ice/atmosphere and ice/ocean fluxes. The data ice component acts very similarly to CICE running in prescribed mode.
sea-ice	ice	xice	dead	
sea-ice	ice	sice	stub	

Model Type	Model Name	Component Name	Type	Description
land-ice	glc	glc	active	
land-ice	glc	sglc	stub	
coupler	cpl	cpl	active	The CCSM4 coupler was built primarily through a collaboration of the NCAR CCSM Software Engineering Group and the Argonne National Laboratory (ANL). The MCT coupling library provides much of the infrastructure. cpl7 is used in CCSM4 and is technically a completely new driver and coupler compared to CCSM3.

CCSM Component Sets

The CCSM components can be combined in numerous ways to carry out various scientific or software experiments. A particular mix of components, *along with* component-specific configuration and/or namelist settings is called a component set or "compset." CCSM has a shorthand naming convention for component sets that are supported out-of-the-box.

The compset name usually has a well defined first letter followed by some characters that are indicative of the configuration setup. Each compset name has a corresponding short name. Users are not limited to the predefined component set combinations. A user may define their own component set.

See the component set table for a complete list of supported compset options. Running **create_newcase** with the option "-list" will also always provide a listing of the supported out-of-the-box component sets for the local version of CCSM4. The first letter of a compset name generally indicates which components are used:

Designation	Components	Details
A	datm,dlnd,dice,docn,sglc	All DATA components with stub glc (used primarily for testing)
B	cam,clm,cice,pop2,sglc	FULLY ACTIVE components with stub glc

Designation	Components	Details
C	datm,dlnd,dice,pop2,sglc	POP active with data atm, lnd(runoff), and ice plus stub glc
D	datm,slnd,cice,docn,sglc	CICE active with data atm and ocean plus stub land and glc
E	cam,clm,cice,docn,sglc	CAM, CLM, and CICE active with data ocean (som mode) plus stub glc
F	cam,clm,cice,docn,sglc	CAM, CLM, and CICE(prescribed mode) active with data ocean (sstdata mode) plus stub glc
G	datm,dlnd,cice,pop2,sglc	POP and CICE active with data atm and lnd(runoff) plus stub glc
H	datm,slnd,cice,pop2,sglc	POP and CICE active with data atm and stub land and glc
I	datm,clm,sice,socn,sglc	CLM active with data atm and stub ice, ocean, and glc
S	satm,slnd,sice,socn,sglc	All STUB components (used for testing only)
X	xatm,xlnd,xice,xocn,sglc	All DEAD components except for stub glc (used for testing only)

CCSM Grids

The grids are specified in CCSM4 by setting an overall model resolution. Once the overall model resolution in set, components will read in appropriate grids files and the coupler will read in appropriate mapping weights files. Coupler mapping weights are always generated externally in CCSM4. The components will send the grid data to the coupler at initialization, and the coupler will check that the component grids are consistent with each other and with the mapping weights files.

In CCSM4, the ocean and ice must be on the same grid, but unlike CCSM3, the atmosphere and land can now be on different grids. Each component determines its own unique grid decomposition based upon the total number of pes assigned to that component.

CCSM4 supports several types of grids out-of-the-box including single point, finite volume, spectral, cubed sphere, displaced pole, and tripole. These grids are used internally by the models. Input datasets are usually on the same grid but in some cases, they can be interpolated from regular lon/lat grids in the data models. The finite volume and spectral grids are generally associated with atmosphere and land models but the data ocean and data ice models are also supported on those grids. The cubed sphere grid is used only by the active atmosphere model, cam. And the displaced pole and tripole grids are used by the ocean and ice models. Not every grid can be run by every component.

CCSM4 has a specific naming convention for individual grids as well as the overall resolution. The grids are named as follows:

- "[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 convection 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 gx1v6 or g16 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. The short name is [agrid]_[lgrid]_[oigrid]. An example is ne30_f19_g16.

The model resolution is specified by setting a combination of these resolutions. In general, the overall resolution is specified in one of the two following ways for resolutions where the atmosphere and land grids are identical or not.

"[algrid]_[oigrid]"

In this grid, the atmosphere and land grid are identical and specified by the value of "algrid". The ice and ocean grids are always identical and specified by "oigrid". For instance, f19_g16 is the finite volume 1.9x2.5 grid for the atmosphere and land components combined with the gx1v6 displaced pole grid for the ocean and ice components.

"[agrid]_[lgrid]_[oigrid]" or "[agrid][lgrid]_[oigrid]" (for short names)

In this case, the atmosphere, land, and ocean/ice grids are all unique. For example ne30_f19_g16 is the cubed sphere ne30np4 atmospheric grid running with the finite volume 1.9x2.5 grid for the land model combined with the gx1v6 displaced pole grid running on the ocean and ice models.

For a complete list of currently supported grid resolutions, please see the supported resolutions table.

The ocean and ice models run on either a Greenland dipole or a tripole grid (see figures). The Greenland Pole grid is a latitude/longitude grid, with the North Pole displaced over Greenland to avoid singularity problems in the ocn and ice models. Similarly, the Poseidon tripole grid (http://climate.lanl.gov/Models/POP/) is a latitude/longitude grid with three poles that are all centered over land.

Greenland Pole Grid

Poseidon Tripole Grid

CCSM Machines

Scripts for supported machines, prototype machines and generic machines are provided with the CCSM4 release. Supported machines have machine specific files and settings added to the CCSM4 scripts and are machines that should run CCSM cases out-of-the-box. Machines are supported in CCSM on an individual basis and are usually listed by their common site-specific name. To get a machine ported and functionally supported in CCSM, local batch, run, environment, and compiler information must be configured in the CCSM scripts. Prototype machines are machines in the CCSM user community that CCSM has been ported to and the machine specific files and settings were provided by the user community. Prototype machines all start with the prefix prototype_. These machines may not work out-of-the-box, however, to the best of NCAR's knowledge these machine specific files and settings worked at one time. Generic machine generally refers more to classes of machines, like IBM AIX or a linux cluster with an intel compiler, and the generic machine names in CCSM4 all start with the generic_ prefix. Generic machines require that a user provide some settings via command line options with create_newcase and then some additional effort will generally be required to get the case running. Generic machines are handy for quickly getting a case running on a new platform, and they also can accelerate the porting process. For more information on porting, see Chapter 7. To get a list of current machines in each of these categories (supported, prototype and generic) run script create_newcase with option -list from the $CCSMROOT directory.

The list of available machines are documented in CCSM machines . Running create_newcase with the "-list" option will also show the list of available machines for the current local version of CCSM.4 Supported machines have undergone the full CCSM porting process. A prototype machine is provided by the user community and may not work out-of-the-box, but it is a good starting point for a port to a new machine of the same type. A generic machine is provided as a starting point for new users to introduce a machine that does not belong to any of the above categories. The machines available in each of these categories changes as access to machines change over time.

CCSM Validation

Although CCSM4.0 can be run out-of-the-box for a variety of resolutions, component combinations, and machines, MOST combinations of component sets, resolutions, and machines have not undergone rigorous scientific climate validation.

Long control runs are being carried out, and these will be documented in future versions of this guide, located at http://www.ccsm.ucar.edu/models/ccsm4.0. Model output from these long control runs will accompany future releases, and these control runs should be scientifically reproducible on the original platform or other platforms. Bit-for-bit reproducibility cannot be guaranteed due to variations in compiler or system versions.

Users should carry out your own validations on any platform prior to doing scientific runs or scientific analysis and documentation.

CCSM Software/Operating System Prerequisites

The following are the Operating System requirements and additional software requirements for installing and running CCSM 4.0. The recommended versions are known to work; however, we encourage the CCSM user community to test other versions of the required software packages based on local availability.

- Operating System: Linux, AIX 5.3.9.X
- Software: Required packages for installing and running CCSM 4.0 are given in the table below. The table lists recommended versions for required packages not

included in the CCSM release. In addition to these external packages, required software packages PIO and MCT are included in the CCSM 4.0 release and build.

Table 1-1. Required Software Packages not included in the CCSM 4.0 release

Machine	Software Package Name	Version Recommendations
Cray XT Series	PGI Fortran	pgf95 9.0.x
Cray XT Series	PGI C	pgcc
Cray XT Series	MPI	MPT 3.5.1
IBM Power Series	IBM Fortran	xlf 12.1.0.6
IBM Power Series	IBM C	xlC 10.0.0.3
IBM Power Series	MPI	POE 5.1.1.3 (IBM's proprietary implementation of MPI is a component of POE)
Generic Linux Machine	Intel Fortran	ifort (intel64) 3.2.02 or ifort (intel64) 10.1.018
Generic Linux Machine	Intel C	icc 10.1.018
Generic Linux Machine	MPI	OpenMPI or vendor provided MPI, OpenMPI 1.2.8
All machines	netCDF or pnetCDF	netcdf 3.6.0, netcdf 3.6.2, netcdf 3.6.3 note: use same fortran compiler for netCDF as you use for CCSM 4.0
All machines	ESMF	4.0.0rp1 note: this is an optional external library

Note: CCSM may not compile with pgi compilers prior to release 9.0.x. PGI Fortran Version 7.2.5 aborts with an internal compiler error when compiling CCSM4.0, specifically POP.

Caution

NetCDF must be built with the same Fortran compiler as CCSM. In the netCDF build the FC environment variable specifies which Fortran compiler to use. CCSM is written mostly in Fortran, netCDF is written in C. Because there is no standard way to call a C program from a Fortran program, the Fortran to C layer between CCSM and netCDF will vary depending on which Fortran compiler you use for CCSM. When a function in the netCDF library is called from a Fortran application, the netCDF Fortran API calls the netCDF C library. If you do not use the same compiler to build netCDF and CCSM you will in most cases get errors from netCDF saying certain netCDF functions cannot be found.

Parallel-netCDF, also referred to as pnetcdf, is optional. Pnetcdf version 1.1.1. or later is requried for CCSM 4.0. It is a library that is file-format compatible with netCDF, and provides higher performance by using MPI-IO. Pnetcdf is turned on inside pio by setting the PNETCDF_PATH variable in the pio CONFIG_ARGS in the Macros.$MACH file. You must also specify that you want pnetcdf at runtime via the io_type argument that can be set to either "netcdf" or "pnetcdf" for each component. Note: testing of pnetcdf with CCSM4.0 is limited, and its use is not recommended in most cases.

Downloading CCSM

Downloading the code and scripts

CCSM release code will be made available through a Subversion repository. Access to the code will require Subversion client software in place that is compatible with our Subversion server software, such as a recent version of the command line client, svn. Currently, our server software is at version 1.4.2. We recommend using a client at version 1.5 or later, though older versions may suffice. We cannot guarantee a client older than 1.4.2. For more information or to download open source tools, visit:

`http://subversion.tigris.org/`[2]

With a valid svn client installed on the machine where CCSM4 will be built and run, the user may download the latest version of the release code. First view the available release versions with the following command:

```
> svn list https://svn-ccsm-release.cgd.ucar.edu/model_versions
```

When contacting the Subversion server for the first time, the following certificate message will likely be generated:

```
Error validating server certificate for 'https://svn-ccsm-release.cgd.ucar.edu:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
 - The certificate hostname does not match.
 - The certificate has expired.
Certificate information:
 - Hostname: localhost.localdomain
 - Valid: from Feb 20 23:32:25 2008 GMT until Feb 19 23:32:25 2009 GMT
 - Issuer: SomeOrganizationalUnit, SomeOrganization, SomeCity, SomeState, --
 - Fingerprint: 86:01:bb:a4:4a:e8:4d:8b:e1:f1:01:dc:60:b9:96:22:67:a4:49:ff
(R)eject, accept (t)emporarily or accept (p)ermanently?
```

After accepting the certificate, the repository will request a username and password. Be aware that the request may set current machine login id as the default username. Once correctly entered, the username and password will be cached in a protected subdirectory of the user's home directory so that repeated entry of this information will not required for a given machine.

The release tags should follow a recognizable naming pattern, and they can be checked out from the central source repository into a local sandbox directory. The following example shows how to checkout model version ccsm4.0_a01:

```
> svn co https://svn-ccsm-release.cgd.ucar.edu/model_versions/ccsm4_0_a01 \
         ccsm4_working_copy
```

Caution

If a problem was encountered during checkout, which may happen with an older version of the client software, it may appear to have downloaded successfully, but in fact only a partial checkout has occurred. To ensure a successful download, make sure the last line of svn output has the following statement:

```
Checked out revision XXXXX.
```

Or, in the case of an 'svn update' or 'svn switch':

```
Updated to revision XXXXX.
```

This will create a directory called `ccsm4_working_copy` that can be used to modify, build, and run the model. The following Subversion subcommands can be executed in the working sandbox directory.

For various information regarding the release version checked out...

```
> svn info
```

For a listing of files that have changed since checkout...

```
> svn status
```

For a description of the changes made to the working copy...

```
> svn diff
```

Obtaining new release versions of CCSM

To update to a newer version of the release code you can download a new version of CCSM 4.0 from the svn central source repository in the following way:

Suppose for example that a new version of ccsm4.0 is available at https://svn-ccsm-release.cgd.ucar.edu/model_versions/ccsm4_0_<newversion>. You can download it in the same way you downloaded the first CCSM 4.0 version. For directions on downloading the code, please refer to the ccsm4 User's Guide at http://www.ccsm.ucar.edu/models/ccsm4.0/ccsm_doc/book1.html

As an alternative, some users may find the svn switch operation useful. In particular, if you've used svn to check out the previous release, ccsm4_0_<previousversion>, and if you've made modifications to that code, you should consider using the svn switch operation. This operation will not only upgrade your code to the version ccsm4_0_<newversion>, but will also attempt to reapply your modifications to the newer version.

How to use the svn switch operation:

Suppose you've used svn to check out ccsm4_0_<previousversion> into the directory called /home/user/ccsm4_0

1. Make a backup copy of /home/user/ccsm4_0 -- this is important in case you encounter any problems with the update
2. cd to the top level of your ccsm4_0 code tree...

   ```
   > cd /home/user/ccsm4_0
   ```

3. Issue the following svn command...

   ```
   > svn switch https://svn-ccsm-release.cgd.ucar.edu/model_versions/ccsm4_0_<newver
   ```

The svn switch operation will upgrade all the code to the new ccsm4_0_<newversion> version, and for any files that have been modified, will attempt to reapply those modifications to the newer code.

Note that an update to a newer version of the release code may result in conflicts with modified files in the local working copy. These conflicts will likely require that differences be resolved manually before use of the working copy may continue. For help in resolving svn conflicts, please visit the subversion website,

http://svnbook.red-bean.com/en/1.5/svn.tour.cycle.html#svn.tour.cycle.resolve[3]

A read-only option is available for users to view via a web browser at

https://svn-ccsm-release.cgd.ucar.edu[4]

where the entire CCSM4 release directory tree can be navigated.

The following examples show common problems and their solutions.

Problem 1: If the hostname is typed incorrectly:

```
> svn list https://svn-ccsm-release.cgd.ucar.edu/model_versions/ccsm4_0_<version>
svn: PROPFIND request failed on '/model_versions/ccsm4_0_<version>'
svn: PROPFIND of '/model_versions/ccsm4_0_<version>': Could not resolve hostname 'svn-c
```

Problem 2: If http is typed instead of https:

```
> svn list http://svn-ccsm-release.cgd.ucar.edu/model_versions/ccsm4_0_<version>
svn: PROPFIND request failed on '/model_versions/ccsm4_0_<version>'
svn: PROPFIND of '/model_versions/ccsm4_0_<version>': could not connect to server (http
```

Downloading input data

Input datasets are needed to run the model. CCSM input data will be made available through a separate Subversion input data repository. The username and password for the input data repository will be the same as for the code repository.

> **Note:** The input data repository contains datasets for many configurations and resolutions and is well over 1 TByte in total size. DO NOT try to download the entire dataset.

Datasets can be downloaded on a case by case basis as needed and CCSM now provides tools to check and download input data automatically.

A local input data directory should exist on the local disk, and it also needs to be set in the CCSM scripts via the variable $DIN_LOC_ROOT_CSMDATA. For supported machines, this variable is preset. For generic machines, this variable is set as an argument to **create_newcase**. Multiple users can share the same $DIN_LOC_ROOT_CSMDATA directory.

The files in the subdirectories of $DIN_LOC_ROOT_CSMDATA should be write-protected. This prevents these files from being accidentally modified or deleted. The directories in $DIN_LOC_ROOT_CSMDATA should generally be group writable, so the directory can be shared among multiple users.

As part of the process of generating the CCSM executable, the utility, **check_input_data** is called, and it attempts to locate all required input data for the case based upon file lists generated by components. If the required data is not found on local disk in $DIN_LOC_ROOT_CSMDATA, then the data will be downloaded automatically by the scripts or it can be downloaded by the user by invoking **check_input_data** with the -export command argument. If you want to download the input data manually you should do it before you build CCSM.

It is possible for users to download the data using svn subcommands directly, but use of the **check_input_data** script is highly recommended to ensure that only the required datasets are downloaded. Again, users are STRONGLY DISCOURAGED from downloading the entire input dataset from the repository due to the size.

Quick Start (CCSM Workflow)

The following quick start guide is for versions of CCSM that have already been ported to the local target machine. If CCSM has not yet been ported to the target machine, please see Chapter 7. If you are new to CCSM4, please consider reading the introduction first

These definitions are required to understand this section:

- $COMPSET refers to the component set.
- $RES refers to the model resolution.
- $MACH refers to the target machine.
- $CCSMROOT refers to the CCSM root directory.
- $CASE refers to the case name.
- $CASEROOT refers to the full pathname of the root directory where the case ($CASE) will be created.
- $EXEROOT refers to the executable directory. ($EXEROOT is normally NOT the same as $CASEROOT).
- $RUNDIR refers to the directory where CCSM actually runs. This is normally set to $EXEROOT/run.

This is the procedure for quickly setting up and running a CCSM case.

1. Download CCSM (see Download CCSM).
2. Select a machine, a component, and a resolution from the list displayed after invoking this command:

   ```
   > cd $CCSMROOT/scripts
   > create_newcase -list
   ```

   See the component set table for a complete list of supported compset options.

   See the resolution table for a complete list of model resolutions.

   See the machines table for a complete list of machines.
3. Create a case.

   The **create_newcase** command creates a case directory containing the scripts and xml files to configure a case (see below) for the requested resolution, component set, and machine. **create_newcase** has several required arguments and if a generic machine is used, several additional options must be set (invoke create_newcase -h for help).

   If running on a supported machine, ($MACH), then invoke **create_newcase** as follows:

   ```
   > create_newcase -case $CASEROOT \
            -mach $MACH \
            -compset $COMPSET \
            -res $RES
   ```

   If running on a new target machine, see Chapter 7.
4. Configure the case.

   Issuing the **configure** command creates component namelists and machine specific build and run scripts. Before invoking **configure**, modify the case settings in $CASEROOT as needed for the experiment.

   a. **cd** to the $CASEROOT directory.

      ```
      > cd $CASEROOT
      ```

   b. Modify configuration settings in `env_conf.xml` and/or in `env_mach_pes.xml` (optional). (Note: To edit any of the env xml files, use the xmlchange command. invoke **xmlchange** -h for help.)

   c. Invoke the **configure** command.

      ```
      > configure -case
      ```
5. Build the executable.

   a. Modify build settings in `env_build.xml` (optional).

b. Run the build script.

```
> $CASE.$MACH.build
```

6. Run the case.

   a. Modify runtime settings in `env_run.xml` (optional). In particular, set the DOUT_S variable to FALSE.

   b. Submit the job to the batch queue. This example uses a submission command for a Cray computer:

   ```
   > qsub $CASE.$MACH.run
   ```

7. When the job is complete, review the following directories and files

   a. $RUNDIR. This directory is set in the env_build.xml file. This is the location where CCSM was run. There should be log files there for every component (ie. of the form cpl.log.yymmdd-hhmmss). Each component writes its own log file. Also see whether any restart or history files were written. 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 ".

   b. $CASEROOT/logs. The log files should have been copied into this directory if the run completed successfully.

   c. $CASEROOT. There could be a standard out and/or standard error file.

   d. $CASEROOT/CaseDocs. The case namelist files are copied into this directory from the $RUNDIR.

   e. $CASEROOT/timing. There should be a couple of timing files there that summarize the model performance.

   f. $DOUT_S_ROOT/$CASE. This is the archive directory. If DOUT_S is FALSE, then no archive directory should exist. If DOUT_S is TRUE, then log, history, and restart files should have been copied into a directory tree here.

Notes

1. http://www.ccsm.ucar.edu/
2. http://subversion.tigris.org
3. http://svnbook.red-bean.com/en/1.5/svn.tour.cycle.html#svn.tour.cycle.resolve
4. https://svn-ccsm-release.cgd.ucar.edu

Chapter 2. Creating a Case

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

How to create a new case

In what follows, $CCSMROOT is the full pathname of the root directory of your ccsm source code. First use the -h option to document the **create_newcase** options. Then use the -l option to determine which component sets, resolutions, and machines are supported.

```
> cd $CCSMROOT/scripts
> create_newcase -h
> create_newcase -l
```

As explained in ccsm compsets, a component set (compset) defines the specific model components that will be used in a given CCSM configuration, along with any component-specific namelist or configuration settings that are specific to this configuration. See the component set table for a complete list of supported compset options. If you want to create a custom compset, create an appropriate xml compset file and use the create_newcase option -compset_file on the create_newcase command line. For more information, see the frequently asked quesitons (FAQ) section How do I create my own compset?

See the grids table for a complete list of supported grids options.

See the machines table for a complete list of machines.

> **Note:** CCSM component sets and resolutions have both short and long names. Either the short or long name can be entered as input to **create_newcase**. As an example, the component set B_1850_RAMPCO2_CN has the short name B1850RMCN. Similarly, the resolution, 0.9x2.5_gx1v6 has the short name f09_g16. Both the long and short names appear in the output from **create_newcase** -l, where the short name always appears in parentheses.

For a generic machine, **create_newcase** can be invoked with the following arguments:

```
> create_newcase -case [case name] \
           -mach [machine name] \
           -compset [compset name] \
           -res [resolution] \
           -scratchroot [executable root directory] \
           -din_loc_root_csmdata [input data root directory] \
           -max_tasks_per_node [max mpi tasks per node] \
           [-pes_file [user-defined pes-setup file]] \
           [-compset_file [user-defined compset file]] \
           [-pecount [S, M, L, X1, or X2]] \
           [-silent] [-verbose] \
           [-xmlmode normal/expert]
```

For a non-generic machine, **create_newcase** can be invoked with the following arguments:

```
> create_newcase -case [case name] \
           -mach [machine name] \
           -compset [compset name] \
           -res [resolution] \
           [-pes_file [user-defined pes-setup file]] \
           [-compset_file [user-defined compset file]] \
           [-pecount [S, M, L, X1, or X2]] \
           [-silent] [-verbose] \
```

```
[-xmlmode normal/expert]
```

Note: -case, -mach, -compset and -res are required arguments to **create_newcase**. In addition, -scratchroot, -din_loc_root_csmdata and -max_tasks_per_node are additional required arguments when a generic machine is targeted.

If you want to use your own pes setup file, specify the full pathname of that file for the optional -pes_file argument. The sample pes_file format is provided at `$CCSMROOT/sample_pes_file.xml`.

