User's Guide to the Community Atmosphere Model CAM-5.3

User's Guide to the Community Atmosphere Model CAM-5.3

Publication date This document was last updated on 2014-07-09 18:28:26.

Table of Contents

Acknowledgments v
1. Introduction 1
    Changes from previous release 1
    Getting Help -- Other User Resources 2
        The CAM Web Page 2
        The CESM Bulletin Board 2
        Reporting bugs 2
2. Building and Running CAM 3
    Sample Interactive Session 4
        Configuring CAM for serial execution 4
        Specifying the Fortran compiler 5
        Dealing with compiler wrappers 7
        Configuring CAM for parallel execution 7
        Building CAM 9
        Building the Namelist 9
        Acquiring Input Datasets 12
        Running CAM 13
    Sample Run Scripts 13
3. Input datasets 14
    SST and Sea Ice Boundary Files 14
4. Model Output 16
    Model History Files 16
        General Features of History Files 16
        Default History Fields and Master Field Lists 18
5. Physics modifications via the namelist 20
    Radiative Constituents 20
        Default `rad_climate` for `cam4` physics 20
        Default `rad_climate` for `cam5` physics 21
        Diagnostic radiative forcing 24
A. The **configure** utility 26
    How **configure** is called from the CESM scripts 26
    Arguments to **configure** 26
        CAM configuration 27
        SCAM configuration 29
        CAM parallelization 29
        CAM parallelization when running standalone with CICE 29
        General options 30
        Surface components 31
        CAM standalone build 31
    Environment variables recognized by **configure** 33
B. The **build-namelist** utility 34
    Options to **build-namelist** 35
    Environment variables used by **build-namelist** 36
    CAM Namelist variables 37
References 38

List of Examples

2.1. Use **build-namelist** to specify a dataset in a non-default location. 12
2.2. Acquire missing dataset 12
4.1. Timestamps for a year of monthly averages 17
4.2. Timestamps for five daily averages 18
5.1. Modify a radiatively active gas 23
5.2. Aerosol radiative forcing 24
5.3. Black carbon radiative forcing 25

Acknowledgments

We wish to acknowledge members of NCAR's Atmospheric Modeling and Predictability Section (AMP), CESM Software Engineering Group (CSEG), and Computation and Information Systems Laboratory (CISL) for their contributions to the development of CAM-5.3.

The new model would not exist without the significant input from members of the CESM Atmospheric Model Working Group (AMWG) too numerous to mention. Rich Neale (NCAR), Minghua Zhang (SUNY), Mark Taylor (SNL) and Leo Donner (GFDL), were co-chairs of the AMWG during part or all of the development of CAM-5.3.

We would like to acknowledge the substantial contributions to the CAM effort from the National Science Foundation, the Department of Energy, the National Oceanic and Atmospheric Administration, and the National Aeronautics and Space Administration.

Chapter 1. Introduction

The Community Atmosphere Model version CAM-5.3 is released as the atmosphere component of the Community Earth System Model version CESM-1.2. It is the latest in a series of global atmosphere models whose development is guided by the Atmosphere Model Working Group [/working_groups/Atmosphere/] (AMWG) of the Community Earth System Model [/models/cesm1.2/] (CESM) project. CAM is used as both a standalone model and as the atmospheric component of the CESM. CAM has a long history of use as a standalone model by which we mean that the atmosphere is coupled to an active land model (CLM), a thermodynamic only sea ice model (a special configuration of CICE), and a data ocean model (DOCN). When one speaks of "doing CAM simulations" the implication is that it's a standalone configuration that is being used. When CAM is coupled to active ocean and sea ice models then we refer to the model as CESM.

CAM provides a framework for running the "Whole Atmosphere" configurations; WACCM, and WACCM-X. To run CAM in a WACCM or WACCM-X configuration the user is referred to the CESM-1.2 User's Guide [/models/cesm1.2/cesm/doc/usersguide/book1.html].

In versions of CAM before 4.0 the driver for the standalone configuration was completely separate code from what was used to couple the components of the CCSM. One of the most significant software changes in CAM-4.0 was a refactoring of how the land, ocean, and sea ice components are called which enabled the use of the CCSM coupler to act as the CAM standalone driver (this also depended on the complete rewritting of the CCSM coupler to support sequential execution of the components). Hence, for the CESM1 model, just as for CCSM4 before it, it is accurate to say that a CAM standalone configuration is nothing more than a special configuration of CESM in which the active ocean and sea ice components are replaced by data ocean and thermodynamic sea ice components.