Here is a simple example of using **create_newcase** for a non-generic machine.

```
> cd $CCSMROOT/scripts
> create_newcase -case ~/ccsm4/b40.B2000 \
                 -compset B_2000 \
                 -res 0.9x1.25_gx1v6 -mach bluefire
```

This example creates a $CASEROOT directory `~/ccsm4/b40.B2000` where $CASE is b40.B2000 with a model resolution of 0.9x1.25_gx1v6 (a 1-degree atmosphere/land grid with a nominal 1-degree ocean/ice grid using the gx1v6 ocean mask). The component set B_2000 uses fully active components configured to produce a present-day simulation.

Note: The complete example appears in the basic example. $CASE can include letters, numbers, ".", and "_". Note that **create_newcase** creates the $CASEROOT directory. If the directory already exists, it prints a warning and aborts.

create_newcase creates the directory $CASEROOT, which is specified by the -case option. In $CASEROOT, **create_newcase** installs the files and directories that are responsible for configuring, building, and running the case. For example, the above command creates the following files and directories in `~/ccsm4/b40.B2000/`. (Note that user-modifiable files/directories appear in italics.)

Directory or Filename	Description
LockedFiles/	A directory that holds copies of files that should not be changed.
Macros.bluefire	Contains machine-specific makefile directives. In the current release, the Macros have been organized into groups of machine-dependent files each containing site-specific and machine-specific options.
README/	A directory of README files for the components.
README.case	A file detailing the **create_newcase** usage in creating your case. This is a good place to keep track of runtime problems and changes.
SourceMods	A directory where users can place modified source code.
Tools/	A directory containing support utility scripts. Users should never need to access the contents of this directory.

Directory or Filename	Description
check_input_data	A script that checks for various input datasets and moves them into place.
configure	A script used to configure your case.
create_production_test	A script used to create a test of your case.
env_build.xml	Controls model build settings (see customizing a build).
env_case.xml	Sets model components and resolution. This file *cannot* be modified after a case has been created. To make changes, re-run **create_newcase** with different options.
env_conf.xml	Controls general settings including run initialization type (see the Section called *Setting the case initialization type* in Chapter 3), coupler mapping files, component configuration, and namelist generation. Sets environment variables that are used by the component template scripts to generate component namelist and build scripts (see customizing components).
env_mach_pes.xml	Controls component machine-specific processor layout (see the Section called *Setting the case PE layout* in Chapter 3). The settings in this are critical to a well-load-balanced simulation (see loadbalancing a run).
env_mach_specific	A file used to set a number of machine-specific environment variables for building and/or running. This file can be edited at any time. However, build environment variables should not be edited after a build is invoked.
env_run.xml	Controls run-time settings such as length of run, frequency of restarts, output of coupler diagnostics, and short-term and long-term archiving. See running a case.
xmlchange	A script used to modify values in the xml files.

For more complete information about the files in the case directory, see the Section called *What are all these directories and files in my case directory?* in Chapter 11

Note: Since default values are provided for the above xml file variables, you could now go to configuring a case and configure your case. However, you should first understand what variables you might want to change and how these xml variables are used by the scripts. Please continue reading below if you are a new user.

The xml variables in the `env_*.xml` files are translated into csh environment variables with the same name by the script **Tools/ccsm_getenv**. Conversion of xml variables to environment variables is used by numerous script utilities as part of con-

figuring, building, and running a given case. It is important to note that you do not explicitly see this conversion.

Note: Users can only modify the xml variables. Users cannot modify the csh environment variables directly.

Complete lists of CCSM environment variables in the xml files that appear in $CASEROOT are provided in env_case.xml variables, env_conf.xml variables , env_mach_pes.xml variables , env_build.xml variables , and env_run.xml variables.

Modifying an xml file

Users can edit the xml files directly to change the variable values. However, modification of variables in the xml scripts is best done using the **xmlchange** script in the $CASEROOT directory since it performs variable error checking as part of changing values in the xml files. To invoke the xmlchange script:

```
xmlchange -file [name] -id [name] -val [name] -help -silent -verbose -file
```

`-file`

The xml file to be edited.

`-id`

The xml variable name to be changed.

`-val`

The intended value of the variable associated with the -id argument.

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 "'".

`-silent`

Turns on silent mode. Only fatal messages will be issued.

`-verbose`

Echoes all settings made by configure.

`-help`

Print usage info to STDOUT.

Cloning a case (Experts only)

This is an advanced feature provided for expert users. If you are a new user, skip this section.

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** either from $CCSMROOT or from the directory where you want the new case to be created. It has the following arguments:

`-case`

The name or path of the new case.

`-clone`

The full pathname of the case to be cloned.

`-silent`

Enables silent mode. Only fatal messages will be issued.

`-verbose`

Echoes all settings.

`-help`

Prints usage instructions.

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

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

When invoking **create_clone**, the following files are cloned in the new $CLONEROOT case directory. Note that the new case directory will be identical to the cloned case directory *except* for the original cloned scripts $CASEROOT.$MACH.build, $CASEROOT.$MACH.clean_build, $CASEROOT.$MACH.run, and $CASEROOT.$MACH.l_archive, which will have new names in the new case.

> **Important::** Do not change anything in the `env_case.xml` file. In addition, if you want to modify `env_conf.xml`, the new case will no longer be a clone, and you will need to configure -cleanall, which removes all files associated with all previous invocations of the configure script. 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. The `Buildconf/` directory will be removed, however. As a result, any changes to the namelist generation scripts in `Buildconf/` will *not* be preserved. Before invoking this command, make backup copies of your "resolved" component namelists in the `Buildconf/` directory if modifications to the generated scripts were made.

Another approach to duplicating a case is to use the information in that case's README.case file to create a new case. Note that this approach 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.

Chapter 3. Configuring a Case

Configure Overview

Configure generates buildnml and buildexe scripts for each component in the Buildconf directory. It also generates build, run, l_archive, and clean_build scripts in the CASEROOT directory. These scripts generate namelist for components and build and run the CCSM4 model.

configure (invoked with the -case option) uses variables in env xml files to generate a new `Buildconf/` directory and several new files in $CASEROOT.

> **Note:** Any user modifications to `env_conf.xml` and `env_mach_pes.xml` must be done *before* **configure** is invoked. In the simplest case, **configure** can be run *without* modifying either of these files and default settings will be then be used.

Before exploring the details of **configure**, it is important to understand the concept of locked env files. The env files are "locked" after the variables have been used by other parts of the system and cannot be changed. The scripts do this by "locking" a file and not permitting the user to modify that file. More information on locking files can be found in the Section called *Why is there file locking and how does it work?* in Chapter 11

The configure command must be run in the $CASEROOT directory and must be invoked with one of the following options:

```
configure [-help] [-case] \
          [-cleanmach] [-cleannamelist] [-cleanall]
```

`-case`

sets up the case for build and run phases. It creates `Buildconf/`, `$CASE.$MACH.run`, `$CASE.$MACH.build`, `$CASE.$MACH.clean_build`, `$CASE.$MACH.l_archive`, directories and files in $CASEROOT.

`-cleanmach`

Moves all machine-related files to a date-stamped backup directory under `MachinesHist/`. These files include: `Macros.$MACH`, `$CASE.$MACH.build`, `$CASE.$MACH.clean_build`, `$CASE.$MACH.l_archive`, and `$CASE.$MACH.run`. It also unlocks `env_mach_pes.xml`, `Macros.$MACH`, and `env_build.xml`, so users can reset machine tasks and threads and rerun configure. Reconfiguring with -cleanmach results in the loss of any local modifications to the local build and run scripts. But the `Buildconf/` directory will *not* be updated in this process. As a result, local changes to namelists will be preserved.

If you only modify `env_mach_pes.xml` after running **configure**, do the following:

```
> configure -cleanmach
> # Make changes to env_mach_pes.xml
> configure -case
```

`-cleannamelist`

Moves `Buildconf/` to a date-stamped backup directory under `MachinesHist/` and unlocks `env_conf.xml`. Reconfiguring with -cleannamelist results in the loss of any local modifications to the Buildconf buildnml and buildexe files. But the local build and run scripts will be preserved.

If you only want to modify `env_conf.xml` after running **configure**, do the following:

```
> configure -cleannamelist
> # Make changes to env_conf.xml here
> configure -case
```

`-cleanall`

This performs the functions of both the -cleanmach and -cleannamelist options. All files associated with the previous invocation of **configure** are moved to a time-stamped directory in `MachinesHist`. The $CASEROOT directory will now appear as if **create_newcase** had just been run with the exception that local modifications to the `env_*.xml` files are preserved. After further modifications are made to `env_conf.xml` and `env_mach_pes.xml`, you must run **configure -case** before you can build and run the model. Reconfiguring results in the loss of all local modifications to the component buildnml or buildexe files in Buildconf as well as the loss of all local modifications to the local build and run scripts.

`-help`

Lists all options with short descriptions.

Configure generates buildnml and buildexe scripts for each component in the Buildconf directory. It also generates build, run, l_archive, and clean_build scripts in the CASEROOT directory. These scripts are now sufficient to build and run the model.

Table 3-1. Result of invoking configure

File or Directory	Description
Buildconf/	Contains scripts that generate component libraries and utility libraries (e.g., PIO, MCT) and scripts that generate component namelists.
$`CASE.$MACH.build`	Creates the component and utility libraries and model executable (see building CCSM).
$`CASE.$MACH.run`	Runs the CCSM model and performs short-term archiving of output data (see running CCSM). Contains the necessary batch directives to run the model on the required machine for the requested PE layout.
$`CASE.$MACH.l_archive`	Performs long-term archiving of output data (see long-term archiving). This script will only be created if long-term archiving is available on the target machine.
$`CASE.$MACH.clean_build`	Removes all object files and libraries and unlocks `Macros.$MACH` and `env_build.xml`. This step is required before a clean build of the system.
env_derived	Contains environmental variables derived from other settings. Should *not* be modified by the user.

Customizing the configuration

Before calling **configure**, first customize the default configuration. To customize the default configuration, modify `env_conf.xml` and `env_mach_pes.xml` *before* invoking

configure. The env_build.xml and env_run.xml files can also be changed at this step.

env_mach_pes.xml contains variables that determine the layout of the components across the hardware processors. Those variables specify the number of processors for each component and determine the layout of components across the processors used. See env_mach_pes.xml variables for a summary of all env_mach_pes.xml variables.

env_conf.xml contains several different kinds of variables including variables for case initialization, variables that specify the regridding files, and variables that set component-specific namelists and component-specific CPP variables. See env_conf.xml variables for a summary of all env_conf.xml variables.

Setting the case PE layout

Optimizing the throughput or efficiency of a CCSM experiment often involves customizing the processor (PE) layout for load balancing. The component PE layout is set in env_mach_pes.xml.

CCSM4 has significant flexibility with respect to the layout of components across different hardware processors. In general, its CCSM components -- atm, lnd, ocn, ice, glc, and cpl -- can run on overlapping or mutually unique processors. Each component is associated with a unique MPI communicator. In addition, the driver runs on the union of all processors and controls the sequencing and hardware partitioning. The processor layout for each component is specified in the env_mach_pes.xml file 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.

For example, these settings in env_mach_pes.xml:

```
<entry id="NTASKS_OCN" value="128" />
<entry id="NTHRDS_OCN" value="1" />
<entry id="ROOTPE_OCN" value="0" />
```

cause the ocean component to run on 128 hardware processors with 128 MPI tasks using one thread per task starting from global MPI task 0 (zero).

In this next example:

```
<entry id="NTASKS_ATM" value="16" />
<entry id="NTHRDS_ATM" value="4"  />
<entry id="ROOTPE_ATM" value="32" />
```

the atmosphere component will run on 64 hardware processors using 16 MPI tasks and 4 threads per task starting at global MPI task 32. There are NTASKS, NTHRDS, and ROOTPE input variables for every component in env_mach_pes.xml. There are some important things to note.

- NTASKS must be greater or equal to 1 (one) even for inactive (stub) components.
- NTHRDS must be greater or equal to 1 (one). If NTHRDS is set to 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. 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.
- If all components have identical NTASKS, NTHRDS, and ROOTPE set, all components will run sequentially on the same hardware processors.

The root processor is set relative to the MPI global communicator, not the hardware processor counts. For instance, in the following example:

```
<entry id="NTASKS_ATM" value="16" />
<entry id="NTHRDS_ATM" value="4"  />
<entry id="ROOTPE_ATM" value="0"  />
<entry id="NTASKS_OCN" value="64" />
<entry id="NTHRDS_OCN" value="1"  />
<entry id="ROOTPE_OCN" value="16" />
```

the atmosphere and ocean are running 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. The batch submission script ($CASE.$MACH.run) should 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 set ROOTPE_OCN=64 in the preceding 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 "configure -case" has been invoked without first invoking "configure -cleanmach". For an example of changing pes, see the Section called *Changing PE layout* in Chapter 9

Setting the case initialization type

The case initialization type is set in `env_conf.xml`. A CCSM run can be initialized in one of three ways; startup, branch, or hybrid. The variable $RUN_TYPE determines the initialization type and is set to "startup" by default when **create_newcase** is invoked. 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.

RUN_TYPE

Run initialization type. Valid values: startup, hybrid, branch. Default: startup.

RUN_STARTDATE

Start date for the run in yyyy-mm-dd format. This is only used for startup or hybrid runs.

RUN_REFCASE

Reference case for hybrid or branch runs.

RUN_REFDATE

Reference date in yyyy-mm-dd format for hybrid or branch runs.

This is a detailed description of the different ways that CCSM initialization runs.

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 (normally the second day).

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_conf.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 in `env_conf.xml` 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 for an example.

hybrid

A hybrid run indicates that CCSM will be initialized more like a startup, but will use 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 CCSM 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 (normally the second day), and the coupler does a "cold start" without a restart file.

Setting component-specific variables

To understand how the component-specific variables in `env_conf.xml` (e.g., CAM_CONFIG_OPTS) are used to set compile and namelist settings for that component, you first need to understand how **configure** uses the variables in `env_conf.xml` to create the files in `Buildconf/`.

In each $CASEROOT directory, the subdirectory `$CASEROOT/Tools/Templates` contains files of the form $component[.cpl7].template, where $component corresponds to each of the model components that is part of the selected component set. The .cpl7 appears in some templates and not in others. **configure** translates the `$env_*.xml` xml variables to csh environment variables, and each of the component template scripts uses those environment variables to create `Buildconf/$component.buildexe.csh` (which creates the component library) and `Buildconf/$component.buildnml.csh` (which creates the component namelist).

```
configure
   ⇓
Tools/Templates/$component.cpl7.template
   ⇓
Buildconf/$component.buildexe.csh
Buildconf/$component.buildnml.csh
```

When the model run is submitted, `$CASE.$MACH.run` will call `Buildconf/$component.buildnml.csh` to produce the run-time component namelists:

```
$CASE.$MACH.run
   ⇓
Buildconf/$component.buildnml.csh
   ⇓
$RUNDIR/$model_in
```

As an example, for CAM

```
$CASE.$MACH.run
   ⇓
Buildconf/cam.buildnml.csh
   ⇓
$RUNDIR/atm_in
```

Important:: Component namelists should normally be set using `env_conf.xml` variables (see below). If a namelist needs to be modified after **configure** is called, then this should be done in `Buildconf/$component.buildnml.csh`. Note that if **configure -cleanall** or **configure -cleannamelist** is called, then any local changes to `Buildconf/$component.buildnml.csh` will be lost.

The discussions in the sections below assume the following:

- References to `$component.cpl7.template` refer to `Tools/Templates/$component.cpl7.template`
- References to `$component.buildexe.csh` and `$component.buildnml.csh` refer to `Buildconf/$component.buildexe.csh` and `Buildconf.$component.buildnml.csh`.

CAM variables

The following are CAM-specific `env_conf.xml` variables

CAM's configure utility[1] is invoked by `cam.cpl7.template` as:

```
configure
   ⇓
Tools/Templates/cam.cpl7.template
   ⇓
$CCSMROOT/models/atm/cam/bld/configure \
   -ccsm_seq -ice none -ocn none -spmd \
   -dyn $CAM_DYCORE -res $ATM_GRID \
   $CAM_CONFIG_OPTS \
  ...
   ⇓
 camconf/
 # Do Not Modify contents of camconf/
   ⇓
 Buildconf/cam.buildexe.csh
```

Note that $CAM_DYCORE and $ATM_GRID are `env_conf.xml` and `env_case.xml` variables, respectively.

CAM's build-namelist utility[2] is invoked by `cam.cpl7.template` as:

```
configure
   ⇓
Tools/Templates/cam.cpl7.template
   ⇓
$CCSMROOT/models/atm/cam/bld/build-namelist \
   -use_case $CAM_NML_USE_CASE \
   -namelist "$CAM_NAMELIST_OPTS /"
   ...
   ⇓
 camconf/
 # Do Not Modify contents of camconf/
   ⇓
 Buildconf/cam.buildnml.csh
```

The following `env_conf.xml` variables are used by CAM to invoke its configure and build-namelist utilities.

CAM_CONFIG_OPTS

Provides option(s) for CAM's configure utility (see above). CAM_CONFIG_OPTS are normally set as compset variables (e.g., "-phys cam3_5_1 -chem waccm_mozart") and in general should not be modified for supported compsets. Recommendation: If you want to modify this value for your experiment, use your own (user-defined component sets).

CAM_NML_USE_CASE

Provides the use_case option for CAM's build-namelist utility (see above). CAM's build-namelist leverages groups of namelist options (use cases) that are often paired with CAM's configure options. These use cases are xml files located in `$CCSMROOT/models/atm/cam/bld/namelist_files/use_cases`. In general, this variable should not be modified for supported compsets. Recommendation: If you want to modify this value for your experiment, use your own (user-defined component sets.

CAM_NAMELIST_OPTS

Provides options to the -namelist argument in CAM's build-namelist utility (see above).

This serves to specify namelist settings directly on the command line by supplying a string containing Fortran namelist syntax. The resulting namelist will appear in `cam.buildnml.csh`.

Note: To insert a single quotation mark (apostrophe) when setting CAM_NAMELIST_OPTS, use the string "'". Also note that the "$" symbol should not be used in CAM_NAMELIST_OPTS.

This example shows how to use **xmlchange** to set CAM_NAMELIST_OPTS:

```
xmlchange -id CAM_NAMELIST_OPTS\
          -val ncdata=&apos;cam_0.9x1.25.1860.nc&apos;
```

If you want to modify numerous cam namelist values, you can use an alternate scheme: Place a file `user_nl` containing modified cam namelist settings in `SourceMods/src.cam`. For example, `user_nl` could contain the following:

```
&psolar_inparm
 solar_const = 1363.27
/
&cam_inparm
 ch4vmr = 1860.0e-9
 nhtfrq = -24
/
```

and the above settings would appear in `cam.buildnml.csh`.

CLM variables

The following are CLM-specific `env_conf.xml` variables

CLM's **configure** utility [3] is invoked by `clm.cpl7.template` as:

```
configure -case
   ⇓
Tools/Templates/clm.cpl7.template
   ⇓
$CCSMROOT/models/lnd/clm/bld/configure \
   -mode ext_ccsm_seq \
    -comp_intf cpl_$COMP \
    -usr_src $CASEROOT/SourceMods/src.clm \
    $CLM_CONFIG_OPTS
    ...
    ⇓
  clmconf/
  # Do Not Modify contents of clmconf/
    ⇓
  Buildconf/clm.buildexe.csh
```

CLM's build-namelist utility [4] is also invoked by `clm.cpl7.template` as:

```
configure -case
   ⇓
Tools/Templates/clm.cpl7.template
   ⇓
$CCSMROOT/lnd/clm/bld/build-namelist
  -clm_usr_name $CLM_USRDAT_NAME \
  -res $LND_GRID -mask $OCN_GRID \
  -clm_start_type $START_TYPE
  -use_case $CLM_NML_USE_CASE
  -namelist "&clm_inparm $CLM_NAMELIST_OPTS /"
   ...
   ⇓
 clmconf/
 # Do Not Modify contents of clmconf/
   ⇓
 Buildconf/clm.buildnml.csh
```

Note: Confusion can arise here. CLM supports the values of default, cold, arb_ic, and startup for the -clm_start_type argument. A value of cold implies always starting with arbitrary initial conditions. A value of arb_ic implies starting with arbitrary initial conditions *if* initial conditions do not exist. A value of startup implies that initial conditions *must* be used, and the **configure -case** will abort if one isn't provided (either from the CLM XML namelist database, or entered with CLM_NAMELIST_OPTS or `user_nl_clm`). If "default" is entered, the CLM build-namelist will determine the setting based on the resolution.

$START_TYPE (above) is a derived variable in `clm.cpl7.template`. For a hybrid run, $START_TYPE is set to "startup", otherwise it is set to "default" unless $$CLM_FORCE_COLDSTART is set to "on", in which case is it set to "cold". It is unfortunate that the name "startup" is the same as the name used for initializing a CCSM run, because in this case it means something very different.

CLM_CONFIG_OPTS

Provides option(s) for CLM's **configure** utility (see above). CLM_CONFIG_OPTS are normally set as compset variables (e.g., -bgc cn).

Do not modify this variable. If you want to modify this for your experiment, use your own (user-defined component sets).

This is an advanced flag and should only be used by expert users.

CLM_NAMELIST_OPTS

CLM-specific namelist settings for -namelist option in CLM's **build-namelist** (see above). We recommend that if you want to change the clm namelist settings such as the initial dataset (finidat) or the history output defaults (hist_nhtfrq) you either include a `user_nl_clm` or you manually edit the resulting `Buildconf/clm.buildnml.csh`.

This is an advanced flag and should only be used by expert users.

CLM_FORCE_COLDSTART

Flag to CLM's **build-namelist** to force CLM to do a cold start. Valid values are on, off. The "on" value forces the model to spin up from a cold-start (arbitrary initial conditions).

This is an advanced flag and should only be used by expert users.

CLM_USRDAT_NAME

Dataset name for user-created datasets. This is used as the argument to **build-namelist -clm_usr_name** (see above). An example of such a dataset would be, `1x1pt_boulderCO_c090722`. The default value is UNSET.

This is an advanced flag and should only be used by expert users.

CLM_PT1_NAME

Grid name when the CLM/ATM grid is a single point. This is used in I compsets only.

This is an advanced flag and should only be used by expert users.

CLM_CO2_TYPE

Determines how CLM will determine where CO_2 is set. If `constant`, it will be set to CCSM_CO2_PPMV, if set otherwise, the atmosphere model *MUST* send it to CLM. CLM_CO2_TYPE is normally set by the specific compset, since it *HAS* to be coordinated with settings for the atmospheric model.

Do not modify this variable. If you want to modify for your experiment, use your own (user-defined component sets).

This is an advanced flag and should only be used by expert users.

CLM_NML_USE_CASE

Determines the use-case that will be sent to the CLM **build-namelist**. CLM_CO2_TYPE is normally set by the specific compset.

This is an advanced flag and should only be used by expert users.

CICE variables

The following are CICE-specific `env_conf.xml` variables

CICE's configure utility[5] is invoked by `cice.cpl7.template` as:

```
configure -case
   ⇓
Tools/Templates/cice.cpl7.template
   ⇓
$CCSMROOT/models/ice/cice/bld/configure \
   -hgrid $ICE_GRID  \
   -mode $CICE_MODE  \
   $CICE_CONFIG_OPTS \
   ...
   ⇓
 ciceconf/
 # Do Not Modify contents of ciceconf/
   ⇓
Buildconf/cice.buildexe.csh
```

CICE's build-namelist utility[6] is invoked by `cice.cpl7.template` as:

```
configure -case
   ⇓
Tools/Templates/cice.cpl7.template
   ⇓
$CCSMROOT/ice/cice/bld/build-namelist
   -presaero_type $CICE_PRESAERO_TYPE
   -namelist "&cice_inparm $CICE_NAMELIST_OPTS /"
   ...
   ⇓
 ciceconf/
 # Do Not Modify contents of ciceconf/
   ⇓
Buildconf/cice.buildnml.csh
```

CICE_MODE

Option to CICE's configure utility for the -mode argument (see above). Valid values are prognostic, prescribed, thermo_only. The default is prognostic.

CICE_CONFIG_OPTS

Provides option(s) for CICE's configure utility (see above). Default value is "-ntr_aero 3 -ntr_pond 1 -ntr_iage 1 -ntr_FY 1".

CICE_PRESAERO_TYPE

Option to CICE's build-namelist utility for the -prescaero flag (see above). This determines the CICE prescribed aerosol type and is only used if not obtained from atmosphere.

CICE_NAMELIST_OPTS

CICE-specific namelist settings for -namelist option (see above).

In addition, $CASEROOT/configure also generates the CICE's block decomposition in env_build.xml as follows (also see env_build.xml variables):

```
configure -case
   ⇓
$NTASKS_ICE and $NTHRDS_ICE
   ⇓
Tools/Templates/generate_cice_decomp.xml
   ⇓
Tools/Templates/cice_decomp.xml
   ⇓
sets env_build.xml CICE_BLCKX
sets env_build.xml CICE_BLCKY
sets env_build.xml CICE_MXBLCKS
sets env_build.xml CICE_DECOMPTYPE
   ⇓
CPP variables in cice.buildexe.csh
```

POP2 variables

The following are POP2-specific env_conf.xml variables

POP2's namelist construction utility[7] is invoked by pop2.cpl7.template in the following sequence:

```
configure -case
   ⇓
Tools/Templates/pop2.cpl7.template
   ⇓
$CCSMROOT/models/ocn/pop2/input_templates/pop2_in_build.csh
$CCSMROOT/models/ocn/pop2/input_templates/ocn.*.setup.csh
   ⇓
Buildconf/pop2.buildnml.csh
```

POP2's script to generate the ocn binary library is created directly from pop2.cpl7.template:

```
configure -case
   ⇓
Tools/Templates/pop2.cpl7.template
   ⇓
Buildconf/pop2.buildexe.csh
```

In addition, configure also generates POP2's block decomposition in env_build.xml as follows (also see env_build.xml variables):

```
configure -case
   ⇓
$NTASKS_OCN and $NTHRDS_OCN
   ⇓
Tools/Templates/generate_pop_decomp.xml
   ⇓
Tools/Templates/pop_decomp.xml
   ⇓
sets env_build.xml POP_BLCKX
sets env_build.xml POP_BLCKY
sets env_build.xml POP_MXBLCKS
sets env_build.xml POP_DECOMPTYPE
   ⇓
CPP variables in pop2.buildexe.csh
```

The following variables are used by the POP2 scripts to generate the settings used in your $CASE.

OCN_CHL_TYPE

Determine provenance of surface Chl for radiative penetration computations. Valid values are diagnostic, prognostic. The default is diagnostic. This option is used in the POP2 ecosystem model, which will be available in the CESM1.0 release.

OCN_CO2_TYPE

Determine provenance of atmospheric CO2 for gas flux computation. Valid values are constant, prognostic. The default is constant. This option is used in the POP2 ecosystem model, which will be available in the CESM1.0 release.

OCN_COUPLING

Determine surface freshwater and heat forcing settings. Valid values are full, partial. The full option yields settings that are appropriate for coupling to an active atmospheric model (e.g., a B-type compset). The partial option yields settings that are appropriate for coupling to a data atmospheric model (e.g., a C or G-type compset). The create_newcase command selects the appropriate setting for this variable based on the specified compset. Users should not change this setting.

OCN_ICE_FORCING

Determine under-ice forcing settings. Valid values are active, inactive. The active option yields settings that are appropriate for coupling to an active ice model (e.g., a B or G-type compset). The inactive option yields settings that are appropriate for coupling to a data ice model (e.g., a C-type compset). The create_newcase command selects the appropriate setting for this variable based on the specified compset. Users should not change this setting.

OCN_TRANSIENT

Determine settings for transient forcing datasets (e.g., atmospheric pCFC concentrations). Valid values are unset, 1850-2000. The create_newcase command selects the appropriate setting for this variable based on the specified compset. Users should not change this setting. This option is used in the POP2 ecosystem model, which will be available in the CESM1.0 release.

DATM variables

The following are DATM-specific `env_conf.xml` variables

DATM is discussed in detail in Data Model's User's Guide[8]. DATM is normally used to provide observational forcing data (or forcing data produced by a previous run using active components) to drive 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.

DATM uses the `datm.cpl7.template` as follows:

```
configure
   ⇓
Tools/Templates/datm.cpl7.template
   ⇓
Buildconf/datm.builexe.csh
```

and

```
configure
   ⇓
Tools/Templates/datm.cpl7.template
   ⇓
$CCSMROOT/scripts/ccsm_utils/build_streams
   ⇓
Buildconf/datm.buildml.csh
```

The following are CCSM environment variables that are used by `datm.cpl7.template`:

DATM_MODE

Mode for data atmosphere component (datm). Valid values are CORE2_NYF, CLM_QIAN, CLM1PT. The default is CORE2_NYF.

CORE2_NYF (CORE2 normal year forcing) is the DATM mode used in C and G compsets.

CLM_QIAN and CLM1PT are DATM modes using observational data for forcing CLM in I compsets.

DATM_CLMNCEP_YR_ALIGN

For I compset only. Year align (simulation year corresponding to starting year) for CLM_QIAN mode. Default value is 1.

DATM_CLMNCEP_YR_START

For I compset only. Starting year to loop data over for CLM_QIAN mode. Default value is 2004.

DATM_CLMNCEP_YR_END

For I compset only. Ending year to loop data over for CLM_QIAN mode. Default value is 2004.

DLND variables

The following are DLND-specific `env_conf.xml` variables

DLND is discussed in detail in Data Model's User's Guide[9]. The data land model is different from the other data models because it can run as a purely data-runoff model (reading in runoff data), or as a purely data-land model (reading in coupler history data for atm/land fluxes and land albedos produced by a previous run), or both. In general, the data land model is only used to provide runoff forcing data to POP2 when running C or G compsets.

DLND uses the `datm.cpl7.template` as follows:

```
configure
   ⇓
Tools/Templates/dlnd.cpl7.template
   ⇓
Buildconf/dlnd.builexe.csh
```

and

```
configure
   ⇓
Tools/Templates/dlnd.cpl7.template
   ⇓
$CCSMROOT/scripts/ccsm_utils/build_streams
   ⇓
Buildconf/dlnd.buildml.csh
```

The following are variables that are used by `dlnd.cpl7.template`.

DLND_MODE

DLND mode. Valid values are CPLHIST and NULL. In CPLHIST mode, land forcing data (produced by CLM) from a previous model run is output in coupler history files and read in by the data land model. In NULL mode, land forcing is set to zero and not utilized. The default is NULL.

DLND_RUNOFF_MODE

DLND_RUNOFF mode. Valid values are CPLHIST, RX1, and NULL. In RX1 mode, observational 1-degree runoff data is used. In CPLHIST mode, runoff data from a previous model run is read in. In NULL mode, the runoff data is set to zero. In CPLHIST mode, land forcing data from a previous model run is output by the coupler and read in by the data land model. In NULL mode, land forcing is set to zero and not used. The default is RX1.

DICE variables

The following are DICE-specific `env_conf.xml` variables

DICE is discussed in detail in Data Model's User's Guide[10]. 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). DICE acts very similarly to CICE running in prescribed mode.) Currently, this component is only used to drive POP2 in C compsets.

DICE uses the `dice.cpl7.template` as follows:

```
configure
   ⇓
Tools/Templates/dice.cpl7.template
   ⇓
Buildconf/dice.builexe.csh
```

and

```
configure
   ⇓
Tools/Templates/dice.cpl7.template
   ⇓
$CCSMROOT/scripts/ccsm_utils/build_streams
   ⇓
Buildconf/dice.buildml.csh
```

The following are variables that are used by `dice.cpl7.template`.

DICE_MODE

DICE mode. Valid value is ssmi.

DOCN variables

The following are DOCN-specific `env_conf.xml` variables

The following are variables that are used by `docn.cpl7.template`.

DOCN is discussed in detail in Data Model's User's Guide[11].

The data ocean component (DOCN) always returns SSTs to the driver. In CCSM, atmosphere/ocean fluxes are computed in the coupler. Therefore, the data ocean model does not compute fluxes like the data ice model. DOCN has two distinct modes of operation. It 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 (in conjunction with CICE running in prescribed mode) is used in all F component sets.

DOCN running as a slab ocean model is used (in conjunction with CICE running in prognostic mode) in all E compsets.

For prescribed mode, default yearly climatological datasets are provided for various model resolutions. For multi-year runs requiring AMIP datasets of sst/ice_cov fields, you need to set the variables for DOCN_SSTDATA_FILENAME, DOCN_SSTDATA_YEAR_START, and DOCN_SSTDATA_YEAR_END. CICE in prescribed mode also uses these values.

DOCN uses the `docn.cpl7.template` as follows:

```
configure
   ⇓
Tools/Templates/docn.cpl7.template
   ⇓
Buildconf/docn.builexe.csh
```

and

```
configure
   ⇓
Tools/Templates/docn.cpl7.template
   ⇓
$CCSMROOT/scripts/ccsm_utils/build_streams
   ⇓
Buildconf/docn.buildml.csh
```

DOCN_MODE

DOCN mode. Valid values are prescribed, som. Default is prescribed.

DOCN_SSTDATA_FILENAME

Sets sst/ice_cov filename for AMIP runs, only used in F compset. Default is UNSET.

DOCN_SSTDATA_YEAR_START

Sets start year of sst/ice_cov for AMIP runs, only used in F compset. Default is -999.

DOCN_SSTDATA_YEAR_END

Sets end year of sst/ice_cov for AMIP runs, only used in F compset. Default is -999.

Driver/coupler variables

The following are CPL-specific `env_conf.xml` variables

CPL_EPBAL

Provides EP balance factor for precip for POP2. A factor computed by POP2 is applied to precipitation so that precipitation balances evaporation and ocn global salinity does not drift. This is intended for use when coupling POP2 to a DATM. Only used for C and G compsets.

Valid values are off, ocn. Default is off.

CPL_ALBAV

If false, albedos are computed with the assumption that downward solar radiation from the atm component has a diurnal cycle and zenith-angle dependence. This is normally the case when using an active atm component.

If true, albedos are computed with the assumption that downward solar radiation from the atm component is a daily average quantity and does not have a zenith-angle dependence. This is often the case when using a data atm component.

This is only used for C and G compsets. Valid values are true, false. Default is false.

CCSM_BGC

If the value is not "none", the coupler is compiled so that optional BGC related fields are exchanged between component models. Currently only "none" and "CO2A" are supported. CO2A will activate sending diagnostic and prognostic co2 from the atm component to the lnd component.

Valid values are: none, CO2A.

MAP_A2OF_FILE

atm-to-ocn mapping file for fluxes (currently first-order conservative).

MAP_A2OS_FILE

atm-to-ocn mapping file for states (currently bilinear).

MAP_O2AF_FILE

ocn-to-atm mapping file for fluxes (currently first-order conservative).

MAP_O2AS_FILE

ocn-to-atm mapping file for states (currently bilinear).

MAP_A2LF_FILE

atm-to-land mapping file for fluxes (currently first-order conservative).

MAP_A2LS_FILE

atm-to-land mapping file for states (currently bilinear).

MAP_R2O_FILE_R05

0.5-degree runoff-to-ocn mapping file.

MAP_R2O_FILE_RX1

1-degree runoff-to-ocn mapping file.

MAP_R2O_FILE_R19

19-basin runoff-to-ocn mapping file.

Other variables

The following variables impact more than one component.

CCSM_CO2_PPMV

This set the namelist values of CO2 ppmv for CAM and CLM. This variables is introduced to coordinate this value among multiple components.

Reconfiguring a Case

If env_conf.xml or env_mach_pes.xml variables need to be changed after configure has been called, the case scripts will need to be reconfigured. To do this, run

```
> cd $CASEROOT
> configure -cleanall
> configure -case
```

This will update the buildnml and buildexe files in the Buildconf directory and the case build and run scripts in the CASEROOT directory. If only variables in env_conf.xml have change, clean and reconfigure just the files in Buildconf by doing

```
> cd $CASEROOT
> configure -cleannamelist
> configure -case
```

If only variables in env_mach_pes.xml have changed, clean and reconfigure the build and run scripts by doing

```
> cd $CASEROOT
> configure -cleanmach
> configure -case
```

Note that the -cleanall option does a combination of the -cleannamelist and -cleanmach options. Reconfiguring with -cleanall results in the loss of all local modifications to the component buildnml or buildexe files in Buildconf as well as the loss of all local modifications to the local build and run scripts.

Summary of Files in the Case Directory

This table summarizes the directories and files that are created by configure. For more complete information about the files in the case directory, see the Section called *What are all these directories and files in my case directory?* in Chapter 11

Table 3-2. Result of invoking configure

File or Directory	Description
Buildconf/	Contains scripts that generate component libraries and utility libraries (e.g., PIO, MCT) and scripts that generate component namelists.
$CASE.$MACH.build	Creates the component and utility libraries and model executable (see building CCSM).

File or Directory	Description
$CASE.$MACH.run	Runs the CCSM model and performs short-term archiving of output data (see running CCSM). Contains the necessary batch directives to run the model on the required machine for the requested PE layout.
$CASE.$MACH.l_archive	Performs long-term archiving of output data (see long-term archiving). This script will only be created if long-term archiving is available on the target machine.
$CASE.$MACH.clean_build	Removes all object files and libraries and unlocks Macros.$MACH and env_build.xml. This step is required before a clean build of the system.
env_derived	Contains environmental variables derived from other settings. Should *not* be modified by the user.

Notes

1. http://www.ccsm.ucar.edu/models/ccsm4.0/cam/docs/users_guide/book1.html
2. http://www.ccsm.ucar.edu/models/ccsm4.0/cam/docs/users_guide/book1.html
3. http://www.ccsm.ucar.edu/models/ccsm4.0/clm/models/lnd/clm/doc/UsersGuide/book1.html
4. http://www.ccsm.ucar.edu/models/ccsm4.0/clm/models/lnd/clm/doc/UsersGuide/book1.html
5. http://www.ccsm.ucar.edu/models/ccsm4.0/cice/doc/index.html
6. http://www.ccsm.ucar.edu/models/ccsm4.0/cice/doc/index.html
7. http://www.ccsm.ucar.edu/models/ccsm4.0/pop/doc/users/
8. http://www.ccsm.ucar.edu/models/ccsm4.0/data8/book1.html
9. http://www.ccsm.ucar.edu/models/ccsm4.0/data8/book1.html
10. http://www.ccsm.ucar.edu/models/ccsm4.0/data8/book1.html
11. http://www.ccsm.ucar.edu/models/ccsm4.0/data8/book1.html

Chapter 4. Building a Case

After configuring a case, the model executable can be built by running **$CASE.$MACH.build** which will:

1. create the component namelists in $RUNDIR ($EXEROOT/run) by calling the `Buildconf/` scripts `$component.buildnml.csh`.
2. check for the required input data sets and download missing data automatically on local disk, and if successful proceed to the following steps.
3. create the necessary utility libraries by calling the `Buildconf/` scripts `mct.buildlib`, `pio.buildlib` and `csm_share.buildlib`.
4. create the necessary component libraries by calling the `Buildconf/` scripts `$component.buildexe.csh`.
5. create the model executable by calling the `Buildconf/` scripts `ccsm.buildexe.csh`.

Note: `$CASEROOT/Tools/Makefile` and `$CASEROOT/Macros.$MACH` are used to generate all necessary utility and component libraries and the model executable.

A user does not need to change the default build settings to create the executable. However, the CCSM scripts provide the user with a great deal of flexibility in customizing various aspects of the build process. It is therefore useful for a user to become familiar with these in order to make optimal use of the system.

Input data

All active and data components use input datasets. A local disk needs to be populated with input data in order to run CCSMwith these components. For all machines, input data is provided as part of the release via data from a subversion input data server. However, on supported machines (and some non-supported machines), data already exists in the default local-disk input data area (as specified by $DIN_LOC_ROOT_CSMDATA (see below).

Input data is handled by the build process as follows:

- **configure** and buildnml scripts create listings of required component input datasets in the `Buildconf/$component.input_data_list` files.
- `&$CASE.$MACH.build`; in the prestage step checks for the presence of the required input data files in the root directory $DIN_LOC_ROOT_CSMDATA. 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, the build script will abort and 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 following variables in `env_run.xml` determine where you should expect input data to reside on local disk and how this input data will be handled during the run.

DIN_LOC_ROOT_CSMDATA

The root directory of CCSM input data for the selected machine. Usually a shared disk area.

DIN_LOC_ROOT

The inputdata area used for the current case. Normally, this is set to DIN_LOC_ROOT_CSMDATA but the system provides flexibility for a user to specify a distinct directory. This might be needed on certain machines if the user needs to have the input data reside in a special disk area associated with the executable directory or the batch nodes, rather than in the default local disk directory.

DIN_LOC_ROOT_CLMQIAN

CLM-specific root directory for CLM QIAN type input forcing data. This directory will only be used for I (CLM/DATM) compsets.

PRESTAGE_DATA

Allows the case input data root directory (DIN_LOC_ROOT) to differ from the machine's root input data directory (DIN_LOC_ROOT_CSMDATA).

If PRESTAGE_DATA is FALSE (the default) then DIN_LOC_ROOT will be set to DIN_LOC_ROOT_CSMDATA.

If PRESTAGE_DATA is TRUE, then DIN_LOC_ROOT will be set to `EXEROOT/inputdata`. In addition, data will be copied from DIN_LOC_ROOT_CSMDATA to DIN_LOC_ROOT before the model builds. The input data in `EXEROOT/inputdata` will then be used for the model run.

User-created input data

If you want to use new user-created dataset(s), give these dataset(s) names that are different than the names in $DIN_LOC_ROOT. The best way to access these user-specified datasets is to use the script `$CCSMROOT/scripts/link_dirtree`. **link_dirtree** creates a virtual copy of the input data directory by linking one directory tree to another. The full directory structure of the original directory is duplicated and the files are linked. Invoke the following for usage:

```
> cd $CCSMROOT/scripts
> link_dirtree -h
```

link_dirtree can be conveniently used to generate the equivalent of a local copy of $DIN_LOC_ROOT_CSMDATA which can then be populated with user-specified input datasets. For example, you can first generate a virtual copy of $DIN_LOC_ROOT_CSMDATA in /user/home/newdata with the following command:

```
> link_dirtree $DIN_LOC_ROOT_CSMDATA /user/home/newdata
```

then incorporate the new dataset(s) directly into the appropriate directory in /user/home/newdata.

Important:: If you place a new dataset for $component in `/user/home/newdata`, then `Buildconf/$component.buildnml.csh` and `Buildconf/$component.input_data_list` must be modified to point to this new dataset.

Using the input data server

The script `$CASEROOT/check_input_data` determines if the required data files for the case exist on local disk in the appropriate subdirectory of $DIN_LOC_ROOT_CSMDATA. If any of the required datasets do not exist

locally, `check_input_data` provides the capability for downloading them to the $DIN_LOC_ROOT_CSMDATA directory hierarchy via interaction with the 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_CSMDATA -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_CSMDATA -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_CSMDATA. For what to expect when interacting with a Subversion repository, see downloading input data.

Build-time variables

The `env_build.xml` file sets variables that control various aspects of building the model executable. Most of the variables should not be modified by users. The variables that you can modify are discussed in more detail below.

EXEROOT

The CCSM executable root directory. This is where the model builds its executable and by default runs the executable. Note that $EXEROOT needs to have enough disk space for the experimental configuration requirements. As an example, CCSM can produce more than a terabyte of data during a 100-year run, so you should set $EXEROOT to scratch or tmp space and frequently back up the data to a mass storage device.

RUNDIR

The directory where the executable will be run. By default this is set to $EXEROOT/run. RUNDIR allows you to keep the run directory separate from the build directory.

BUILD_THREADED

Valid values are TRUE and FALSE. The default is FALSE.

If FALSE, the component libraries are built with OpenMP capability only if the NTHREADS_ setting for that component is greater than 1 in `env_mach_pes.xml`.

If TRUE, the component libraries are always built with OpenMP capability.

DEBUG

Flag to turn on debugging for run time and compile time. Valid values are TRUE, FALSE. The default is FALSE.

If TRUE, compile-time debugging flags are activated that you can use to verify software robustness, such as bounds checking.

Important:: On IBM machines, floating point trapping is not activated for production runs (i.e., non-DEBUG), due to performance penalties associated with turning on these flags.

GMAKE_J

Number of processors for gmake (integer). 0 < GMAKE_J < [number of processors/node]. $GMAKE_J allows a faster build on multi-processor machines. If the build fails in different places without other changes, setting this to 1 may help.

OCN_TRACER_MODULES

A POP2-specific setting for turning on different ocean tracer modules. Valid values are any combination of: iage, cfc, ecosys.

Compiler settings

Compiler settings are located in the two files `env_mach_specific` and `Macros.$MACH`. The `env_mach_specific` file is a shell script that sets various machine specific configure options such as modules and MPI or system environment variables. This script is run as part of every build or run step, and accounts for settings not included in the CCSM xml env files. The `Macros.$MACH` file contains the machine specific build options used in the CCSM Makefile. Both of these files are usually involved in defining build options, and the env_mach_specific file might also contain critical settings for the run phase.

If you are running at NCAR, env_mach_specific also contains variables to set up mass storage archiving. You need to modify these if you activate long-term archiving on the mass store.

You need to modify these files for user-defined machines during the process of porting CCSM to your machine.

User-modified source code

Each model component ($component) has an associated directory, `$CASEROOT/SourceMods/src.$component`, where you can place modified source code before building the executable. Any source code from $component that is placed in `$CASEROOT/SourceMods/src.$component/` will automatically be compiled when the model is built and will overwrite the default source code. For example, placing user-modifed cam source code in `$CASEROOT/SourceMods/src.cam` will cause the user-modified routines to be used instead of the routines in `$CCSMROOT/models/atm/cam`.

If you want to modify numerous cam namelist values, you can use place a file, `user_nl`, containing modified cam namelist settings in `SourceMods/src.cam` and this file will be used by the **configure** -case; command to generate the appropriate namelist for CAM.

Building the executable

After customizing your build options, and adding any user-modified source code, you are ready to build the case executable.

```
> cd $CASEROOT
> $CASE.$MACH.build
```

Diagnostic comments will appear as the build proceeds.