Since the CAM standalone model is just a special configuration of CESM it can be run using the CESM scripts. This is done by using one of the "F" compsets and is described in the CESM-1.2 User's Guide [/models/cesm1.2/cesm/doc/usersguide/book1.html]. The main advantage of running CAM via the CESM scripts is to leverage the high level of support that those scripts provide for doing production runs of predefined experiments on supported platforms. The CESM scripts do things like: setting up reasonable runtime environments; automatically retrieving required input datasets from an SVN server; and archiving output files. But CAM is used in a lot of environments where the complexity of production ready scripts is not necessary. In these instances the flexibility and simplicity of being able to completely describe a run using a short shell script is a valuable option. In either case though, the ability to customize a CAM build or runtime configuration depends on being able to use the utilities described in this document. Any build configuration can be set up via appropriate commandline arguments to CAM's **configure** utility, and any runtime configuration can be set up with appropriate arguments to CAM's **build-namelist** utility. Issues that are specific to running CAM from the CESM scripts will not be discussed in this guide. Rather we focus on issues that are independent of which scripts are used to run CAM, although there is some attention given in this guide to the construction of simple scripts designed for running CAM in its standalone mode.

Changes from previous release

This information is available from the CESM-1.2 home page [/models/cesm1.2/].

- New science features in CAM-5.3 [/models/cesm1.2/tags/cesm1_2/whatsnew_science.html].
- New software features in CAM-5.3 [/models/cesm1.2/tags/cesm1_2/whatsnew_software.html].
- Summary of answer changes [/models/cesm1.2/tags/cesm1_2/answerchanges.html].
- Known problems [/models/cesm1.2/tags/cesm1_2/knownproblems.html].

Getting Help -- Other User Resources

The CAM Web Page

The central source for information on CAM is the CAM web page [/models/cesm1.2/cam].

The CESM Bulletin Board

The CESM Bulletin Board is a moderated forum for rapid exchange of information, ideas, and topics of interest relating to all components of the CESM. This includes sharing software tools, datasets, programming tips and examples, as well as discussions of questions, problems and workarounds. The primary motivation for the establishment of this forum is to facilitate and encourage communication between the users of the CESM around the world. This bulletin board will also be used to distribute announcements related to CESM.

The CESM Bulletin Board is here: http://bb.cgd.ucar.edu/.

Reporting bugs

If a user should encounter bugs in the code (i.e., it doesn't behave in a way in which the documentation says it should), the problem should be reported electronically to the CESM Bulletin Board [http://bb.cgd.ucar.edu/]. When writing a bug report the guiding principle should be to provide enough information so that the bug can be reproduced. The following list suggests the minimal information that should be contained in the report:

1. The version number of the CCSM or CESM release that CAM is part of.

2. The architecture on which the code was built. Include relevent information such as the Fortran compiler, MPI library, etc.

3. The **configure** commandline. If it is this command that is failing, then report the output from this command. It can also be very useful to run this command with the `-v` option to turn on verbose output.

4. The **build-namelist** commandline. If it is this command that is failing, then report the output from this command. It can also be very useful to run this command with the `-v` option to turn on verbose output.

5. Model printout. Ideally this would contain a stack trace. But it should at least contain any error messages printed to the output log.

Please note that CAM is a research tool, and not all features contained in the code base are supported.

Chapter 2. Building and Running CAM

This chapter describes how to build and run CAM in its standalone configuration. We do not provide scripts that are setup to work out of the box on a particular set of platforms. If you would like this level of support then consider running CAM from the CESM scripts (see CESM-1.2 User's Guide [/models/cesm1.2/cesm/doc/usersguide/book1.html]). We do however provide some examples of simple run scripts which should provide a useful starting point for writing your own scripts (see the section called “Sample Run Scripts”).