The following line indicates that the component namelists have been generated successfully:

```
....
CCSM BUILDNML SCRIPT HAS FINISHED SUCCESSFULLY
....
```

When the required case input data in $DIN_LOC_ROOT has been successfully checked, you will see:

```
CCSM PRESTAGE SCRIPT STARTING
...
...
CCSM PRESTAGE SCRIPT HAS FINISHED SUCCESSFULLY
```

Finally, the build script generates the utility and component libraries and the model executable. There should be a line for the mct and pio libraries, as well as each of the components. Each is date stamped, and a pointer to the build log file for that library or component is created. Successful completion is indicated by:

```
CCSM BUILDEXE SCRIPT HAS FINISHED SUCCESSFULLY
```

The build log files have names of the form $model.bldlog.$datestamp and are located in $RUNDIR. If they are compressed (indicated by a .gz file extension), then the build ran successfully.

Invoking **$CASE.$MACH.build** creates the following directory structure in $EXEROOT:

```
$EXEROOT/atm
$EXEROOT/ccsm
$EXEROOT/cpl
$EXEROOT/csm_share
$EXEROOT/glc
$EXEROOT/ice
$EXEROOT/lib
$EXEROOT/lnd
$EXEROOT/mct
$EXEROOT/ocn
$EXEROOT/pio
$EXEROOT/run
```

The atm/, ccsm/, cpl/, glc/, ice/, lnd/, and ocn/ subdirectories each contain an 'obj/' directory where the compiled object files for the model component is placed. These object files are collected into libraries that are placed in 'lib/' along with the mct/mpeu, pio, and csm_share libraries. Special include modules are also placed in lib/include. The model executable 'ccsm.exe' is placed in $`EXEROOT/run` along with component namelists. During the model run, component logs, output datasets, and restart files are also placed in this directory.

Rebuilding the executable

The model should be rebuilt under the following circumstances:

If `env_conf.xml`, `env_build.xml` or `$Macros.$MACH` 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
> $CASE.$MACH.clean_build
> $CASE.$MACH.build
```

If ONLY the PE layout has been modified in `env_mach_pes.xml` (see setting the PE layout) then it's possible that a clean is not required.

```
> cd $CASEROOT
```

```
> $CASE.$MACH.build
```

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

```
ERROR SMP STATUS HAS CHANGED
  SMP_BUILD = a0l0i0o0g0c0
  SMP_VALUE = a1l0i0o0g0c0
  A manual clean of your obj directories is strongly recommendend
  You should execute the following:
    ./b39pA1.bluefire.clean_build
  Then rerun the build script interactively
  ---- OR ----
  You can override this error message at your own risk by executing
    ./xmlchange -file env_build.xml -id SMP_BUILD -val 0
  Then rerun the build script interactively
```

and suggest that the model be rebuilt from scratch.

Note: The user is responsible for manually rebuilding the model when needed. If there is any doubt, you should rebuild.

Chapter 5. Running a case

To run a case, the user must submit the batch script `$CASE.$MACH.run`. In addition, the user needs to also modify `env_run.xml` for their particular needs.

`env_run.xml` contains variables which may be modified during the course of a model run. These 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. In general, the user needs to only set the variables $STOP_OPTION and $STOP_N. The other coupler 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.

Customizing runtime settings

As mentioned above, variables that control runtime settings are found in `env_run.xml`. In the following, we focus on the handling of run control (e.g. length of run, continuing a run) and output data. We also give a more detailed description of CCSM restarts.

Setting run control variables

Before a job is submitted to the batch system, the user needs first check that the batch submission lines in `$CASE.$MACH.run` are appropriate. These lines should be checked and modified accordingly for appropriate account numbers, time limits, and std-out/stderr file names. The user should then modify `env_run.xml` to determine the key run-time settings, as outlined below:

`CONTINUE_RUN`

Determines if the run is a restart run. Set to FALSE when initializing a startup, branch or hybrid case. Set to TRUE when continuing a run. (logical)

When you first begin a branch, hybrid or startup run, CONTINUE_RUN must be set to FALSE. When you successfully run and get a restart file, you will need to change CONTINUE_RUN to TRUE for the remainder of your run. Details of performing model restarts are provided below.

`RESUBMIT`

Enables the model to automatically resubmit a new run. To get multiple runs, set RESUBMIT greater than 0, then RESUBMIT will be decremented and the case will be resubmitted. The case will stop automatically resubmitting when the RESUBMIT value reaches 0.

Long CCSM runs can easily outstrip supercomputer queue time limits. For this reason, a case is usually run as a series of jobs, each restarting where the previous finished.

`STOP_OPTION`

Ending simulation time.

Valid values are: [none, never, nsteps, nstep, nseconds, nsecond, nminutes, nminute, nhours, nhour, ndays, nday, nmonths, nmonth, nyears, nyear, date, ifdays0, end] (char)

`STOP_N`

Provides a numerical count for $STOP_OPTION. (integer)

STOP_DATE

Alternative yyyymmdd date option, negative value implies off. (integer)

REST_OPTION

Restart write interval.

Valid values are: [none, never, nsteps, nstep, nseconds, nsecond, nminutes, nminute, nhours, nhour, ndays, nday, nmonths, nmonth, nyears, nyear, date, ifdays0, end] (char)

Alternative yyyymmdd date option, negative value implies off. (integer)

REST_N

Number of intervals to write a restart. (integer)

REST_DATE

Model date to write restart, yyyymmdd

STOP_DATE

Alternative yyyymmdd date option, negative value implies off. (integer)

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.

Output data

Each CCSM component produces its own output datasets consisting of history, restart and output log files. Component history files are in netCDF format whereas component restart files may be in netCDF or binary format and are used to either exactly restart the model or to serve as initial conditions for other model cases.

Archiving is a phase of a CCSM model run where the generated output data is moved from $RUNDIR (normally $EXEROOT/run) to a local disk area (short-term archiving) and subsequently to a long-term storage system (long-term archiving). It has no impact on the production run except to clean up disk space and help manage user quotas. Short and long-term archiving environment variables are set in the `env_mach_specific` file. Although short-term and long-term archiving are implemented independently in the scripts, there is a dependence between the two since the short-term archiver must be turned on in order for the long-term archiver to be activated. In `env_run.xml`, several variables control the behavior of short and long-term archiving. These are described below.

LOGDIR

Extra copies of the component log files will be saved here.

DOUT_S

If TRUE, short term archiving will be turned on.

DOUT_S_ROOT

Root directory for short term archiving. This directory must be visible to compute nodes.

DOUT_S_SAVE_INT_REST_FILES

If TRUE, perform short term archiving on all interim restart files, not just those at the end of the run. By default, this value is FALSE. This is for expert users ONLY and requires expert knowledge. We will not document this further in this guide.

DOUT_L_MS

If TRUE, perform long-term archiving on the output data.

DOUT_L_MSROOT

Root directory on mass store system for long-term data archives.

DOUT_L_HTAR

If true, DOUT_L_HTAR the long-term archiver will store history data in annual tar files.

DOUT_L_RCP

If TRUE, long-term archiving is done via the rcp command (this is not currently supported).

DOUT_L_RCP_ROOT

Root directory for long-term archiving on rcp remote machine. (this is not currently supported).

Several important points need to be made about archiving:

- By default, short-term archiving is enabled and long-term archiving is disabled.
- 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.
- If long-term archiving is enabled, files will be moved to $DOUT_L_MSROOT by `$CASE.$MACH.l_archive`, which is run as a separate batch job after the successful completion of a model run.
- Users should generally turn off short term-archiving when developing new CCSM code.
- If long-term archiving is not enabled, users must monitor quotas and usage in the $DOUT_S_ROOT/ directory and should manually clean up these areas on a frequent basis.

Standard output generated from each CCSM component is saved in a "log file" for each component in $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 beneath the case 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.

As an example, for the following example settings

```
DOUT_S = TRUE
DOUT_S_ROOT = /ptmp/$user/archive
DOUT_L_MS = TRUE
DOUT_L_MSROOT /USER/csm/b40.B2000
```

the run will automatically submit the `$CASE.$MACH.l_archive` to the queue upon its completion to archive the data. The system is not bulletproof, and the user will want to verify at regular intervals that the archived data is complete, particularly during long running jobs.

Load balancing a case

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. This is certainly true for CCSM4. It is strongly recommended that a user perform a load-balancing exercise on their proposed model run before undertaking a long production run.

CCSM4 has significant flexibility with respect to the layout of components across different hardware processors. In general, there are six unique models (atm, lnd, ocn, ice, glc, cpl) that are managed independently in CCSM4, each with a unique MPI communicator. In addition, the driver runs on the union of all processors and controls the sequencing and hardware partitioning.

Please see the section on setting the case PE layout for a detailed discussion of how to set processor layouts and the example on changing the PE layout .

Model timing data

In order to perform a load balancing exercise, the user must first be aware of the different types of timing information produced by every CCSM run. How this information is used is described in detail in using model timing data.

A summary timing output file is produced after every CCSM run. This file is placed in `$CASEROOT/timing/ccsm_timing.$CASE.$date`, where $date is a datestamp set by CCSM 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_summary.$CASE.$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), 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 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.

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 setup 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, its worth occasionally reviewing 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 de-

compositions at each processor count. If the cost and scaling of the components are unknown, several short tests can be carried 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 concurrency options for CCSM4:

- fully sequential
- fully sequential except the ocean running concurrently
- fully sequential except the ice and land running concurrently with each other
- atmosphere running sequentially with the land and ice which are running concurrently and then the ocean running concurrently with everything
- finally, it makes best sense for the coupler to run on a subset of the atmosphere processors and that can be sequentially or concurrently 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 hardwire 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 `$CASE.$MACH.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, run cases multiple times.

The Run

Setting the time limits

Before you can run the job, you need to make sure the batch queue variables are set correctly for the specific run being targeted. This is done currently by manually editing `$CASE.$MACH.run`. The user should carefully check the batch queue submission lines and make sure that you have appropriate account numbers, time limits, and stdout file names. 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, the user can maximize $CASE.$MACH.run queue limit and change $STOP_OPTION and $STOP_N in `env_run.xml`. For example, say a model's throughput is 4.7 simulated_years/day. On bluefire, the maximum runtime limit is 6 hours. 4.7 model years/24 hours * 6 hours = 1.17 years. On the massively parallel computers, there is always some variability in how long it will take a job to run. On some machines, you may need to leave as much as 20% buffer time in your run to guarantee that jobs finish reliably before the time limit. For that reason we will set our model to run only one model year/job. Continuing to assume that the run is on bluefire, in `$CASE.bluefire.run` set

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

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

```
./xmlchange -file env_run.xml -id STOP_OPTION   -val nyears
./xmlchange -file env_run.xml -id STOP_N        -val 1
./xmlchange -file env_run.xml -id REST_OPTION   -val nyears
./xmlchange -file env_run.xml -id REST_N        -val 1
```

Submitting the run

Once you have configured and built the model, submit `$CASE.$MACH.run` to your machine's batch queue system. For example on NCAR's IBM, bluefire,

```
> # for BLUEFIRE
> bsub < $CASE.bluefire.run
> # for CRAY
> qsub $CASE.jaguar.run
```

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

When executed, the run script, `$CASE.$MACH.run`, will:

- Check to verify that the env files are consistent with the configure and build scripts
- Verify that required input data is present on local disk (in $DIN_LOC_ROOT_CSMDATA) and run the buildnml script for each component
- Run the CCSM model. Put timing information in `$LOGDIR/timing`. If $LOGDIR is set, copy log files back to $LOGDIR
- If $DOUT_S is TRUE, component history, log, diagnostic, and restart files will be moved from $RUNDIR to the short-term archive directory, $DOUT_S_ROOT.
- If $DOUT_L_MS is TRUE, the long-term archiver, `$CASE.$MACH.l_archive`, will be submitted to the batch queue upon successful completion of the run.

- If $RESUBMIT >0, resubmit `$CASE.$MACH.run`

NOTE: This script does NOT execute the build script, `$CASE.$MACH.build`. Building CCSM is now done only via an interactive call to the build script.

If the job runs to completion, you should have "SUCCESSFUL TERMINATION OF CPL7-CCSM" near the end of your STDOUT file. New data should be in the subdirectories under $DOUT_S_ROOT, or if you have long-term archiving turned on, it should be automatically moved to subdirectories under $DOUT_L_MSROOT.

If the job failed, there are several places where you should look for information. 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 runtime errors for more information.

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

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 `env_conf.xml` 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.ocn.ovf
- rpointer.ocn.restart

If short-term archiving is turned on, then the model archives the component restart datasets and pointer files into `$DOUT_S_ROOT/rest/yyyy-mm-dd-sssss`, where yyyy-mm-dd-sssss is the model date at the time of the restart (see below for more details). If long-term archiving these restart then archived in `$DOUT_L_MSROOT/rest`. DOUT_S_ROOT and DOUT_L_MSROOT are set in `env_run.xml`, and can be changed at any time during the run.

Backing up to a previous restart

If a run encounters problems and crashes, the user will normally have to back up to a previous restart. Assuming that short-term archiving is enabled, the user needs to find the latest `$DOUT_S_ROOT/rest/yyyy-mm-dd-ssss/` directory that was create and copy the contents of that directory into their run directory ($RUNDIR). The user 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.

Data flow during a model run

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.

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. Furthermore, if short-term archiving is disabled, then long-term archiving will not be allowed.

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:

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

`hist/` contains component history output for the run.

`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

For long production runs that generate many giga-bytes of data, the user normally wants to move the output data from local disk to a long-term archival location. Long-term archiving can be activated by setting $DOUT_L_MS to TRUE in `env_run.xml`. By default, the value of this variable is FALSE, and long-term archiving is disabled. If the value is set to TRUE, then the following additional variables are: $DOUT_L_MSROOT, $DOUT_S_ROOT DOUT_S (see variables for output data management).

As was mentioned above, if long-term archiving is enabled, files will be moved out of $DOUT_S_ROOT to $DOUT_L_ROOT by `$CASE.$MACH.l_archive`„ which is run as a separate batch job after the successful completion of a model run.

Testing a case

After the case has built and has demonstrated the ability to run via a short test, it is important to formally test exact restart capability before a production run is started. See the Section called *create_production_test* in Chapter 8 for more information about how to use create_production_test.

Chapter 6. Post Processing CCSM Output

A completed run of CCSM4.0 will produce a large number of files, each with many variables. Post processing these files presents several unique challenges. Various techniques and tools have been developed to meet these challenges. Each component maintains its own post-processing utility suite that can be accessed from the release code repository[1]. Component post-processing utilities are currently provided only as a service to the community. Informal community support is provided via the CCSM bulletin board[2]. General questions should be submitted to this bulletin board; however, any bugs found in the packages should be reported to the appropriate working group liaison.

Once approval is granted for access to the release repository, users may freely download the diagnostic packages. Users are urged to 'export' the packages from the repository rather than checking them out. For example:

```
> svn export https://svn-ccsm-release.cgd.ucar.edu/model_diagnostics/atm/cam/amwg_diag2
```

will result in the user obtaining the post processing utilities "amwg_diag2.5". For more details on utilizing the release repository, please see Downloading CCSM for more details.

Notes

1. https://svn-ccsm-release.cgd.ucar.edu/model_diagnostics
2. http://bb.cgd.ucar.edu/

Chapter 7. Porting CCSM

One of the first steps many users will have to address is getting the CCSM4 model running on their local machine. This section addresses that step. This section will describe two different ways of going about that. First, using a generic machine to setup a case, get that case running, then backing out the new machine settings. Second, setting up some new machine settings, creating a case, testing it, and iterating on the machine settings. There are similarities and overlap in both methods. The generic method is likely to produce a running case faster. But eventually, users will want to setup the CCSM4 scripts so their local machine is supported out-of-the-box. This greatly eases setting up cases and benefits groups of users by requiring the port be done only once. Finally, some steps to validate the model will be recommended.

Note: When porting using either of the two methods described above, you will want to initially get a dead, X, compset running at a low resolution. So you could, for instance, start with an X compset at resolution f45_g37. This allows you to determine whether all prerequisite software is in place and working. Once that is working move to an A compset with resolution f45_g37. Once that's working, run a B compset at resolution f45_g37. Finally when all the previous steps have run correctly, run your target compset and resolution.

Porting to a new machine

Porting using a generic machine

This section describes how to setup a case using a generic machine name and then within that case, how to modify the scripts to get that case running on a local machine. In this section, the case name test1 and the generic machine generic_linux_intel will be used in the example. But the specific casename, generic machine, resolution, and compset to test is at the discretion of the user.

1. Run **create_newcase** choosing a generic machine name that is closest to the local machine type. Typing

   ```
   > create_newcase -l
   ```

   will provide a list of possible machines. The generic machines start with the name "generic_". The generic machines are different from the supported machines because extra inline documentation is provided and the user will have to modify some of the resolved scripts.

   Additional command line arguments are required for the generic machines to help setup some of the local environment variables. Typing

   ```
   > create_newcase -h
   ```

   provides a description of the command line arguments. The **create_newcase** will look something like this for a generic machine

   ```
   > cd ccsm4/scripts
   > create_newcase -case test1 \
                    -res f19_g16 \
                    -compset X \
                    -mach generic_linux_intel \
                    -scratchroot /ptmp/username \
                    -din_loc_root_csmdata /home/ccsm/inputdata \
                    -max_tasks_per_node 8 \
   ```

2. Run configure.

   ```
   > cd test1
   > configure -case
   ```

If there are errors at this step, the best approach might be to port starting from the machine files instead of a generic machine. See the Section called *Porting via user defined machine files*.

3. Edit the scripts to be consistent with the local machine. Search for "GENERIC_USER" in the scripts. That tag will highlight inline documentation and areas that will likely need to be modified. In particular, modifications will be needed in the following files.
   - env_mach_specific is where modules, paths, or machine environment variables need to be set. See the "GENERIC_USER" inline documentation in that file.
   - Macros.generic_linux_intel is a Macros file for gmake for the system. In general, that entire file should be reviewed but there are some particular comments about setting the paths for the netcdf and mpi external libraries. See the "GENERIC_USER" inline documentation in that file. While CCSM supports use of pnetcdf in pio, it's generally best to ignore that feature during initial porting. pio works well with standard netcdf.
   - test1.generic_linux_intel.run is the job submission script. Modifications are needed there to address the local batch environment and the job launch. See the "GENERIC_USER" inline documentation in that file.

4. Build the case

   ```
   > ./test1.generic_linux_intel.build
   ```

   This step will often fail if paths to compilers, compiler versions, or libraries are not set properly, if compiler options are not set properly, or if machine environment variables are not set properly. Review and edit the env_mach_specific and Macros.generic_linux_intel files, clean the build,

   ```
   > ./test1.generic_linux_intel.clean_build
   ```

   and try rebuilding again.

5. Run the job using the local job submission command. qsub is used here for example.

   ```
   > qsub test1.generic_linux_intel.run
   ```

   The job will fail to submit if the batch commands are not set properly. The job could fail to run if the launch command is incorrect or if the batch commands are not set consistent with the job resource needs. Review the run script and try resubmitting.

6. Once a case is running, then the local setup for the case can be converted into a specific set of machine files, so future cases can be setup using the user defined machine name, not the generic machine, and cases should be able to run out-of-the-box. This step is very similar to the steps associated with porting using user defined machine files, see the Section called *Porting via user defined machine files*.

   Basically, files in ccsm4/scripts/ccsm_utils/Machines will be added or modified to support the user defined machine out-of-the-box. An env_machopts, Macros, and mkbatch file will be added and the config_machines.xml file will be modified. First, pick a name that will be associated with the local machine. Generally, that's the name of the local machine, but it could be anything. bugsbunny will be used in the description to follow and the bugsbunny setup will be based on the test1 example case above that is running on bugsbunny. To add bugsbunny to the list of supported machines, do the following
   - Edit ccsm4/scripts/ccsm_utils/Machines/config_machines.xml. Add an entry for bugsbunny. A good idea is to copy one of the existing entries and then edit it. The machine specific env variables that need to be set in config_machines.xml for bugsbunny are already set in the env files in the test1 case directory that was created from the generic machine. Those values can be translated directly into the config_machines.xml files for

bugsbunny. That's a starting point anyway. In some cases, variables might need to be made more general. For instance, the port person's user name and the initial test case should not appear in the variable definitions.

- Copy the env_mach_specific file from the test1 case directory to ccsm4/scripts/ccsm_utils/Machines as follows

```
> cd ccsm4/scripts/test1
> cp env_mach_specific ../ccsm_utils/Machines/env_machopts.bugsbunny
```

- Copy the Macros file from the test1 case directory to ccsm4/scripts/ccsm_utils/Machines as follows

```
> cd ccsm4/scripts/test1
> cp Macros.generic_linux_intel  ../ccsm_utils/Machines/Macros.bugsbunny
```

Then edit the ccsm4/scripts/ccsm_utils/Machines/Macros.bugsbunny file and delete everything up to the lines

```
#==============================================================================
# The following always need to be set
```

That first section of the Macros file is added automatically when a case is configured so should not be included in the machine specific setting.

- Create a mkbatch.bugsbunny file in ccsm4/scripts/ccsm_utils/Machines. The easiest way to do this is probably to copy the mkbatch.generic_linux_intel file from that directory to mkbatch.bugsbunny

```
> cd ccsm4/scripts/ccsm_utils/Machines
> cp mkbatch.generic_linux_intel mkbatch.bugsbunny
```

Then edit the mkbatch.bugsbunny to match the changes made in the test1.generic_linux_intel.run file in the test1 case. Remove the GENERIC_USER inline documentation and where that documentation existed, update the batch commands and job launch commands to be consistent with the test1 run script. The first part of the mkbatch script computes values that can be used in the batch commands. It might require some extra iteration to get this working for all cases, processor counts, and processor layouts.

- Test the new machine setup. Create a new case based on test1 using the bugsbunny machine setup

```
> cd ccsm4/scripts
> create_newcase -case test1_bugsbunny \
                 -res f09_g16 \
                 -compset X \
                 -mach bugsbunny
```

Then configure, build, and run the case and confirm that test1_bugsbunny runs fine and is consistent with the original test1 case. Once that works, test other configurations then move to port validation, see the Section called *Port Validation*.

Porting via user defined machine files

This section describes how to add support for a new machine using machine specific files. The basic approach is to add support for the new machine to the CCSM4 scripts directly and then to test and iterate on that setup. Files in ccsm4/scripts/ccsm_utils/Machines will be added or modified to support the user defined machine out-of-the-box. An env_machopts, Macros, and mkbatch file will be added and the config_machines.xml file will be modified. First, pick a name that will be associated with the local machine. Generally, that's the name of the local machine, but it could be anything. wilycoyote will be used in the description to follow. It's also helpful to identify an existing supported machine that is similar to your machine to use as a starting point in porting. If the user defined machine is a

linux cluster with an intel compiler, then after reviewing the current supported machines using

```
> cd ccsm4/scripts
> ./create_newcase -l
```

dublin_intel, hadley, or generic_linux_intel would be good candidates as starting points. Starting with a generic machine provides some additional inline documentation to aid in porting. If a generic machine is used, search for the tag "GENERIC_USER" in the scripts for additional documentation. In the example below, dublin_intel will be used as the starting point. To add wilycoyote to the list of supported machines, do the following

- Edit ccsm4/scripts/ccsm_utils/Machines/config_machines.xml. Add an entry for wilycoyote. A good idea is to copy one of the existing entries and then edit the values for wilycoyote. You could start with the dublin_intel settings although nearly any machine will be ok. There are several variable settings here. The definition of these variables can be found in the appendix, see Appendix D, Appendix E, Appendix F, Appendix G, and Appendix H. Some of the important ones are MACH which should be set to wilycoyote, EXEROOT which should be set to a generic working directory like /tmp/scratch/$CCSMUSER/$CASE, DIN_LOC_ROOT_CSMDATA which should be set to the path to the ccsm inputdata directory, BATCHQUERY and BATCHJOBS which specify the query and submit command lines for batch jobs and are used to chain jobs together in production, and MAX_TASKS_PER_NODE which set the maximum number of tasks allowed on each hardware node.
- Copy an env_machopts file to env_machopts.wilycoyote. Start with the dublin_intel file.

  ```
  > cd ccsm4/scripts/ccsm_utils/Machines
  > cp env_machopts.dublin_intel env_machopts.wilycoyote
  ```

  Edit env_machopts.wilycoyote to change the environment setup, paths, modules, and environment variables to be consistent with wilycoyote.
- Copy a Macros file to Macros.wilycoyote. Start with the dublin_intel file.

  ```
  > cd ccsm4/scripts/ccsm_utils/Machines
  > cp Macros.dublin_intel Macros.wilycoyote
  ```

  Then review and edit the Macros.wilycoyote file as needed. Pay particular attention to the compiler name, and the netcdf and mpi paths. While the compiler options for a given compiler are pretty consistent across machines, invoking the compiler and the local paths for libraries are not. While CCSM supports use of pnetcdf in pio, it's generally best to ignore that feature during initial porting. pio works well with standard netcdf.
- Copy a mkbatch file to mkbatch.wilycoyote file. Start with the dublin_intel file.

  ```
  > cd ccsm4/scripts/ccsm_utils/Machines
  > cp mkbatch.dublin_intel mkbatch.wilycoyote
  ```

  Then edit the mkbatch.wilycoyote to be consistent with wilycoyote. In particular, the batch commands and the job launching will probably need to be changed. The batch commands and setup are the first section of the script. The job launching can be found by searching for the string "CSM EXECUTION".
- After an initial pass is made to setup the new machine files, try creating a case, building and running. Getting this to work will be an iterative process. Changes will probably be made in both the machine files in ccsm4/scripts/ccsm_utils/Machines for wilycoyote and in the case as testing proceeds. Whenever the machine files are updated, a new case should be setup. Whenever something is changed in the case scripts to fix a problem, that change should be migrated back to the wilycoyote machine files. In general, it's probably easiest to modify the machine files and create new cases until the case configures successfully. Once the case is configuring, it's often easiest to edit the case scripts

to fix problems in the build and run. Once a case is running, those changes in the case need to be backed out into the wilycoyote machine files and then those machine files can be tested with a new case.

```
> cd ccsm4/scripts
> create_newcase -case test_wily1 \
                 -res f19_g16 \
                 -compset X \
                 -mach wilycoyote
> cd test_wily1
> configure -case
> ./test_wily1.wilycoyote.build
> qsub test_wily1.wilycoyote.run
```

Eventually, the machine files should work for any user and any configuration for wilycoyote.

Port Validation

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.f19_g16.X
   ERI.T31_g37.A
   ERI.f19_g16.B1850CN
   ERS.f19_f19.F
   ERS.f19_g16.I
   ERS.T62_g16.C
   ERS.T62_g16.D
   ERT.f19_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 CAM error growth test[1].
   b. Perform a CLM perturbation error growth test (as described in the CLM User's Guide[2]).
   c. Follow the CCSM4.0 CICE port-validation procedure.[3]
   d. Follow the CCSM4.0 POP2 port-validation procedure.[4]
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 [5]). Model output data for these runs will be available on the Earth System Grid (ESG) [6] as well.

Notes

1. http://www.ccsm.ucar.edu/models/atm-cam/port/
2. http://www.ccsm.ucar.edu/models/ccsm4.0/clm/models/lnd/clm/doc/UsersGuide/book1.html
3. http://www.ccsm.ucar.edu/models/ccsm4.0/cice/validation/index.html
4. http://www.ccsm.ucar.edu/models/ccsm4.0/pop/validation/index.html
5. http://www.ccsm.ucar.edu/experiments/ccsm4.0/diagnostics/
6. http://www.ccsm.ucar.edu/models/ccsm4.0/model_esg/

Chapter 8. CCSM Testing

Testing overview

CCSM4 has a few tools that support automated testing of the model. In general, these should be used only after the model has been ported to the target machine (see Chapter 7). The tools are create_production_test, create_test, and create_test_suite. The create_production_test tool is executed from a working case, and it tests exact restartability of that case setup in a separate directory. The create_test tool allows a user to quickly setup and run one of several supported tests. The create_test_suite tool quickly allows a user to setup and run a list of supported tests. Each of these tools will be described below.

create_production_test

In general, after configuring and testing a case and before starting a long production job based on that case, it's important to verify that the model restarts exactly. This is a standard requirement of the system and will help demonstrate stability of the configuration technically. The tool create_production_test is located in the case directory, and it sets up an ERT two month exact restart test in a separate directory based on the current case. To use it, do the following

```
> cd $CASEROOT
> ./create_production_test
> cd ../$CASE_ERT.$MACH
> $CASE_ERT.$MACH.build
submit $CASE_ERT.$MACH.run
Check your test results. A successful test produces "PASS" as
  the first word in the file, $CASE_ERT.$MACH/TestStatus
```

If the test fails, see the Section called *Debugging Failed Tests* for test debugging guidance.

create_test

The create_test tool is located in the scripts directory and can be used to setup a standalone test case. The test cases are fixed and defined within the CCSM scripts. To see the list of test cases or for additional help, type "create_test -help" from the scripts directory. To use create_test, do something like

```
> cd $CCSMROOT/scripts
> ./create_test -testname ERS.f19_g16.X.bluefire -testid t01
> cd ERS.f19_g16.X.bluefire.t01
> ERS.f19_g16.X.bluefire.t01.build
submit ERS.f19_g16.X.bluefire.t01.test
Check your test results. A successful test produces "PASS" as
  the first word in the file TestStatus
```

The above sets up an exact restart test (ERS) at the 1.9x2.5_gx1v6 resolution using a dead model compset (X) for the machine bluefire. The testid provides a unique tag for the test in case it needs to be rerun (i.e. using -testid t02). Some things to note about CCSM tests

- For more information about the create_test tool, run "create_test -help".
- Test results are set in the TestStatus file. The TestStatus.out file provides additional details.

- Tests are not always easily re-runable from an existing test directory. Rather than rerun a previous test case, it's best to setup a clean test case (i.e. with a new testid).
- The costs of tests vary widely. Some are short and some are long.
- If a test fails, see the Section called *Debugging Failed Tests*.
- There are -compare and -generate options for the create_test tool that support regression testing.
- There are extra test options that can be added to the test such as _D, _E, or _P*. These are described in more detail in the create_test -help output.

The test status results have the following meaning

Test Result	Description
BFAIL	compare test couldn't find base result
CHECK	manual review of data is required
ERROR	test checker failed, test may or may not have passed
FAIL	test failed
GEN	test has been generated
PASS	test passed
PEND	test has been submitted
RUN	test is currently running OR it hung, timed out, or died ungracefully
SFAIL	generation of test failed in scripts
UNDEF	undefined result

The following tests are available at the time of writing

Test	Description
SMS	5 day smoke test
ERS	10 day exact restart from startup
ERP	2 month exact restart from startup
ERB	branch/exact restart test
ERH	hybrid/exact restart test
ERI	hybrid/branch/exact restart test
ERT	exact restart from startup, history file test
SEQ	sequencing bit-for-bit test
PEA	single processor testing
PEM	pe counts mpi bit-for-bit test
PET	pe counts mpi/openmp bit-for-bit test
CME	compare mct and esmf interfaces test

create_test_suite

The create_test_suite tool is located in the scripts directory and can be used to setup a suite of standalone test cases automatically. To use this tool, a list of tests needs to ex-

ist in a file. Some examples can be found in the directory scripts/ccsm_utils/Testlists. create_test_suite in invoked on a list of tests and then the full list of tests is generated. In addition, an automated submission script and reporting script are created. The cs.submit script reduces the time to submit multiple test cases significantly. To use this tool, do something like the following

```
create a list of desired tests in some filename, i.e. my_lists
> create_test_suite -input_list my_lists -testid suite01
> ./cs.status.suite01
> ./cs.submit.suite01.$MACH
> ./cs.status.suite01
```

The cs.status script is generated by create_test_suite, and it reports the status of all the tests in the suite. The cs.submit script builds and submits all the tests sequentially. The cs.submit script should only be executed once to build and submit all the tests. The cs.status script can be executed as often as needed to check the status of the tests. When all the tests have completed running, then the results are static and complete. To help debug failed tests, see the Section called *Debugging Failed Tests*..

Debugging Failed Tests

This section describes what steps can be taken to try to identify why a test failed. The primary information associated with reviewing and debugging a run can be found in the Section called *Troubleshooting runtime problems* in Chapter 10.

First, verify that a test case is no longer in the batch queue. If that's the case, then review the possible test results and compare that to the result in the TestStatus file. Next, review the TestStatus.out file to see if there is any additional information about what the test did. Finally, go to the troubleshooting section and work through the various log files.

Finally, there are a couple other things to mention. If the TestStatus file contains "RUN" but the job is no longer in the queue, it's possible that the job either timed out because the wall clock on the batch submission was too short, or the job hung due to some run-time error. Check the batch log files to see if the job was killed due to a time limit, and if it was increase the time limit and either resubmit the job or generate a new test case and update the time limit before submitting it.

Also, a test case can fail because either the job didn't run properly or because the test conditions (i.e. exact restart) weren't met. Try to determine whether the test failed because the run failed or because the test did not meet the test conditions. If a test is failing early in a run, it's usually best to setup a standalone case with the same configuration in order to debug problems. If the test is running fine, but the test conditions are not being met (i.e. exact restart), then that requires debugging of the model in the context of the test conditions.

Not all tests will pass for all model configurations. Some of the issues we are aware of are

- All models are bit-for-bit reproducible with different processor counts EXCEPT pop. The BFBFLAG must be set to TRUE in the env_run.xml file if the coupler is to meet this condition. There will be a performance penalty when this flag is set.
- All models can be run with mixed mpi/openmp parallelism except pop, although the performance of the openmp implementation varies widely. Efforts are underway now to update pop to support openmp usage.
- Some of the active components cannot run with the mpi serial library. This library takes the place of mpi calls when the model is running on one processors and MPI is not available or not desirable. The mpi serial library is part of the CCSM release and is invoked by setting the USE_MPISERIAL variable in env_build.xml to TRUE.

An effort is underway to extend the mpi serial library to support all components' usage of the mpi library with this standalone implementation.

Chapter 9. Use Cases

The basic example

This specifies all the steps necessary to create, configure, build, and run a case. The following assumes that $CCSMROOT is /user/ccsmroot.

1. Create a new case named b40.B2000 in the ~/ccsm4 directory. Use a present-day control compset at 1-degree resolution on bluefire.

```
> cd /user/ccsmroot
> create_newcase -case ~/ccsm4/b40.B2000 \
                 -compset B_2000 \
                 -res 0.9x1.25_gx1v6 \
                 -mach bluefire
```

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

```
> cd ~/ccsm4/b40.B2000
> ./configure -case
> b40.B200.bluefire.build
```

3. Create a production test. Go to the test directory. Build the test first, then run the test and check the TestStatus (the first word should be PASS).

```
> ./create_production_test
> cd ../b40.B2000_ERT
> b40.B2000_ERT.bluefire.build
> bsub < b40.B2000_ERT.bluefire.test
> cat TestStatus
```

4. Go back to the case directory, set the job to run 12 model months, use an editor to change the time limit in the run file to accommodate a 12-month run, and submit the job.

```
> cd ../b40.B2000
> xmlchange -file env_run.xml -id STOP_OPTION -val nmonths
> xmlchange -file env_run.xml -id STOP_N -val 12
> # use an editor to change b40.B2000.bluefire.run "#BSUB -W 1:30" to "#BSUB -W 6:
> bsub < b40.B2000.bluefire.run
```

5. Make sure the run succeeded.

```
> grep "SUCCESSFUL TERMINATION" poe.stdout.*
```

6. 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 -file env_run.xml -id RESUBMIT -val 10
> bsub < b40.B2000.bluefire.run
```

Setting up a branch or hybrid run

The section setting the case initialization discussed starting a new case as a branch run or hybrid run by using data from a previous run. First you need to create a new case. Assume that $CCSMROOT is set to /user/ccsmroot and that $EXEROOT is `/ptmp/$user/b40.B2000p`. Finally, assume that the branch or hybrid run is being carried out on NCAR's IBM system, bluefire.

```
> cd /user/ccsmroot/scripts
> create_newcase -case ~/ccsm4/b40.B2000p \
                 -compset B_2000 \
```

```
                -res 0.9x1.25_gx1v6 \
                -mach bluefire
> cd ~/ccsm4/b40.B2000p
```

For a branch run, modify `env_conf.xml` to branch from b40.B2000 at year 0001-02-01.

```
> xmlchange -file env_conf.xml -id RUN_TYPE    -val branch
> xmlchange -file env_conf.xml -id RUN_REFCASE -val b40.B2000
> xmlchange -file env_conf.xml -id RUN_REFDATE -val 0001-02-01
```

For a hybrid run, modify `env_conf.xml` to start up from b40.B2000 at year 0001-02-01.

```
> xmlchange -file env_conf.xml -id RUN_TYPE    -val hybrid
> xmlchange -file env_conf.xml -id RUN_REFCASE -val b40.B2000
> xmlchange -file env_conf.xml -id RUN_REFDATE -val 0001-02-01
```

For a branch run, `env_conf.xml` for b40.B2000p should be identical to b40.B2000, except for the $RUN_TYPE setting. In addition, any modifications introduced into `~/ccsm4/b40.B2000/Buildconf/$component.buildnml.csh`, should be re-introduced into b40.B2000p.

Configure and build the case executable.

```
> configure -case
> b40.B200p.bluefire.build
```

Prestage the necessary restart/initial data in $RUNROOT (assumed to be `/ptmp/$user/b40.B2000p/run`). Note that `/ptmp/$user/b40.B2000p/run` was created during the build. Assume that the restart/initial data is on the NCAR mass store.

```
> cd /ptmp/$user/b40.B2000br/run
> msrcp "mss:/CCSM/csm/b40.B2000/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, use an editor to change the time limit in the run file to accommodate a 12-month run, then submit the job.

```
> ~/ccsm4/b40.B2000p
> xmlchange -file env_run.xml -id STOP_OPTION -val nmonths
> xmlchange -file env_run.xml -id STOP_N -val 12
> # use an editor to change b40.B2000.bluefire.run "#BSUB -W 1:30" to "#BSUB -W 6:00"
> bsub < b40.B2000p.bluefire.run
```

Verify that the run succeeded.

```
> grep "SUCCESSFUL TERMINATION" poe.stdout.*
```

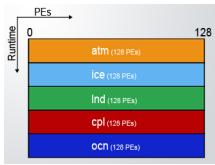
Change the run to a continuation run. Set it to resubmit itself 10 times so that it will run a total of 11 years (including the initial year), then resubmit the case.

```
> xmlchange -file env_run.xml -id CONTINUE_RUN -val TRUE
> xmlchange -file env_run.xml -id RESUMIT -val 10
> bsub < b40.B2000p.bluefire.run
```

Changing PE layout

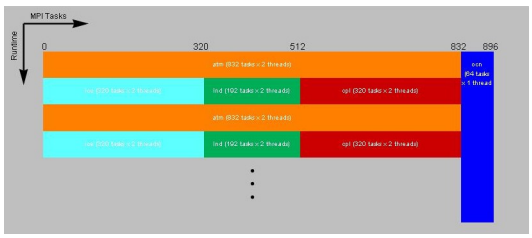
This example modifies the PE layout for our original run, b40.B2000. We now target the model to run on the jaguar supercomputer and modify our PE layout to use a common load balance configuration for CCSM on large CRAY XT5 machines.

In our original example, b40.B2000, we used 128 pes with each component running sequentially over the entire set of processors.



128-pes/128-tasks layout

Now we change the layout to use 1728 processors and run the ice, lnd, and cpl models concurrently on the same processors as the atm model while the ocean model will run on its own set of processors. The atm model will be run on 1664 pes using 832 MPI tasks each threaded 2 ways and starting on global MPI task 0. The ice model is run using 320 MPI tasks starting on global MPI task 0, but not threaded. The lnd model is run on 384 processors using 192 MPI tasks each threaded 2 ways starting at global MPI task 320 and the coupler is run on 320 processors using 320 MPI tasks starting at global MPI task 512. The ocn model uses 64 MPI tasks starting at global MPI task 832.



1728-pes/896-tasks layout

Since we will be modifying `env_mach_pes.xml` after `configure` was invoked, the following needs to be invoked:

```
> configure -cleanmach
> xmlchange -file env_mach_pes.xml -id NTASKS_ATM -val 832
> xmlchange -file env_mach_pes.xml -id NTHRDS_ATM -val 2
> xmlchange -file env_mach_pes.xml -id ROOTPE_ATM -val 0
> xmlchange -file env_mach_pes.xml -id NTASKS_ICE -val 320
> xmlchange -file env_mach_pes.xml -id NTHRDS_ICE -val 1
> xmlchange -file env_mach_pes.xml -id ROOTPE_ICE -val 0
> xmlchange -file env_mach_pes.xml -id NTASKS_LND -val 192
> xmlchange -file env_mach_pes.xml -id NTHRDS_LND -val 2
> xmlchange -file env_mach_pes.xml -id ROOTPE_LND -val 320
> xmlchange -file env_mach_pes.xml -id NTASKS_CPL -val 320
> xmlchange -file env_mach_pes.xml -id NTHRDS_CPL -val 1
> xmlchange -file env_mach_pes.xml -id ROOTPE_CPL -val 512
> xmlchange -file env_mach_pes.xml -id NTASKS_OCN -val 64
> xmlchange -file env_mach_pes.xml -id NTHRDS_OCN -val 1
> xmlchange -file env_mach_pes.xml -id ROOTPE_OCN -val 832
> configure -mach
```

Note that since env_mach_pes.xml has changed, the model has to be reconfigured and rebuilt.

It is interesting to compare the timings from the 128- and 1728-processor runs. The timing output below shows that the original model run on 128 pes cost 851 pe-hours/simulated_year. Running on 1728 pes, the model cost more than 5 times as much, but it runs more than two and a half times faster.

```
128-processor case:
Overall Metrics:
```

```
Model Cost: 851.05 pe-hrs/simulated_year (scale= 1.00)
Model Throughput: 3.61 simulated_years/day

1728-processor case:
Overall Metrics:
Model Cost: 4439.16 pe-hrs/simulated_year (scale= 1.00)
Model Throughput: 9.34 simulated_years/day
```

See understanding load balancing CCSM for detailed information on understanding timing files.

Setting CAM output fields

In this example, we further modify our b40.B2000p code to set various CAM output fields. The variables that we set are listed below. See CAM Namelist Variables[1] for a complete list of CAM namelist variables.

`avgflag_pertape`

Sets the averaging flag for all variables on a particular history file series. Default is to use default averaging flags for each variable. Average (A), Instantaneous (I), Maximum (X), and Minimum (M).

`nhtfrq`

Array of write frequencies for each history files series.

When NHTFRQ(1) = 0, the file will be a monthly average. Only the first file series may be a monthly average.

When NHTFRQ(i) > 0, frequency is input as number of timesteps.

When NHTFRQ(i) < 0, frequency is input as number of hours.

`mfilt`

Array of number of time samples to write to each history file series (a time sample is the history output from a given timestep).

`ndens`

Array specifying output format for each history file series. Valid values are 1 or 2. '1' implies output real values are 8-byte and '2' implies output real values are 4-byte. Default: 2,2,2,2,2,2

`fincl1 = 'field1', 'field2', ...`

List of fields to add to the primary history file.

`fincl[2..6] = 'field1', 'field2', ...`

List of fields to add to the auxiliary history file.

`fexcl1 = 'field1', 'field2', ...`

List of field names to exclude from the default primary history file (default fields on the Master Field List).

`fexcl[2..6] = 'field1', 'field2',...`

List of the field names to exclude from the auxiliary history files.

In the `$CASEROOT/Buildconf/cam.buildnml.csh` file, namelists are delineated with an ampersand followed by the namelist's name. Namelists end with a slash. For example, the first namelist might look like this:

```
& phys_ctl_nl
atm_dep_flux   = .false.
deep_scheme    = 'ZM'
eddy_scheme    = 'HB'
microp_scheme  = 'RK'
shallow_scheme = 'Hack'
srf_flux_avg = 0
/
```

Just before the ending slash for the cam_inparm namelist, add the following lines:

```
avgflag_pertape = 'A','I'
nhtfrq = 0 ,-6
mfilt  = 1 , 30
ndens  = 2 , 2
fincl1 = 'FSN200','FSN200C','FLN200',
         'FLN200C','QFLX','PRECTMX:X','TREFMXAV:X','TREFMNAV:M',
         'TSMN:M','TSMX:X'
fincl2 = 'T','Z3','U','V','PSL','PS','TS','PHIS'
```

avgflag_pertape specifies how the output data will be averaged. In the first output file, `b40.2000p.cam2.h0.yyyy-mm.nc`, data will be averaged monthly. In the second output file, `b40.2000p.cam2.h1.yyyy-mm-dd.nc`, data will be instantaneous.

nhtfrq sets the frequency of data writes, so `b40.2000p.cam2.h0.yyyy-mm.nc` will be written as a monthly average, while `b40.2000p.cam2.h1.yyyy-mm-dd.nc` will contain time slices that are written every 6 hours.

mfilt sets the model to write one time sample in `b40.2000p.cam2.h0.yyyy-mm.nc` and 30 time samples in `b40.2000p.cam2.h1.yyyy-mm-dd.nc`.

ndens sets both files to have 32-bit netCDF format output files.

fincl1 sets the output fields for b40.2000p.cam2.h0.yyyy-mm.nc. A complete list of the CAM output fields appears here. In this example, we've asked for more variables than will fit on a Fortran line. As you can see, it is all right to split variable lists across lines. Also in this example, we've asked for maximum values of TREFMXAV and TSM, and minimum values of TREFMNAV and TSMN.

fincl2 sets the output fields for b40.2000p.cam2.h1.yyyy-mm-dd.nc, much the same as fincl1 sets output fields for b40.2000p.cam2.h0.yyyy-mm.nc, only in this case, we are asking for instantaneous values rather than averaged values, and choosing different output fields.

Setting CAM forcings

To set the greenhouse gas forcings, we must first understand the namelist variables associated with them. See CAM Namelist Variables[2] for a complete list of CAM namelist variables.

`scenario_ghg`

Controls treatment of prescribed co2, ch4, n2o, cfc11, cfc12 volume mixing ratios. May be set to 'FIXED' or 'RAMPED' or 'RAMP_CO2_ONLY'.

FIXED => volume mixing ratios are fixed and have either default or namelist input values.

RAMPED => volume mixing ratios are time interpolated from the dataset specified by bndtvghg.

RAMP_CO2_ONLY => only co2 mixing ratios are ramped at a rate determined by the variables ramp_co2_annual_rate, ramp_co2_cap, and ramp_co2_start_ymd.

Default: FIXED

`bndtvghg`

Full pathname of time-variant boundary dataset for greenhouse gas surface values. Default: set by build-namelist.

`rampyear_ghg`

If scenario_ghg is set to "RAMPED" then the greenhouse gas surface values are interpolated between the annual average values read from the file specified by bndtvghg. In that case, the value of this variable (> 0) fixes the year of the lower bounding value (i.e., the value for calendar day 1.0) used in the interpolation. For example, if rampyear_ghg = 1950, then the GHG surface values will be the result of interpolating between the values for 1950 and 1951 from the dataset. Default: 0

To set the following variables to their associated values, edit $CASEROOT/Buildconf/cam.buildnml.csh and add the following to the cam_inparm namelist:

```
scenario_ghg = 'RAMPED'
bndtvghg = '$DIN_LOC_ROOT/atm/cam/ggas/ghg_hist_1765-2005_c091218.nc'
rampyear_ghg = 2000
```

Initializing the ocean model with a spun-up initial condition

The recommended method for initializing the CCSM active ocean model (pop2) in a CCSM startup case is to use the default settings; these initialize the ocean model from a state of rest. Occasionally, however, researchers are interested in initializing the ocean model from a "spun up" ocean condition that was generated from an existing CCSM run. To accommodate their request, a nonstandard method of initializing the model was developed. It is called the startup_spunup option. The startup_spunup initialization is a research option that is designed for use by expert users only.

Because of the complex interactions between the ocean-model parameterizations used to generate the spun-up case, $CASE_SP, and those used in the new startup case, it is impossible to provide a single recommended spun-up ocean initial condition for all cases. Instead, researchers must carefully select an existing solution whose case conditions closely match those in the new case. A mismatch of options between the spun-up case and the new case can result in scientifically invalid solutions.

When a startup_spunup case is necessary, use this procedure:

1. Create a new case, $CASE. By default, $CASE will be a "startup run."