In order to build and run CAM the following are required:

- The source tree. CAM-5.3 is distributed with CESM-1.2. To obtain the source code go to the section "Acquiring the Code" on the CESM Home Page [/models/cesm1.2/index.html]. When we refer to the root of the CAM source tree, this is the same directory as the root of the CESM source tree. This directory is referred to throughout this document as `$CAM_ROOT`.
- Perl (version 5.4 or later).
- A GNU version of the **make** utility.
- Fortran and C compilers. The Fortran compiler needs to support at least the Fortran95 standard.
- A NetCDF library (version 4.1.3 or later) that has the Fortran APIs built using the same Fortran compiler that is used to build the rest of the CAM code. This library is used extensively by CAM both to read input datasets and to write the output datasets. The NetCDF source code is available here [http://www.unidata.ucar.edu/downloads/netcdf/]. We have updated the required NetCDF library version from 3.6 to 4.1.3 due to a recently discovered bug which affects all previous versions of the NetCDF library. The bug only occurs in special circumstances that are not that easy to replicate, however the result is that corrupt files are silently created. A more complete description of the bug is here [https://www.unidata.ucar.edu/jira/browse/NCF-22].
- Input datasets. The required datasets depend on the CAM configuration. Determining which datasets are required for any configuration is discussed in the section called “Building the Namelist”. Acquiring those datasets is discussed in the section called “Acquiring Input Datasets”.

To build CAM for SPMD execution it will also be necessary to have an MPI library (version 1 or later). As with the NetCDF library, the Fortran API should be build using the same Fortran compiler that is used to build the rest of CAM. Otherwise linking to the library may encounter difficulties, usually due to inconsistencies in Fortran name mangling.

Building and running CAM takes place in the following steps:

1. Configure model
2. Build model
3. Build namelist
4. Execute model

Configure model. This step is accomplished by running the **configure** utility to set the compile-time parameters such as the dynamical core (Eulerian Spectral, Semi-Lagrangian Spectral, Finite Volume, or Spectral Element), horizontal grid resolution, and the type of parallelism to employ (shared-memory and/or distributed memory). The **configure** utility is discussed in Appendix A, *The **configure** utility*.

Build model. This step includes compiling and linking the executable using the GNU make command (**gmake**). **configure** creates a Makefile in the directory where the build is to take place. The user then need only change to this directory and execute the **gmake** command.

Build namelist. This step is accomplished by running the **build-namelist** utility, which supports a variety of options to control the run-time behavior of the model. Any namelist variable recognized by CAM can be changed by the user via the **build-namelist** interface. There is also a high level "use case" functionality which makes it easy for the user to specify a consistent set of namelist variable settings for running particular types of experiments. The **build-namelist** utility is discussed in Appendix B, *The **build-namelist** utility*.

Execute model. This step includes the actual invocation of the executable. When running using distributed memory parallelism this step requires knowledge of how your machine invokes (or "launches") MPI executables. When running with shared-memory parallelism (using OpenMP) you may also set the number of OpenMP threads. On most HPC platforms access to the compute resource is through a batch queue system. The sample run scripts discussed in the section called “Sample Run Scripts” show how to set the batch queue resources on several HPC platforms.

Sample Interactive Session

The following sections present an interactive C shell session to build and run a default version of CAM. Most often these steps will be encapsulated in shell scripts. An important advantage of using a script is that it acts to document the run you've done. Knowing the source code tree, and the **configure** and **build-namelist** commands provides all the information needed to replicate a run.

For the interactive session the shell variable `camcfg` is set to the directory in the source tree that contains the CAM **configure** and **build-namelist** utilities (`$CAM_ROOT`/models/atm/cam/bld).

```
Much of the example code in this document is set off in sections like this.
Many examples refer to files in the distribution source tree using
filepaths that are relative to distribution root directory, which we
denote, using a UNIX shell syntax, by $CAM_ROOT.  The notation indicates
that CAM_ROOT is a shell variable that contains the filepath.  This could
just as accurately be referred to as $CCSMROOT since the root directory of
the CESM distribution is the same as the root of the CAM distribution
which is contained within it.
```

Configuring CAM for serial execution

We start by changing into the directory in which the CAM executable will be built, and then setting the environment variables `INC_NETCDF` and `LIB_NETCDF` which specify the locations of the NetCDF include files and library. This information is required by **configure** in order for it to produce the `Makefile`. The NetCDF library is require by all CAM builds. The directories given are just examples; the locations of the NetCDF include files and library are system dependent. The information provided by these environment variables could alternatively be provided via the commandline arguments **`-nc_inc`** and **`-nc_lib`**.