```
> create_newcase -case ~/ccsm4/b40.B2000ocn \
                 -mach bluefire \
                 -compset B20TR \
                 -res 0.9x1.25_gx1v6
> cd ~/ccsm4/b40.B2000ocn
> configure -case
```

2. Specify the startup_spunup option in the pop2_in namelist file by editing the $CASE/Buildconf/pop2.buildnml.csh script. Find the namelist init_ts_nml and change

   `set init_ts_suboption = 'null'` to

   `set init_ts_suboption = 'spunup'.`

3. The ocean restart filename is of the form ${CASE_SP}.pop.r.$date, where $date is the model date of your spun-up dataset. If the ocean restart files were written in binary format, a companion ascii-formatted restart "header" file will also exist. The companion header file will have the same name as the restart file, except that it will have the suffix ".hdr" appended at the end of the filename. You must copy both the binary restart file and the header file to your data directory.
4. The spun-up ocean restart and restart header files must be available to your new case. Copy them directly into $RUNDIR. It is critically important to copy both the binary restart file and its companion header file to the $RUNDIR.

   ```
   > cp ${CASE_SP}.pop.r.$date       $RUNDIR
   > cp ${CASE_SP}.pop.r.${date}.hdr $RUNDIR
   ```

5. Redefine the ocean-model initial-condition dataset by editing $CASE/Buildconf/pop2.buildnml.csh. Go to the pop2_in namelist section and edit the init_ts_nml namelist variable init_ts_file. Change

   `set init_ts_file = '$init_ts_filename'` to

   `set init_ts_file = '${CASE_SP}.pop.r.$date'`

   Note that the model will automatically look for the ${CASE_SP}.pop.r.${date}.hdr file in $RUNDIR.
6. Build and run as usual.

Taking a run over from another user

If you ever a need to take over a production run from another user, follow this procedure:

1. Create a clone of the production case. The case name needs to be the same, but the new filepath needs to be different.

   ```
   > $CCSMROOT/scripts/create_clone -clone $CASEROOT -case $NEWCASEROOT
   ```

2. Configure the case for the new user:

   ```
   > cd $NEWCASEROOT
   > configure -case
   ```

3. Rebuild for the new user:

   ```
   > ./$CASE.$MACH.build
   ```

4. Copy the restart and rpointer files from the original run directory:

   ```
   > cp $CASEROOT/run/$CASE* $NEWCASEROOT/run/.
   > cp $CASEROOT/run/rpointer* $NEWCASEROOT/run/.
   ```

5. Copy the archive directory contents:

   ```
   > cp $ORIGDIR/archive/$CASE $NEWDIR/archive/$CASE
   ```

6. Submit the run:

   ```
   > bsub < $CASE.$MACH.run
   ```

Here is an example:

```
> $CCSMROOT/scripts/create_clone -clone /user/b40.1850 -case /newuser/b40.B2000
> cd /newuser/b40.B2000
> configure -case
> ./b40.B2000.bluefire.build
> cp /user/b40.B2000/run/b40.B2000* /newuser/b40.B2000/run/.
> cp /user/b40.B2000/run/rpointer* /newuser/b40.B2000/run/.
> cp -r /ptmp/user/archive/b40.B2000/* /ptmp/newuser/archive/b40.B2000/.
> bsub < b40.B2000.bluefire.run
```

Use of an ESMF library and ESMF interfaces

CCSM4 supports use of either the CCSM designed component interfaces which is based on MCT datatypes or ESMF compliant component interfaces. In both cases, the driver and component models remain fundamentally the same. The ESMF interface implementation exists in CCSM to support further development and testing of an ESMF driver or ESMF couplers and to allow CCSM model components to interact with other coupled systems using ESMF coupling standards. The ESMF implementation in CCSM4 has been tested with ESMF version 4.0.0rp1. The MCT based interfaces are used by default.

CCSM4 does not require the ESMF library nor that a local copy of the ESMF library exist. A small subset of local Fortran code exists in CCSM to support CCSM usage of the ESMF_Clock in the model. ESMF is not provided as part of the CCSM4 release. It must be downloaded[3] and installed separately. If an ESMF library has been installed locally, the ESMF_Clock Fortran code provided with CCSM can be turned off, and the ESMF library can be used in CCSM via setting of the environment variables USE_ESMF_LIB and ESMF_LIBDIR. ESMF_LIBDIR specifies the path to the local ESMF library and USE_ESMF_LIB is a logical that specifies whether to use the external ESMF library. In general, the MCT interfaces can be run with either an external ESMF library or the local Fortran code. However, the ESMF interfaces must be run with an official version of the ESMF library.

To run with the ESMF interfaces, set the following env variables and rebuild a clean version of the model.

```
- cd to your case directory
- edit env_build.xml
  - set COMP_INTERFACE to "ESMF"
  - set USE_ESMF_LIB to "TRUE"
  - set ESMF_LIBDIR to a valid installation directory of ESMF version 4.0.0rp1
- run *.clean_build to remove any previous build if necessary
- run *.build to build the model
- submit the *.run script
```

To run with the MCT interfaces, set the following env variables and rebuild a clean version of the model.

```
- cd to your case directory
- edit env_build.xml
  - set COMP_INTERFACE to "MCT"
  - set USE_ESMF_LIB to "TRUE" or "FALSE"
  - if USE_ESMF_LIB is TRUE, then set ESMF_LIBDIR to a valid installation of ESMF versi
    directory of ESMF version 4
- run *.clean_build to remove any previous build if necessary
- run *.build to build the model
- submit the *.run script
```

Notes

1. http://www.ccsm.ucar.edu/cgi-bin/eaton/namelist/nldef2html-pub/
2. http://www.ccsm.ucar.edu/cgi-bin/eaton/namelist/nldef2html-pub/
3. http://www.earthsystemmodeling.org/download/index.shtml

Chapter 10. Troubleshooting

Troubleshooting create_newcase

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 configure

Generally, configure errors are reported to the terminal and should provide some guidance about what caused the error. Most of this section deals with the "-case" option of configure which is the biggest step in setting up a case and the supporting input files and scripts. The configure step is fairly extensive and a brief description of what configure does follows. $CASEROOT is the top level case directory (i.e. the location where the env xml files and the configure script is located for a specific case).

The first thing configure does is load the case environment variables. In general, this is done by sourcing the $CASEROOT/Tools/ccsm_getenv. For more information about the environment variables, see the Section called *What's the deal with the CCSM4 env variables and env xml files?* in Chapter 11

Then the first major step for configure is to run the script $CASEROOT/Tools/generate_resolved.csh. This cycles through each of the component template files in $CASEROOT/Tools/Templates sequentially. These component template files are copied from locations in the component source code by create_newcase when the case is created. Each component template file generates a component buildnml.csh and buildexe.csh script in $CASEROOT/Buildconf based on the resolution, configuration, and other env settings. Generally, an error in this phase of configure will point to a specific component. Begin by debugging the component template file in $CASEROOT/Tools/Templates. The component template filename will be something like cam.cpl7.template for the cam component. If there is a bug in the component template file, then it's probably important to fix the original copy of the template file. These can be found in the create_newcase scripts (i.e. search for the string, template).

The specific implementation of the component template files is very much component dependent. However, each must generate a buildnml.csh script in the $CASEROOT/Buildconf directory to generate namelist input on-the-fly. Each template file must generate a buildexe.csh script in the same $CASEROOT/Buildconf directory to support the build of that component. And each template file must support generation of input_data_list files in the $CASEROOT/Buildconf directory either at the configure or build step to specify the required input files for that configuration and component.

Next, configure runs the $CASEROOT/Tools/generate_batch.csh script. This script generates the build and run scripts for the case. This is partly accomplished by running the mkbatch.$MACH script for the particular machine. That script is located in $CCSMROOT/scripts/ccsm_utils/Machines. If there are problems with the resulting build or run script, the error can usually be traced to the setup of the mkbatch.$MACH machine file.

For instance an error like this

```
> create_newcase -case ~/ccsm4/b40.B2000bad \
```

```
                -res 0.23x0.31_0.23x0.31  \
                -mach bluefire            \
                -compset B
> cd ~/ccsm4/b40.B2000bad
> configure -case

Generating resolved namelist, prestage, and build scripts
build-namelist - No default value found for ncdata
user defined attributes:
key=ic_md val=00010101
Died at /user/ccsmroot/models/atm/cam/bld/build-namelist line 2019.
ERROR: generate_resolved.csh error for atm template
configure error: configure generated error in attempting to created resolved scripts
```

indicates the generate_resolved.csh script failed in the atm template, which is the cam template for this compset. It also reports that the cam build-namelist step in the cam template failed at line 2019. In this case, CAM could not find a valid value of ncdata from its default_namelist.xml file. To fix this particular problem, the user can supply an alternative initial dataset and the update the value in either the CAM_CONFIG_OPTS values or in the `SourceMods/src.cam/user_nl`.

Troubleshooting job submission problems

This section addresses problems with job submission. Most of the problems associated with submission or launch are very site specific.

First, make sure the runscript, $CASE.$MACH.run, is submitted using 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.

Review the batch submission options being used. These probably appear at the top of the $CASE.$MACH.run script but also may be set on the command line when submitting a job. 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.

Review the job launch command in the $CASE.$MACH.run script to make sure it's consistent with the site specific recommended tool. This command is usually an mprun, mpiexec, aprun, or something similar. It can be found just after the string "EXECUTION BEGINS HERE" in the $CASE.$MACH.run script.

The batch and run aspects of the $CASE.$MACH.run script is setup by configure and uses a machine specific mkbatch.$MACH script in the $CCSMROOT/scripts/ccsm_utils/Machines directory. If the run script is not producing correct batch scripts or job launching commands, the mkbatch.$MACH script probably needs to be updated.

Troubleshooting runtime 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, or whether a file system became full. If any of those things happened, take appropriate corrective action and resubmit the job.

If it's not clear any of the above caused a case to fail, then there are several places to look for error messages in CCSM4.

- Go the $RUNDIR directory. This directory is set in the env_build.xml file. This is the directory where CCSM runs. 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 each component log file 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 the $CASEROOT directory. The standard out/err file often captures a significant amount of extra CCSM4 output and it 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.
- Go the $RUNDIR directory. Check for core files 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 =
tStamp_write: model date = 10121 0 wall clock = 2009-09-28 09:12:32 avg dt = 60.10 dt =
```

and 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 CCSM4 model over time. For instance, on the last day of every simulated month, CCSM4 typically write netcdf files, and this can be a significant intermittent cost. Also, some models 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 CCSM4, 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 with CCSM4.

Chapter 11. Frequently Asked Questions (FAQ)

What are all these directories and files in my case directory?

The following describes many of the files and directories in the $CASEROOT directory.

Buildconf

is the directory where the buildnml and buildexe component scripts are generated by configure. The input_data_list files are also generated by configure or the buildnml scripts and copied here.

CaseDocs

is a directory where copies of the latest namelist/text input files from the $RUNDIR are stored. These exist only to help document the case setup and run.

LockedFiles

is the directory that holds copies of the locked files.

MachinesHist

is a directory where previous case configurations are stored. In other words, when configure -clean is run, the current configured scripts are copied into this directory, so when configure -case is subsequently run, there is an opportunity to review and compare previous setups.

Macros.$MACH

is the Makefile Macros file for the current configuration. The Makefile is located in the Tools directory and is identical on all machines. The Macros file is a machine and compiler dependent file. This file is locked during the build step.

README.case

provides a summary of the commands used to generate this case.

SourceMods

contains directories for each component where case specific source code modifications can be included. The source files in these directories will always be used in preference to the source code in CCSMROOT. This feature allows users to modify CCSM source code on a case by case basis if that preferable over making modifications in the CCSMROOT sandbox.

$CASE.$MACH.build

is the script that is run interactively to build the CCSM model.

$CASE.$MACH.clean_build

is the script that cleans the CCSM build.

$CASE.$MACH.l_archive

is the script that is submitted to the batch queue to archive CCSM data to the long-term archive disk, like an hpss or mass storage system.

$CASE.$MACH.run

is the script that is submitted to the batch queue to run a CCSM job. This script could also be run interactively if resources allow.

check_input_data

is a tool that checks for missing input datasets and provides a capability for exporting them to local disk.

configure

is the script that is run to generate files in Buildconf and the build and run scripts for a case.

create_production_test

is a tool that generates an exact restart test in a separate directory based on the current case.

env_*.xml files

contain variables used to setup, configure, build, and run CCSM.

logs

is a directory that contains a copy of the component log files from successful case runs.

timing

is a directory that contains timing output from each successful case run.

xmlchange

is a script that supports changing xml variables in the env files.

$CASEROOT/Tools

contains many scripts that are used to setup and configure the CCSM model as well as run it. Some of particular note are

- Makefile is the Makefile that will be used for the build.
- Templates is a directory that contains all the component template files used during the configure phase to generate buildnml and buildexe scripts in the $CASEROOT/Buildconf directory.
- ccsm_buildexe.csh is invoked by $CASEROOT/$CASE.$MACH.build to generate the model executable. This script calls the component buildexe scripts in Buildconf.
- ccsm_buildnml.csh is invoked by $CASEROOT/$CASE.$MACH.build to generate the component namelists in $RUNROOT. This script calls the component buildnml scripts in Buildconf.
- ccsm_check_lockedfiles checks that any files in the $CASEROOT/LockedFiles/ directory match those in the $CASEROOT directory. This helps protect users from overwriting variables that should not be changed.
- ccsm_getenv converts the xml variables in $CASEROOT to csh environmental variables.
- ccsm_l_archive.csh is the script that does long-term archiving of model data as part of the $CASE.$MACH.l_archive batch job.
- ccsm_postrun.csh may run the short & long-term archivers, resubmit the run script, and run the timing script.
- ccsm_prestage.csh checks that required input datasets are available.
- generate_batch.csh is the script that generates resolved run and long-term archiving batch scripts by configure.
- generate_resolved.csh generates resolved buildnml and buildexe scripts in the $CASEROOT/Buildconf directory for model components.
- getTiming.csh generates the timing information.
- getTiming.pl generates timing information and is used by getTiming.csh.
- mkDepends generates Makefile dependencies in a form suitable for inclusion into a Makefile.

- perf_summary.pl generates timing information.
- st_archive.sh is the short-term archive script. It moves model output out of run directory to the short-term archive directory. Associated with DOUT_S and DOUT_S_ROOT env variables in env_run.xml.
- taskmaker.pl derives pe counts and task and thread geometry info based on env var values set in the env_mach_pes file.
- xml2env converts env_*xml files to shell environment variable files that are then sourced for inclusion in the model environment. Used by the ccsm_getenv script.

How do I modify the value of CCSM env variables?

CCSM recommends using the xmlchange tool to modify env variables. xmlchange supports error checking as part of the implementation. Also, using xmlchange will decrease the chance that typographical errors will creep into the xml files. Conversion of the xml files to environment variables can fail silently with certain xml format errors. To use xmlchange, do, for instance,

```
> cd $CASEROOT
> ./xmlchange -file env_run.xml -id STOP_OPTION -val nmonths
> ./xmlchange -file env_run.xml -id STOP_N -val 6
```

which will change the variables STOP_OPTION and STOP_N in the file env_run.xml to the specified values. The xml files can be edited manually, but users should take care not to introduce any formatting errors that could lead to incomplete env settings. See also .

Why aren't my env variable changes working?

It's possible that a formatting error has been introduced in the env xml files. If there appear to be problems with the env variables (i.e. if the model doesn't seem to have consistent values compared to what's set in the xml files), then confirm that the env variables are being set properly. There are a couple of ways to do that. First, run the ccsm_getenv script via

```
> cd $CASEROOT
> source ./Tools/ccsm_getenv
> env
```

and review the output generated by the command "env". The env variables should match the xml settings. Another option is to edit the $CASEROOT/Tools/ccsm_getenv script and comment out the line "rm $i:r". That should leave the shell env files around, and they can then be reviewed. The latter approach should be undone as soon as possible to avoid problems running ccsm_getenv later.

What's the deal with the CCSM4 env variables and env xml files?

CCSM4 cases are configured largely through setting what CCSM4 calls "environment variables". These actually appear to the user as variables defined in xml files. Those files appear in the case directory once a case is created and are named something like env_*.xml. They are converted to actual environment variables via a csh script called ccsm_getenv. That script calls a perl script called xml2env that converts the xml files to shell files that are then sourced and removed. The ccsm_getenv and xml2env exist in the $CASEROOT/Tools directory. The environment variables are specified in xml files to support extra automated error checking and automatic generation of env variable documentation. If you want to have the ccsm environment variables in your local shell environment, do the following

```
> cd $CASEROOT
> source ./Tools/ccsm_getenv
```

You must run the ccsm_getenv from the CASEROOT directory exactly as shown above. There are multiple env_*.xml files including env_case.xml, env_conf.xml, env_mach_pes.xml, env_build.xml, and env_run.xml. To a large degree, the different env files exist so variables can be locked in different phases of the case setup, build, and run process. For more info on locking files, see the Section called *Why is there file locking and how does it work?*. The important point is that env_case.xml variables cannot be changed after create_newcase is invoked. env_conf and env_mach_pes cannot be changed after configure is invoked unless you plan to reconfigure the case. env_build variables cannot be changed after the model is built unless you plan to clean and rebuild. env_run variables can be changed anytime. The CCSM4 scripting software checks that xml files are not changed when they shouldn't be.

CCSM recommends using the xmlchange tool to modify env variables. This will decrease the chance that typographical errors will creep into the xml files. Conversion of the xml files to environment variables can fail silently with certain xml format errors. To use xmlchange, do, for instance,

```
> cd $CASEROOT
> ./xmlchange -file env_run.xml -id STOP_OPTION -val nmonths
> ./xmlchange -file env_run.xml -id STOP_N -val 6
```

which will change the variables STOP_OPTION and STOP_N in the file env_run.xml to the specified values. The xml files can be edited manually, but users should take care not to introduce any formatting errors that could lead to incomplete env settings. If there appear to be problems with the env variables (i.e. if the model doesn't seem to have consistent values compared to what's set in the xml files), then confirm that the env variables are being set properly. There are a couple of ways to do that. First, run the ccsm_getenv script as indicated above and review the output generated by the command "env". The env variables should match the xml settings. Another option is to edit the $CASEROOT/Tools/ccsm_getenv script and comment out the line "rm $i:r". That should leave the shell env files around, and they can then be reviewed. The latter approach should be undone as soon as possible to avoid problems running ccsm_getenv later.

Why is there file locking and how does it work?

In CCSM4, there are several different env xml files. These include env_case.xml, env_conf.xml, env_mach_pes.xml, env_build.xml, and env_run.xml. These are organized so variables can be locked during different phases of the case configuration, build, and run. Locking variables is a feature of CCSM that prevents users from changing variables after they have been resolved (used) in other parts of the scripts system. The variables in env_case are locked when create_newcase is called. The env_conf and env_mach_pes variables are locked when configure is called. The env_build variables are locked when CCSM is built, and the env_run variables are never locked and can be changed anytime. In addition, the Macros file is locked as part of the build step. The $CASEROOT/LockedFiles directory saves copies of the xml files to facilitate the locking feature. In summary

- `env_case.xml` is locked upon invoking **create_newcase** and cannot be unlocked. To change settings in env_case, a new case has to be generated with create_newcase.
- `env_conf.xml` and `env_mach_pes.xml` are locked after running configure -case. After changing variable values in these files, reconfigure the model using "configure -cleanall" (or some variation) and then "configure -case".

- `Macros.$MACH` and `env_build.xml` are locked upon the *successful* completion of **$CASE.MACH.build**. Both `Macros.$MACH` and `env_build.xml` can be unlocked by invoking **$CASE.$MACH.cleanbuild** and then the model should be rebuilt.

How do I change processor counts and component layouts on processors?

See the Section called *Setting the case PE layout* in Chapter 3 or the use case the Section called *Changing PE layout* in Chapter 9.

What is pio?

The parallel IO (PIO) library is included with CCSM4 and is automatically built as part of the CCSM build. Several CCSM4 components use the PIO library to read and/or write data. The PIO library is a set of interfaces that support serial netcdf, parallel netcdf, or binary IO transparently. The implementation allows users to easily configure the pio setup on the fly to change the method (serial netcdf, parallel netcdf, or binary data) as well as various parameters associated with PIO to optimize IO performance.

CCSM4 prefers that data be written in CF compliant netcdf format to a single file that is independent of all parallel decomposition information. Historically, data was written by gathering global arrays on a root processor and then writing the data from the root processor to an external file using serial netcdf. The reverse process (read and scatter) was done for reading data. This method is relatively robust but is not memory scalable, performance scalable, or performance flexible.

PIO works as follows. The PIO library is initialized and information is provided about the method (serial netcdf, parallel netcdf, or binary data), and the number of desired IO processors and their layout. The IO parameters define the set of processors that are involved in the IO. This can be as few as one and as many as all processors. The data, data name and data decomposition are also provided to PIO. Data is written through the PIO interface in the model specific decomposition. Inside PIO, the data is rearranged into a "stride 1" decomposition on the IO processors and the data is then written serially using netcdf or in parallel using pnetcdf.

There are several benefits associated with using PIO. First, even with serial netcdf, the memory use can be significantly decreased because the global arrays are decomposed across the IO processors and written in chunks serially. This is critical as CCSM4 runs at higher resolutions where global arrays need to be minimized due to memory availability. Second, pnetcdf can be turned on transparently potentially improving the IO performance. Third, PIO parameters such as the number of IO tasks and their layout can be tuned to reduce memory and optimize performance on a machine by machine basis. Fourth, the standard global gather and write or read and global scatter can be recovered by setting the number of io tasks to 1 and using serial netcdf.

CCSM4 uses the serial netcdf implementation of PIO and pnetcdf is turned off in PIO by default. Several components provide namelist inputs that allow use of pnetcdf in PIO. To use pnetcdf, a pnetcdf library (like netcdf) must be available on the local machine and PIO pnetcdf support must be turned on when PIO is built. This is done as follows

1. Locate the local copy of pnetcdf. It must be version 1.1.1 or library
2. Set LIB_PNETCDF in the Macros file to the directory of the pnetcdf library (ie. /contrib/pnetcdf1.1.1/lib).
3. Add PNETCDF_PIO to the pio CONFIG_ARGS variable in the Macros file, and set it to the directory of the top level of a standard pnetcdf installation (ie /contrib/pnetcdf1.1.1).

4. Run the clean_build script if the model has already been built.
5. Run the build script to rebuilt pio and the full CCSM4 system.
6. Change component IO namelist settings to pnetcdf and set appropriate IO tasks and layout.

The PNETCDF_PIO variable tells pio to build with pnetcdf support turned on. The LIB_PNETCDF variable tells the CCSM Makefile to link in the pnetcdf library at the link step of the CCSM4 build.

There is an ongoing effort between CCSM, pio developers, pnetcdf developers and hardware vendors to understand and improve the IO performance in the various library layers. To learn more about pio, see http://code.google.com/p/parallelio.[1]

How do I use pnetcdf?

See the Section called *What is pio?*

How do I create my own compset?

Several compsets are hardwired in the CCSM4 release. "create_newcase -l" provides a current listing of supported "out-of-the-box" compsets.

To create a customized compset,

```
> cd $CCSMROOT/scripts
```

Now copy sample_compset_file.xml to another file, e.g. my_compset.xml.

```
> cp sample_compset_file.xml my_compset.xml
```

Edit the file, my_compset.xml, to create your own compset configuration. In particular, the NAME, SHORTNAME, DESC, and COMP_ variables should be specified. The STATUS and CCSM_CCOST variables can be ignored. Note: Other CCSM env variables can also be added here. See scripts/ccsm_utils/Case.template/config_compsets.xml for other variables that might be related to compset configuration.

Next run create_newcase with the optional -compset_file argument.

```
> create_newcase -case mycase -res f19_g16 -compset MYCS -mach mymach -compset_file my_
```

The case `mycase` should have been generated and the configuration should be consistent with settings from the my_compset.xml file.

How are cice and pop decompositions set and how do I override them?

The pop and cice models both have similar decompositions and strategies for specifying the decomposition. Both models support decomposition of the horizontal grid into two-dimensional blocks, and these blocks are then allocated to individual processors inside each component. The decomposition must be specified when the models are built. There are four environment variables in env_build.xml for each model that specify the decomposition used. These variables are POP or CICE followed by _BLCKX, _BLCKY, _MXBLCKS, and _DECOMP. BLCKX and BLCKY specify the size of the local block in grid cells in the "x" and "y" direction. MXBLCKS specifies the maximum number of blocks that might be on any given processor, and DECOMP specifies the strategy for laying out the blocks on processors.

The values for these environment variables are set automatically by scripts in the cice and pop "bld" directories when "configure -case" is run. The scripts that generate the decompositions are

```
models/ocn/pop2/bld/generate_pop_decomp.pl
models/ice/cice/bld/generate_cice_decomp.pl
```

Those tools leverage decompositions stored in xml files,

```
models/ocn/pop2/bld/pop_decomp.xml
models/ice/cice/bld/cice_decomp.xml
```

to set the decomposition for a given resolution and total processor count. The decomposition used can have a significant effect on the model performance, and the decompositions specified by the tools above generally provide optimum or near optimum values for the given resolution and processor count. More information about cice and pop decompositions can be found in each of those user guides.

The decompositions can be specified manually by setting the environment variable POP_AUTO_DECOMP or CICE_AUTO_DECOMP to false in env_mach_pes.xml (which turns off use of the scripts above) and then setting the four BLCKX, BLCKY, MXBLCKS, and DECOMP environment variables in env_build.xml.

In general, relatively square and evenly divided Cartesian decompositions work well for pop at low to moderate resolution. Cice performs best with "tall and narrow" blocks because of the load imbalance for most global grids between the low and high latitudes. At high resolutions, more than one block per processor can result in land block elimination and non-Cartesian decompositions sometimes perform better. Testing of several decompositions is always recommended for performance and validation before a long run is started.

How do I change history file output frequency and content for CAM and CLM during a run?

If you want to change the frequency of output for CAM or CLM (i.e. generate output every 6 model hours instead of once a model day) in the middle of a run, or if you want to change the fields that are output, in the middle of a run, you need to stop the run, rebuild and rerun it with the same casename and branch from the same casename. See the steps below for doing a branch run while retaining the casename.

Rebuilding the case and restarting it where you left off, are necessary because CAM and CLM only read namelist variables once, at the beginning of a run. This is not the case for POP and CICE, they read the namelist input on every restart, and therefore for POP and CICE, you can change output fields and frequency by modifying the appropriate namelist variables and then doing a restart.

The following example shows case B40.20th.1deg which runs from 1850 to 2005, and will generate high frequency output for years 1950 through 2005. CAM will output data every six hours instead of once a day. Also starting at year 1950 additional fields will be output by the model.

1. The first step is to create case b40.20th.1deg and run the case for years 1850 through 1949 with your initial settings for output.
2. Next move your entire case directory, $CASEDIR, somewhere else, because you need to rebuild and rerun the case using the same name.

   ```
   > cd $CASEDIR
   > mv b40.20th.1deg b40.20th.1deg.1850-1949
   ```

3. Now move your run directory, $RUNDIR, somewhere else as well.

   ```
   > cd $RUNDIR
   ```

```
> mv b40.20th.1deg b40.20th.1deg.1850-1949
```

4. Next create a new case in your case directory with the same name, b40.20th.1deg.

```
> cd $CASEDIR/scripts
> create_newcase -mach bluefire -compset B_1850-2000_CN -res f09_g16 -case /fs/cgc
cd $RUNDIR
```

5. Next edit the namelist file, env_conf.xml, in the run directory, $RUNDIR, as follows:

```
> cd $RUNDIR
> xmlchange -file env_conf.xml -id RUN_TYPE -val 'branch'
> xmlchange -file env_conf.xml -id RUN_REFCASE -val 'b40.20th.1deg'
> xmlchange -file env_conf.xml -id RUN_REFDATE -val '1948-01-01'
> xmlchange -file env_conf.xml -id CAM_NML_USE_CASE -val '1850-2005_cam4'
> xmlchange -file env_conf.xml -id BRNCH_RETAIN_CASENAME -val 'TRUE'
> xmlchange -file env_conf.xml -id GET_REFCASE -val 'FALSE'
```

6. Next configure the case and edit the coupler and CAM namelists.

   a. Configure case.

   ```
   > configure -case
   ```

   b. Edit Buildconf/cpl.buildnml.csh. Replace existing brnch_retain_casename line with the following line brnch_retain_casename = .true.

   Edit Buildconf/cam.buildnml.csh. Check that bndtvghg = '$DIN_LOC_ROOT' and add:

```
&cam_inparm
       doisccp = .true.
       isccpdata = '/fis/cgd/cseg/csm/inputdata/atm/cam/rad/isccp.tautab_invt
       mfilt   = 1,365,30,120,240
       nhtfrq  = 0,-24,-24,-6,-3
       fincl2  = 'TREFHTMN','TREFHTMX','TREFHT','PRECC','PRECL','PSL'
       fincl3  = 'CLDICE','CLDLIQ','CLDTOT','CLOUD','CMFMC','CMFMCDZM','FISC
                 'FLDS','FLDSC','FLNS','FLUT','FLUTC','FSDS','FSDSC','FSNS',
                 'FSNSC','FSNTOA','FSNTOAC','LHFLX','OMEGA','OMEGA500',
                 'PRECSC','PRECSL','PS','Q','QREFHT','RELHUM','RHREFHT','SHFl
                 'SOLIN','T','TGCLDIWP','TGCLDLWP','U','V','Z3'
       fincl4  = 'PS:I','PSL:I','Q:I','T:I','U:I','V:I','Z3:I'
       fincl5  = 'CLDTOT','FLDS','FLDSC','FLNS','FLNSC','FSDS','FSDSC','FSNS
                 'LHFLX','PRECC','PRECL','PRECSC','PRECSL','SHFLX',
                 'PS:I','QREFHT:I','TREFHT:I','TS:I'
                  /
```

7. Now build and run the case.

```
> b40.20th.1deg.$MACH.build
> bsub < b40.20th.1deg.$MACH.run
```

Notes

1. http://code.google.com/p/parallelio

Appendix A. Supported Component Sets

The following lists the supported component sets. Note that all the component sets currently use the stub GLC component, sglc. Run "create_newcase -list" from the scripts directory to view the list for the current version of CCSM4.

For an overview of CCSM components and component sets see overview of ccsm components.

Table A-1. Component Sets

Compset (Short name)	Details
A_PRESENT_DAY (A)	Components: datm,dlnd,dice,docn,sglc
	Description: All data model
B_2000 (B)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, present day
B_1850 (B1850)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, pre-industrial
B_1850_CN (B1850CN)	Components: cam,clm,cice,pop2,sglc
	Description: all active components, pre-industrial, with CN (Carbon Nitrogen) in CLM
B_1850_CN_CHEM (B1850CNCHM)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, pre-industrial, with CN (Carbon Nitrogen) in CLM and super_fast_llnl chem in atm
B_1850_RAMPCO2_CN (B1850RMCN)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, pre-industrial with co2 ramp, with CN (Carbon Nitrogen) in CLM
B_1850-2000 (B20TR)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, 1850 to 2000 transient
B_1850-2000_CN (B20TRCN)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, 1850 to 2000 transient, with CN (Carbon Nitrogen) in CLM
B_1850-2000_CN_CHEM (B20TRCNCHM)	Components: cam,clm,cice,pop2,sglc

Compset (Short name)	Details
	Description: All active components, 1850 to 2000 transient, with CN (Carbon Nitrogen) in CLM and super_fast_llnl chem in atm
B_2000_TROP_MOZART (BMOZ)	Components: cam,clm,cice,pop2,sglc
	Description: All active components, with trop_mozart
B_1850_WACCM (BW1850)	Components: cam,clm,cice,pop2,sglc
	Description: all active components, pre-industrial, with waccm
B_1850_WACCM_CN (BW1850CN)	Components: cam,clm,cice,pop2,sglc
	Description: all active components, pre-industrial, with waccm and CN
C_NORMAL_YEAR (C)	Components: datm,dlnd,dice,pop2,sglc
	Description: Active ocean model with COREv2 normal year forcing
D_NORMAL_YEAR (D)	Components: datm,slnd,cice,docn,sglc
	Description: Active ice model with COREv2 normal year forcing
E_2000 (E)	Components: cam,clm,cice,docn,sglc
	Description: Fully active cam and ice with som ocean, present day
E_1850_CN (E1850CN)	Components: cam,clm,cice,docn,sglc
	Description: Pre-industrial fully active ice and som ocean, with CN
F_AMIP (FAMIP)	Components: cam,clm,cice,docn,sglc
	Description: Default resolution independent AMIP is INVALID
F_1850 (F1850)	Components: cam,clm,cice,docn,sglc
	Description: Pre-industrial cam/clm with prescribed ice/ocn
F_2000 (F)	Components: cam,clm,cice,docn,sglc
	Description: Stand-alone cam default, prescribed ocn/ice
F_2000_CN (FCN)	Components: cam,clm,cice,docn,sglc
	Description: Stand-alone cam default, prescribed ocn/ice with CN

Compset (Short name)	Details
F_1850-2000_CN (F20TRCN)	Components: cam,clm,cice,docn,sglc
	Description: 20th Century transient stand-alone cam default, prescribed ocn/ice, with CN
F_WACCM_1850 (FW1850)	Components: cam,clm,cice,docn,sglc
	Description: Pre-industrial cam/clm with prescribed ice/ocn
G_NORMAL_YEAR (G)	Components: datm,dlnd,cice,pop2,sglc
	Description: Coupled ocean ice with COREv2 normal year forcing
H_PRESENT_DAY (H)	Components: datm,slnd,cice,pop2,sglc
	Description: Coupled ocean ice slnd
I_2000 (I)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 2003 and Satellite phenology, CO2 level and Aerosol deposition for 2000
I_1850 (I1850)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 1948 to 1972 and Satellite phenology, CO2 level and Aerosol deposition for 1850
I_1948-2004 (I4804)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 1948 to 2004 and Satellite phenology, CO2 level and Aerosol deposition for 2000
I_1850-2000 (I8520)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 1948 to 2004 and transient Satellite phenology, and Aerosol deposition from 1850 to 2000 and 2000 CO2 level
I_2000_CN (ICN)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 2003 and CN (Carbon Nitrogen) biogeochemistry, CO2 level and Aerosol deposition for 2000
I_1850_CN (I1850CN)	Components: datm,clm,sice,socn,sglc

Compset (Short name)	Details
	Description: Active land model with QIAN atm input data for 1948 to 1972 and CN (Carbon Nitrogen) biogeochemistry, CO2 level and Aerosol deposition for 1850
I_1948-2004_CN (I4804CN)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 1948 to 2004 and CN (Carbon Nitrogen) biogeochemistry, CO2 level and Aerosol deposition for 2000
I_1850-2000_CN (I8520CN)	Components: datm,clm,sice,socn,sglc
	Description: Active land model with QIAN atm input data for 1948 to 1972 and transient CN (Carbon Nitrogen) biogeochemistry, and Aerosol deposition from 1850 to 2000 and 2000 CO2 level
S_PRESENT_DAY (S)	Components: xatm,slnd,sice,socn,sglc
	Description: All stub models plus xatm
X_PRESENT_DAY (X)	Components: xatm,xlnd,xice,xocn,xglc
	Description: All dead model

Appendix B. Supported Grids

The following table lists all the supported grids. Run "create_newcase -list" from the scripts directory to view the list for the current version of CCSM4.

Table B-1. supported grids

Grid (Short-name)	atm_grid	lnd_grid	ice_grid	ocn_grid	atm_grid type	ocn_grid type
pt1_pt1 (pt1)	pt1	pt1	pt1	pt1	NA	NA
0.47x0.63_0.47x0.63 (f05_f05)	0.47x0.63	0.47x0.63	0.47x0.63	0.47x0.63	finite volume	finite volume
0.47x0.63_gx1v6 (f05_g16)	0.47x0.63	0.47x0.63	gx1v6	gx1v6	finite volume	displaced pole
0.47x0.63_tx0.1v2 (f05_t12)	0.47x0.63	0.47x0.63	tx0.1v2	tx0.1v2	finite volume	triple pole
0.9x1.25_0.9x1.25 (f09_f09)	0.9x1.25	0.9x1.25	0.9x1.25	0.9x1.25	finite volume	finite volume
0.9x1.25_gx1v6 (f09_g16)	0.9x1.25	0.9x1.25	gx1v6	gx1v6	finite volume	displaced pole
1.9x2.5_1.9x2.5 (f19_f19)	1.9x2.5	1.9x2.5	1.9x2.5	1.9x2.5	finite volume	finite volume
1.9x2.5_gx1v6 (f19_g16)	1.9x2.5	1.9x2.5	gx1v6	gx1v6	finite volume	displaced pole
4x5_4x5 (f45_f45)	4x5	4x5	4x5	4x5	finite volume	finite volume
4x5_gx3v7 (f45_g37)	4x5	4x5	gx3v7	gx3v7	finite volume	displaced pole
T62_gx3v7 (T62_g37)	96x192	96x192	gx3v7	gx3v7	spectral	displaced pole
T62_tx0.1v2 (T62_t12)	96x192	96x192	tx0.1v2	tx0.1v2	spectral	triple pole
T62_gx1v6 (T62_g16)	96x192	96x192	gx1v6	gx1v6	spectral	displaced pole
T31_T31 (T31_T31)	48x96	48x96	48x96	48x96	spectral	spectral
T31_gx3v7 (T31_g37)	48x96	48x96	gx3v7	gx3v7	spectral	displaced pole
T42_T42 (T42_T42)	64x128	64x128	64x128	64x128	spectral	spectral
10x15_10x15 (f10_f10)	10x15	10x15	10x15	10x15	finite volume	finite volume
ne30np4_1.9x2.5_gx1v6 (ne30_f19_g16)	ne30np4	1.9x2.5	gx1v6	gx1v6	cubed sphere	displaced pole

Appendix C. Supported Machines

The following table lists all supported and generic machines. Run "create_newcase -list" from the scripts directory to view the list for the current version of CCSM4.

Name	Description
bluefire	NCAR IBM p6, os is AIX, 32 pes/node, batch system is LSF
edinburgh_lahey	NCAR CGD Linux Cluster (lahey), 8 pes/node, batch system is PBS
edinburgh_pgi	NCAR CGD Linux Cluster (pgi), 8 pes/node, batch system is PBS
edinburgh_intel	NCAR CGD Linux Cluster (intel), 8 pes/node, batch system is PBS
franklin	NERSC XT4, os is CNL, 4 pes/node, batch system is PBS
hadley	UCB Linux Cluster, os is Linux (ia64), batch system is PBS
intrepid	ANL IBM BG/P, os is BGP, 4 pes/node, batch system is cobalt
jaguar	ORNL XT4, os is CNL, 4 pes/node, batch system is PBS
jaguarpf	ORNL XT5, os is CNL, 12 pes/node, batch system is PBS
kraken	NICS/UT/teragrid XT5, os is CNL, 12 pes/node
midnight	ARSC Sun Cluster, os is Linux (pgi), batch system is PBS
prototype_atlas	LLNL Linux Cluster, Linux (pgi), 8 pes/node, batch system is Moab
prototype_columbia	NASA Ames Linux Cluster, Linux (ia64), 2 pes/node, batch system is PBS
prototype_frost	NCAR IBM BG/L, os is BGL, 8 pes/node, batch system is cobalt
prototype_nyblue	SUNY IBM BG/L, os is BGL, 8 pes/node, batch system is cobalt
prototype_ranger	TACC Linux Cluster, Linux (pgi), 1 pes/node, batch system is SGE
prototype_schirra	NAS (NASA) IBM p5+, os is AIX, 2 pes/node, batch system is PBS
prototype_ubgl	LLNL IBM BG/L, os is BGL, 2 pes/node, batch system is Moab
generic_ibm	generic ibm power system, os is AIX, batch system is LoadLeveler, user-defined
generic_xt	generic CRAY XT, os is CNL, batch system is PBS, user-defined
generic_linux_pgi	generic linux (pgi), os is Linux, batch system is PBS, user-defined

Name	Description
generic_linux_lahey	generic linux (lahey), os is Linux, batch system is PBS, user-defined
generic_linux_intel	generic linux (intel), os is Linux, batch system is PBS, user-defined
generic_linux_pathscale	generic linux (pathscale), os is Linux, batch system is PBS, user-defined

Appendix D. env_case.xml variables

The following table lists all the environment variables set in the `env_case.xml` file. These variables cannot be modified.

Table D-1. env_case.xml variables

Name	Type	Default	Description [Valid Values]
ATM_GRID	char	UNSET	atmosphere grid
ATM_NX	integer	0	number of atmosphere cells in i direction
ATM_NY	integer	0	Number of atmosphere cells in j direction
BLDROOT	char	$CCSMROOT/scripts/ccsm_utils/Build	CCSM scripts build directory location
CASE	char	UNSET	case name
CASEBUILD	char	$CASEROOT/Buildconf	Buildconf directory location
CASEROOT	char	UNSET	full path of case
CASETOOLS	char	$CASEROOT/Tools	Case Tools directory location
CCSMROOT	char	UNSET	ccsm source root directory
CCSMUSER	char	UNSET	case user name
CCSM_CCOST	integer	0	2**n relative cost of compset B is 1 (DO NOT EDIT)
CCSM_COMPSET	char	UNSET	CCSM component set
CCSM_GCOST	integer	0	2**n relative cost of grid f19_g15 is 1 (DO NOT EDIT)
CCSM_LCOMPSET	char	UNSET	CCSM component set, longname
CCSM_MCOST	integer	0	2**n relative cost of machine (DO NOT EDIT)
CCSM_SCOMPSET	char	UNSET	CCSM component set, shortname
CODEROOT	char	$CCSMROOT/models	CCSM source models directory location
COMP_ATM	char	cam	Name of atmospheric component [cam,datm,xatm,satm]

Name	Type	Default	Description [Valid Values]
COMP_CPL	char	cpl	Name of coupling component [cpl]
COMP_GLC	char	sglc	Name of land ice component [xglc,sglc,gglc]
COMP_ICE	char	cice	Name of sea ice component [cice,dice,xice,sice]
COMP_LND	char	clm	Name of land component [clm,dlnd,xlnd,slnd]
COMP_OCN	char	pop2	Name of ocean component [pop2,docn,xocn,socn,camdom]
GLC_GRID	char	UNSET	glc grid (must equal lnd grid)
GLC_NX	integer	0	number of glc cells in i direction
GLC_NY	integer	0	number of glc cells in j direction
GRID	char	UNSET	CCSM grid
ICE_GRID	char	UNSET	ice grid (must equal ocn grid)
ICE_NX	integer	0	number of ice cells in i direction
ICE_NY	integer	0	number of ice cells in j direction
LND_GRID	char	UNSET	land grid
LND_NX	integer	0	Number of land cells in i direction
LND_NY	integer	0	number of land cells in j direction
MACH	char	UNSET	current machine name
OCN_GRID	char	UNSET	ocn grid
OCN_NX	integer	0	number of ocn cells in i direction
OCN_NY	integer	0	number of ocn cells in j direction
OS	char	UNSET	operating system
PTS_MODE	logical	FALSE	Points mode logical for single point [TRUE,FALSE]

Name	Type	Default	Description [Valid Values]
SCRIPTSROOT	char	$CCSMROOT/scripts	CCSM source scripts directory location
SHAREROOT	char	$CCSMROOT/models/csm_share	CCSM source models share code location
UTILROOT	char	$CCSMROOT/scripts/ccsm_utils	CCSM source scripts utils directory location
XMLMODE	char	normal	xml format option, expert removes extra comments from env xml files [normal,expert]

Appendix E. env_conf.xml variables

The following table lists all the possible environment variables that can be set in the env_conf.xml file. Because some of these variables are dependent on the components selected, only a subset of these will appear in $CASEROOT. Default values, where appropriate, are given in parentheses.