NOTE: A common problem is to encounter build failures due to specifying a NetCDF library which was built with a different Fortran compiler than the one used to build CAM. Consult your system's documentation (or some other knowledgeable source) to find the location of the NetCDF library which was built with the Fortran compiler you intend to use.

```
% cd /work/user/cam_test/bld
% setenv INC_NETCDF /usr/local/include
% setenv LIB_NETCDF /usr/local/lib
```

Next we issue the **configure** command (see the example just below). The argument **`-dyn fv`** specifies using the FV dynamical core which is the default for CAM5, but we recommend always adding the dynamical core (dycore for short) argument to **configure** commands for clarity. The argument **`-hgrid 10x15`** specifies the horizontal grid. This is the coarsest grid available for the FV dycore in CAM and is often useful for testing purposes.

We recommend using the **`-test`** option the first time CAM is built on any machine. This will check that the environment is properly set up so that the Fortran compiler works and can successfully link to the NetCDF and MPI (if SPMD is enabled) libraries. Furthermore, if the configuration is for serial execution, then the tests will include both build and run phases which may be useful in exposing run time problems that don't show up during the build, for example when shared libraries are linked dynamically. If any tests fail then it is useful to rerun the **configure** command and add the **`-v`** option which will produce verbose output of all aspects of the configuration process including the tests. If the configuration is for an SPMD build, then no attempt to run the tests will be made. Typically MPI runs must be submitted to a batch queue and are not enabled from interactive sessions. Also the method of launching an MPI job is system dependent. But the build and static linking will still be tested.

```
% $camcfg/configure -dyn fv -hgrid 10x15 -nospmd -nosmp -test
Issuing command to the CICE configure utility:
  $CAM_ROOT/models/ice/cice/bld/configure -hgrid 10x15 -cice_mode prescribed \
  -ntr_aero 0 -nx 24 -ny 19 -bsizex 6 -bsizey 19 -maxblocks 4 -decomptype blkrobin
  -cache config_cache_cice.xml -cachedir /work/user/cam_test/bld
CICE configure done.
MCT configure is done.
creating /work/user/cam_test/bld/Filepath
creating /work/user/cam_test/bld/Makefile
creating /work/user/cam_test/bld/config.h
creating /work/user/cam_test/bld/config_cache.xml
Looking for a valid GNU make... using gmake
Testing for Fortran 90 compatible compiler... using pgf95
Test linking to NetCDF library... ok
CAM configure done.
```

The first line of output from the **configure** command is an echo of the system command that CAM's **configure** issues to invoke the CICE **configure** utility. CICE's **configure** is responsible for setting the values of the CPP macros that are needed to build the CICE code.

After the CICE **configure** is complete the MCT **configure** script is executed to create the Makefile for building MCT as a separate library. There is a status line output to indicate success of that process.

The next four lines of output inform the user of the files being created by **configure**. All these files except for the cache file are required to be in the CAM build directory, so it is generally easiest to be in that directory when **configure** is invoked.

The output from the **`-test`** option tells us that **gmake** is a GNU Make on this machine; that the Fortran compiler is **pgf95**; and that code compiled with the Fortran compiler can be successfully linked to the NetCDF library. The CAM **configure** script is the place where the default compilers are specified. On Linux systems the default is **pgf95**. Finally, since this is a serial configuration no test for linking to the MPI library was done.

Specifying the Fortran compiler

In the previous section the **configure** command was issued without specifying which Fortran compiler to use. For that to work we were depending on the CAM **configure** script to select a default compiler.

One of the differences between the CAM standalone build and a build using the CESM scripts is that CAM's **configure** provides defaults based on the operating system name (as determined by the Perl internal variable `$OSNAME`), while the CESM scripts require the user to specify a specific machine (and compiler if the machine supports more than one) as an argument to the **create_newcase** command.

The CAM makefile currently recognizes the following operating systems and compilers.