Table E-1. env_conf.xml variables

Name	Type	Default	Description [Valid Values]
BRNCH_RETAIN_CASENAME	logical	FALSE	allow same branch casename as reference casename [TRUE,FALSE]
CAM_CONFIG_OPTS	char		CAM configure options, see CAM configure utility for details
CAM_DYCORE	char	fv	CAM dynamical core [eul,fv,homme]
CAM_NAMELIST_OPTS	char		CAM specific namelist settings for -namelist option
CAM_NML_USE_CASE	char	UNSET	CAM namelist use_case
CCSM_BGC	char	none	BGC flag [none,CO2A,CO2B,CO2C,CO2_DMSA]
CCSM_CO2_PPMV	real	379.000	CO2 ppmv
CICE_CONFIG_OPTS	char	-ntr_aero 3 -ntr_pond 1 -ntr_iage 1 -ntr_FY 1	CICE configure options, see CICE configure utility for details
CICE_MODE	char	prognostic	cice mode [prognos-tic,prescribed,thermo_only]
CICE_NAMELIST_OPTS	char		CICE specific namelist settings for -namelist option
CICE_PRESAERO_TYPE	char	data_clim_2000	cice prescribed aerosol type, only used if not obtained from atmosphere [data_clim_2000,data_clim_1850,data_clim

Name	Type	Default	Description [Valid Values]
CLM_CO2_TYPE	char	constant	clm CO2 type, constant means value in CLM namelist, diagnostic or prognostic mean use the value sent from the atmosphere model [constant,diagnostic,prognostic]
CLM_CONFIG_OPTS	char		Options to send to CLM configure (see models/lnd/clm/bld/configure -h for list of options)
CLM_FORCE_COLDSTART	char	off	Value of 'on' forces model to spinup from a cold-start (arbitrary initial conditions) [on,off]
CLM_NAMELIST_OPTS	char		Namelist settings to add to the clm_inparm namelist Note, use ' around character values, as XML parser can't handle quotes inside strings. (list of item=value settings, see models/lnd/clm/bld/namelist_files/namelist
CLM_NML_USE_CASE	char	UNSET	CLM namelist use_case (for a list see models/lnd/clm/bld/build-namelist -use_case list)
CLM_PT1_NAME	char	UNSET	Name of single point resolution dataset to be used in I compset only (for a list see models/lnd/clm/bld/build-namelist -res list)

Name	Type	Default	Description [Valid Values]
CLM_USRDAT_NAME	char	UNSET	Data identifier name for CLM user-created datasets (see Quick-start.userdatasets)
CPL_ALBAV	logical	false	Only used for C,G compsets: if true, compute albedos to work with daily avg SW down [true,false]
CPL_EPBAL	char	off	Only used for C,G compsets: if ocn, ocn provides EP balance factor for precip [off,ocn]
DATM_CLMNCEP_YR_ALIGN	integer	1	year align (simulation year corresponding to starting year) for CLM_QIAN mode
DATM_CLMNCEP_YR_END	integer	2004	ending year to loop data over for CLM_QIAN mode
DATM_CLMNCEP_YR_START	integer	2002	starting year to loop data over for CLM_QIAN mode
DATM_MODE	char	CORE2_NYF	DATM mode [CORE2_NYF,CLM_QIAN,CLM1PT]
DICE_MODE	char	ssmi	DICE mode [ssmi]
DLND_MODE	char	NULL	DLND mode [CPLHIST,NULL]
DLND_RUNOFF_MODE	char	RX1	DLND runoff mode [CPL-HIST,RX1,NULL]
DOCN_MODE	char	prescribed	DOCN mode [prescribed,som]
DOCN_SSTDATA_FILENAME	char	UNSET	Sets sst/ice_cov filename for amip runs, only used in F compset
DOCN_SSTDATA_YEAR_END	integer	-999	Sets year end of sst/ice_cov for amip runs, only used in F compset

Name	Type	Default	Description [Valid Values]
DOCN_SSTDATA_YEAR_START	integer	-999	Sets year start of sst/ice_cov for amip runs, only used in F compset
GET_REFCASE	logical	FALSE	flag for automatically prestaging the refcase restart dataset [TRUE,FALSE]
MAP_A2LF_FILE	char	UNSET	atm to land mapping file for fluxes
MAP_A2LS_FILE	char	UNSET	atm to land mapping file for states
MAP_A2OF_FILE	char	UNSET	atm to ocn flux mapping file for fluxes (currently first order conservative)
MAP_A2OS_FILE	char	UNSET	atm to ocn state mapping file for states (currently bilinear)
MAP_L2AF_FILE	char	UNSET	land to atm mapping file for fluxes
MAP_L2AS_FILE	char	UNSET	land to atm mapping file for states
MAP_O2AF_FILE	char	UNSET	ocn to atm mapping file for fluxes (currently first order conservative)
MAP_O2AS_FILE	char	UNSET	ocn to atm mapping file for states
MAP_R2O_FILE_R05	char	UNSET	runoff (.5 degree) to ocn mapping file
MAP_R2O_FILE_R19	char	UNSET	runoff (19 basin) to ocn mapping file
MAP_R2O_FILE_RX1	char	UNSET	runoff (1 degree) to ocn mapping file

Name	Type	Default	Description [Valid Values]
MPISERIAL_SUPPORT	logical	FALSE	TRUE implies this machine supports the use of the mpiserial lib. Not all machines support the use of the mpiserial lib [TRUE,FALSE]
OCN_CHL_TYPE	char	diagnostic	provenance of surface Chl for radiative penetration computations [diagnos-tic,prognostic]
OCN_CO2_TYPE	char	constant	provenance of atmospheric CO2 for gas flux computation [con-stant,prognostic]
OCN_COUPLING	char	full	surface heat and freshwater forcing, partial is consistent with coupling to a data atm model [full,partial]
OCN_ICE_FORCING	char	active	under ice forcing, inactive is consistent with coupling to a data ice model [active,inactive]
OCN_TRANSIENT	char	unset	specification of transient forcing datasets [unset,1850-2000]
RUN_REFCASE	char	case.std	Reference case for hybrid or branch runs
RUN_REFDATE	char	0001-01-01	Reference date for hybrid or branch runs (yyyy-mm-dd). Used to determine the component dataset that the model starts from. Ignored for startup runs

Name	Type	Default	Description [Valid Values]
RUN_STARTDATE	char	0001-01-01	Run start date (yyyy-mm-dd). Only used for startup or hybrid runs Ignored for branch runs.
RUN_TYPE	char	startup	Run initialization type [startup,hybrid,branch]
USE_MPISERIAL	logical	FALSE	TRUE implies code is built using the mpiserial library. If TRUE, the MPISERIAL_SUPPORT must also be TRUE. FALSE (default) implies that code is built with a real MPI library. If a job uses only one MPI task (e.g. single-column CAM and CLM), the mpiserial lib may be an alternative to real mpi lib [TRUE,FALSE]

Appendix F. env_mach_pes.xml variables

The following table lists all the environment variables set in the `env_mach_pes.xml` file. Default values, where appropriate, are given in parentheses.

Table F-1. env_mach_pes.xml variables

Name	Type	Default	Description [Valid Values]
CCSM_ESTCOST	integer	0	2**n relative cost of case (DO NOT EDIT)
CCSM_PCOST	integer	0	cost relative to 64 pes (DO NOT EDIT)
CCSM_TCOST	integer	0	2**n relative cost of test where ERS is 1 (DO NOT EDIT)
CICE_AUTO_DECOMP	logical	true	if false, user must set the CICE decomp, otherwise configure sets it [true,false]
MAX_TASKS_PER_NODE	integer	UNSET	maximum number of mpi tasks per node
NTASKS_ATM	char	0	number of atmosphere tasks
NTASKS_CPL	char	0	number of coupler mpi tasks
NTASKS_GLC	char	0	number of glc mpi tasks
NTASKS_ICE	char	0	number of ice mpi tasks
NTASKS_LND	char	0	number of land mpi tasks
NTASKS_OCN	char	0	number of ocean mpi tasks
NTHRDS_ATM	char	0	number of atmosphere threads
NTHRDS_CPL	char	0	number of coupler mpi threads
NTHRDS_GLC	char	0	number of glc mpi threads
NTHRDS_ICE	char	0	number of ice mpi threads
NTHRDS_LND	char	0	number of land mpi threads
NTHRDS_OCN	char	0	number of ocean mpi threads

Name	Type	Default	Description [Valid Values]
PES_LEVEL	char	UNSET	pes level determined by automated initialization (DO NOT EDIT)
PES_PER_NODE	char	$MAX_TASKS_PER_NODE	pes per node for cost scaling
POP_AUTO_DECOMP	logical	true	if false, user must set the POP decomp, otherwise configure sets it [true,false]
PSTRID_ATM	integer	1	stride of mpi tasks for atm comp - currently should always be set to 1 [1]
PSTRID_CPL	integer	1	stride of mpi tasks for cpl comp - currently should always be set to 1 [1]
PSTRID_GLC	integer	1	stride of mpi tasks for glc comp - currently should always be set to 1 [1]
PSTRID_ICE	integer	1	stride of mpi tasks for ice comp - currently should always be set to 1 [1]
PSTRID_LND	integer	1	stride of mpi tasks for lnd comp - currently should always be set to 1 [1]
PSTRID_OCN	integer	1	stride of mpi tasks for ocn comp - currently should always be set to 1 [1]
ROOTPE_ATM	char	0	root atm mpi task
ROOTPE_CPL	char	0	root cpl mpi task
ROOTPE_GLC	char	0	root glc mpi task
ROOTPE_ICE	char	0	root ice mpi task
ROOTPE_LND	char	0	root lnd mpi task
ROOTPE_OCN	char	0	root ocn mpi task

Name	Type	Default	Description [Valid Values]
TOTALPES	integer	0	total number of pes (derived automatically - DO NOT EDIT)

Appendix G. env_build.xml variables

The following table lists all the environment variables set in the `env_build.xml` file. Default values, where appropriate, are given in parentheses.

Table G-1. env_build.xml variables

Name	Type	Default	Description [Valid Values]
BUILD_COMPLETE	logical	FALSE	If TRUE, models have been built successfully. (DO NOT EDIT) [TRUE,FALSE]
BUILD_STATUS	integer	0	Status of prior build. (DO NOT EDIT)
BUILD_THREADED	logical	FALSE	TRUE implies always build model for openmp capability If FALSE, build model with openmp capability only if THREAD is greater than 1 [TRUE,FALSE]
CICE_BLCKX	integer	0	Size of cice block in first horiz dimension. Value set by generate_cice_decomp.pl (called by configure). Only expert users should edit this
CICE_BLCKY	integer	0	Size of cice block in second horiz dimension. Value set by generate_cice_decomp.pl (called by configure). Only expert users should edit this
CICE_DECOMPTYPE	char	0	cice block distribution type. Value set by generate_cice_decomp.pl (called by configure). Only expert users should edit this

Name	Type	Default	Description [Valid Values]
CICE_MXBLCKS	integer	0	Max number of cice blocks per processor. Value set by generate_cice_decomp.pl (called by configure). Only expert users should edit this
COMP_INTERFACE	char	MCT	use MCT or ESMF component interfaces [MCT,ESMF]
DEBUG	logical	FALSE	TRUE implies turning on run and compile time debugging [TRUE,FALSE]
ESMF_LIBDIR	char		directory of esmf.mk in pre-built ESMF library
EXEROOT	char	UNSET	executable root directory
GLC_NEC	integer	0	GLC land ice model number of elevation classes [0,3,5,10]
GMAKE_J	integer	UNSET	Number of processors for gmake
HIRES	logical	false	true implies compile with HIRES CPP options. Primarily used by POP to determine physics changes for the eddy-resolving ocean resolution. This is a value that is set by the grid resolution. The user should not edit this [true,false]
INCROOT	char	$EXEROOT/lib/include	case lib include directory
LIBROOT	char	$EXEROOT/lib	case lib directory
OBJROOT	char	$EXEROOT	case build directory

Name	Type	Default	Description [Valid Values]
OCN_TRACER_MODULES	char	iage	Optional ocean tracers. Any combination of: iage cfc ecosys
PIO_CONFIG_OPTS	char	--disable-mct --disable-timing	PIO configure options, see PIO configure utility for details
POP_BLCKX	integer	0	Size of pop block in first horiz dimension. Value determined by gener-ate_pop_decomp.pl which is called by configure. Only expert users should edit this
POP_BLCKY	integer	0	Size of pop block in second horiz dimension. Value determined by gener-ate_pop_decomp.pl which is called by configure. Only expert users should edit this
POP_DECOMPTYPE	char	0	pop block distribution type. Value determined by gener-ate_pop_decomp.pl which is called by configure
POP_MXBLCKS	integer	0	Max number of pop blocks per processor. Value determined by gener-ate_pop_decomp.pl which is called by configure. Only expert users should edit this
RUNDIR	char	$EXEROOT/run	case run directory
SMP_BUILD	char	0	smp status of previous build, coded string. (DO NOT EDIT)

Name	Type	Default	Description [Valid Values]
SMP_VALUE	char	0	smp status of current setup, coded string (DO NOT EDIT)
USE_ESMF_LIB	char	FALSE	TRUE implies using the ESMF library specified by ESMF_LIBDIR or ESMFMKFILE [TRUE,FALSE]

Appendix H. env_run.xml variables

The following table lists all the xml variables set in the `env_run.xml` file.

Table H-1. env_run.xml variables

Name	Type	Default	Description [Valid Values]
AOFLUX_GRID	char	ocn	grid for atm ocn flux calc (atm currently not supported)
ATM_NCPL	char	24	number of atm coupling intervals per day
AVGHIST_DATE	integer	-999	yyyymmdd format, sets coupler time-average history date (like REST_DATE)
AVGHIST_N	char	-999	sets coupler time-average history file frequency (like REST_N)
AVGHIST_OPTION	char	never	sets coupler time-average history file frequency (like REST_OPTION)
BATCHQUERY	char	UNSET	command used to query batch system
BATCHSUBMIT	char	UNSET	command used to submit to batch system
BFBFLAG	logical	FALSE	turns on bit-for-bit reproducibility with varying pe counts in the driver and coupler, performance will likely be reduced [TRUE,FALSE]
BUDGETS	logical	FALSE	logical that turns on diagnostic budgets FALSE means budgets will never be written [TRUE,FALSE]

Name	Type	Default	Description [Valid Values]
BUDGET_ANNUAL	integer	1	output level for annual average budget diagnostics, written only if BUDGETS variable is TRUE, 0=none, 1=net summary, 2=+detailed surface, 3=+detailed atm [0,1,2,3]
BUDGET_DAILY	integer	0	output level for daily average budget diagnostics, written only if BUDGETS variable is TRUE, 0=none, 1=net summary, 2=+detailed surface, 3=+detailed atm [0,1,2,3]
BUDGET_INST	integer	0	output level for instantaneous budget diagnostics, written only if BUDGETS variable is TRUE, 0=none, 1=net summary, 2=+detailed surface, 3=+detailed atm [0,1,2,3]
BUDGET_LONGTERM_EOY	integer	1	output level for longterm average budget diagnostics written at end of year, written only if BUDGETS variable is TRUE, 0=none, 1=net summary, 2=+detailed surface, 3=+detailed atm [0,1,2,3]

Name	Type	Default	Description [Valid Values]
BUDGET_LONGTERM_STOP	integer	0	output level for longterm average budget diagnostics written at end of run, written only if BUDGETS variable is TRUE, 0=none, 1=net summary, 2=+detailed surface, 3=+detailed atm [0,1,2,3]
BUDGET_MONTHLY	integer	1	output level for monthly average budget diagnostics, written only if BUDGETS variable is TRUE, 0=none, 1=net summary, 2=+detailed surface, 3=+detailed atm [0,1,2,3]
CALENDAR	char	NO_LEAP	calendar type [NO_LEAP,GREGORIAN]
CASESTR	char	UNSET	case description
CCSM_BASELINE	char	/UNSET	standard ccsm baselines directory for testing
CCSM_CPRNC	char	/UNSET	standard location of the cprnc tool
CCSM_REPOTAG	char	UNSET	CCSM tag
CHECK_TIMING	logical	TRUE	logical to diagnose model timing at the end of the run [TRUE,FALSE]
CONTINUE_RUN	logical	FALSE	A continue run extends an existing CCSM run exactly. A setting of TRUE implies a continuation run [TRUE,FALSE]
DIN_LOC_ROOT	char	$DIN_LOC_ROOT_CSMDATA	local inputdata directory for CCSM prestaged data
DIN_LOC_ROOT_CLMQIAN	char	UNSET	general ccsm inputdata directory for CLM QIAN datm forcing files

Name	Type	Default	Description [Valid Values]
DIN_LOC_ROOT_CSMDATA	char	UNSET	general ccsm inputdata directory
DOUT_L_HTAR	logical	FALSE	logical to tar up long term archiver history files [TRUE,FALSE]
DOUT_L_MS	logical	FALSE	logical to turn on long term archiving (if DOUT_S is also TRUE) [TRUE,FALSE]
DOUT_L_MSROOT	char	UNSET	local long term archiving root directory
DOUT_S	logical	TRUE	logical to turn on short term archiving [TRUE,FALSE]
DOUT_S_ROOT	char	UNSET	local short term archiving root directory
DOUT_S_SAVE_INTERIM_RESTART_FILES	logical	FALSE	logical to archive all interim restart files, not just those at end of run [TRUE,FALSE]
DRV_THREADING	logical	FALSE	Turns on component varying thread control in the driver [TRUE,FALSE]
EPS_AAREA	real	9.0e-07	error tolerance for differences in atm/land areas in domain checking
EPS_AGRID	real	1.0e-12	error tolerance for differences in atm/land lon/lat in domain checking
EPS_AMASK	real	1.0e-13	error tolerance for differences in atm/land masks in domain checking
EPS_FRAC	real	1.0e-02	error tolerance for differences in fractions in domain checking

Name	Type	Default	Description [Valid Values]
EPS_OAREA	real	1.0e-01	error tolerance for differences in ocean/ice areas in domain checking
EPS_OGRID	real	1.0e-02	error tolerance for differences in ocean/ice lon/lat in domain checking
EPS_OMASK	real	1.0e-06	error tolerance for differences in ocean/ice masks in domain checking
GLC_NCPL	integer	1	number of glc coupling intervals per day (integer)
HISTINIT	logical	FALSE	logical to write an extra initial coupler history file [TRUE,FALSE]
HIST_DATE	integer	-999	yyyymmdd format, sets coupler snapshot history date (like REST_DATE)
HIST_N	char	-999	sets coupler snapshot history file frequency (like REST_N)
HIST_OPTION	char	never	sets coupler snapshot history file frequency (like REST_OPTION)
ICE_NCPL	char	$ATM_NCPL	number of ice coupling intervals per day (integer)
INFO_DBUG	integer	1	level of debug output, 0=minimum, 1=normal, 2=more, 3=too much [0,1,2,3]
LND_NCPL	char	$ATM_NCPL	number of land coupling intervals per day (integer)
LOGDIR	char	$CASEROOT/logs	directory where log files should be copied in addition to archiving

Name	Type	Default	Description [Valid Values]
NPFIX	logical	TRUE	invoke pole vector corrections in map_atmocn
OCN_NCPL	char	1	number of ocn coupling intervals per day (integer)
OCN_TIGHT_COUPLING	logical	FALSE	if TRUE, treats ocean model like lnd/ice in coupling and removes 1 coupling period lag at the cost of concurrency [TRUE,FALSE]
PTS_LAT	real(1)	-999.99	Points mode nearest latitudes
PTS_LON	real(1)	-999.99	Points mode nearest longitudes
REST_DATE	char	$STOP_DATE	date in yyyymmdd format, sets model restart write date in conjuction with REST_OPTION and REST_N
REST_N	char	$STOP_N	sets model restart writes in conjuction with REST_OPTION and REST_DATE (same options as STOP_N)
REST_OPTION	char	$STOP_OPTION	sets frequency of model restart writes (same options as STOP_OPTION) in conjuction with REST_N and REST_DATE
RESUBMIT	integer	0	if RESUBMIT is greater than 0, then case will automatically resubmit if that feature is supported in the run script
SHR_MAP_DOPOLE	logical	TRUE	invoke pole averaging corrections in shr_map_mod weights generation

Name	Type	Default	Description [Valid Values]
STOP_DATE	integer	-999	date in yyyymmdd format, sets the run length in conjuction with STOP_OPTION and STOP_N, can be in addition to STOP_OPTION and STOP_N, negative value implies off
STOP_N	integer	5	sets the run length in conjuction with STOP_OPTION and STOP_DATE

Name	Type	Default	Description [Valid Values]
STOP_OPTION	char	ndays	sets the run length in conjuction with STOP_N and STOP_DATE STOP_OPTION alarms are: [none/never], turns option off [nstep/s] , stops every STOP_N nsteps , relative to current run start time [nsecond/s] , stops every STOP_N nseconds, relative to current run start time [nminute/s] , stops every STOP_N nminutes, relative to current run start time [nhour/s] , stops every STOP_N nhours , relative to current run start time [nday/s] , stops every STOP_N ndays , relative to current run start time [nmonth/s] , stops every STOP_N nmonths , relative to current run start time [nyear/s] , stops every STOP_N nyears , relative to current run start time [date] , stops at STOP_DATE value [ifdays0] , stops at STOP_N calendar day value and seconds equal 0 [end] , stops at end [none,never,nsteps,nstep,nseconds,nsecon
TIMER_LEVEL	integer	4	timer output depth [1,2,3,4,5,6,7,8,9,10]
TIMING_BARRIER	logical	FALSE	if TRUE, turns on the timing barrier calls in the model [TRUE,FALSE]

Name	Type	Default	Description [Valid Values]
TPROF_DATE	integer	-999	yyyymmdd format, sets timing output file date (like REST_DATE)
TPROF_N	char	-999	sets timing output file frequency (like REST_N)
TPROF_OPTION	char	never	sets timing output file frequency (like REST_OPTION but relative to run start date)

Glossary

Branch

One of the three ways to initialize CCSM runs. CCSM runs can be started as startup, hybrid or branch. Branch runs use the BRANCH $RUN_TYPE and the restart files from a previous run. See **setting up a branch run.**

case

The name of your job. This can be any string.

$CASE

The case name. But when running **create_newcase**, it doubles as the case directory path name where build and run scripts are placed. $CASE is defined when you execute the **create_newcase** command;, and set in `env_case.xml`. Please see create a new case.

$CASEROOT

$CASEROOT - the full pathname of the root directory for case scripts (e.g. /user/$CASE). You define $CASEROOT when you execute the **create_newcase**, and is set in `env_case.xml`. $CASEROOT must be unique.

component

component - Each model can be run with one of several components. Examples of components include CAM, CICE, CLM, and POP. Component names will always be in all caps.

component set (compset)

Preset configuration settings for a model run. These are defined in supported component sets.

$CCSMROOT

The full pathname of the root directory of the CCSM source code. $CCSMROOT is defined when you execute the **create_newcase** command.

$EXEROOT

The case executable directory root. $EXEROOT is defined when you execute the **configure** command, and is set in `env_build.xml`.

hybrid run

A type of run. Hybrid runs use the HYBRID $RUN_TYPE and are initialized as an initial run, but use restart datasets from a previous CCSM case. Please see **setting up a hybrid run.**

$MACH

The supported machine name, and is defined in `env_case.xml` when you run the configure command. Please see **supported machines** for the list of supported machines.

model

CCSM is comprised of five models (atm, ice, glc, lnd, ocn) and the coupler, cpl. The word model is loosely used to mean any one of the models or the coupler.

model input data

Refers to static input data for the model. CCSM4 input data are provided as part of the release via an inputdata area or data from a server.

release

A supported version of CCSM4.

restart

Refers to a set of data files and pointer files that the model generates and uses to restart a case.

$RUNDIR

The directory where the model is run, output data files are created, and log files are found when the job fails. This is usually $EXEROOT/run, and is set in env_build.xml by the configure command.

tag

A version of CCSM4. Note: A tag may not be supported.