AIX	xlf95_r, mpxlf95_r
Linux	pgf95 (this is the default)
	lf95
	ifort
	gfortran (has had minimal testing)
	pathf90 (has had minimal testing)
Darwin	xlf95_r, mpxlf95_r, ifort
BGL	blrts_xlf95
BGP	mpixlf95_r

The above list contains two IBM Blue Gene machines; BGL and BGP. The executables on these machines are produced by cross compilation and hence the configure script is not able to determine the machine for which the build is intented. In this case the user must supply this information to **configure** by using the **`-target_os`** option with the values of either **`bgl`** or **`bgp`**.

On a Linux platform several compilers are recognized with the default being **pgf95**. It is assumed that the compiler to be used is in the user's path (i.e., in one of the directories in the `PATH` environment variable). If it isn't then the **`-test`** option will issue an error indicating that the compiler was not found.

Suppose for example that one would like to use the Intel compiler on a local Linux system. The CAM makefile recognizes **ifort** as the name of the Intel compiler. To invoke this compiler use the **`-fc`** argument to **configure**. The following example illustrates the output you get when the compiler you ask for isn't in your `PATH`:

```
% $camcfg/configure -fc ifort -dyn fv -hgrid 10x15 -nospmd -nosmp -test
Issuing command to the CICE configure utility:
  $CAM_ROOT/models/ice/cice/bld/configure -hgrid 10x15 -cice_mode prescribed \
  -ntr_aero 0 -ntasks 1 -nthreads 1 -cache config_cache_cice.xml \
  -cachedir /work/user/cam_test/bld
CICE configure done.
FAILURE: MCT configure
```

In previous CAM versions this problem would be caught by the **`-test`** option, but with the addition of MCT's **configure** the problem is now detected there. By default MCT will be build in a subdirectory of the build directory named `mct`. That directory will contain a file, `config.log`, which should be examined to track down the cause of the failure. In this case the file contains the message:

```
$CAM_ROOT/models/utils/mct/configure: line 3558: ifort: command not found
```

This means that the PATH environment variable has not been correctly set. The first thing to try is to verify the directory that contains the compiler, and then to prepend this directory name to the PATH environment variable.

NOTE: We have made progress porting CAM to the **gfortran** compiler, but it is still not regularly tested or used for production work.

Dealing with compiler wrappers

Another instance where the user needs to supply information about the Fortran compiler type to configure is when the compiler is being invoked by a wrapper script. A common example of this is using the **mpif90** command to invoke the Fortran compiler that was used to build the MPI libraries. This facilitates correct compilation and linking with the MPI libraries without the user needing to add the required include and library directories, or library names. The same benefit is provided by the **ftn** wrapper used on Cray XT and XE systems. In the usual case that a Linux OS is being used, since the CAM makefile will not recognize these compiler names, it will assume that the default compiler is being used, and thus will supply compiler arguments that are appropriate for **pgf90**. The compilation will fail if **pgf90** is not the compiler being invoked by the wrapper script (invoking **configure** with the **`-test`** option is a good way to catch this problem). The way to specify which Fortran compiler is being invoked by a wrapper script is via the **`-fc_type`** argument to configure. This argument takes one of the values **`pgi`**, **`lahey`**, **`intel`**, **`pathscale`**, **`gnu`**, or **`xlf`**.

CAM's **configure** script attempts to determine the compiler type when a compiler specific name is used. It does so by a regular expression match against the unique part of specific compiler names (e.g., any compiler name matching 'pgf' will be given the default type of pgi). If the default is wrong then the user will need to manually override the default via setting the **`-fc_type`** argument.

Configuring CAM for parallel execution

Before moving on to building CAM we address configuring the executable for parallel execution. But before talking about configuration specifics let's briefly discuss the parallel execution capabilities of CAM.

CAM makes use of both distributed memory parallelism implemented using MPI (referred to throughout this document as SPMD [http://en.wikipedia.org/wiki/SPMD]), and shared memory parallelism implemented using OpenMP (referred to as SMP [http://en.wikipedia.org/wiki/Symmetric_multiprocessing]). Each of these parallel modes may be used independently of the other, or they may be used at the same time which we refer to as "hybrid mode". When talking about the SPMD mode we usually refer to the MPI processes as "tasks", and when talking about the SMP mode we usually refer to the OpenMP processes as "threads". A feature of CAM which is very helpful in code development work is that the simulation results are independent of the number of tasks and threads used.

Now consider configuring CAM to run in pure SPMD mode. Prior to the introduction of CICE as the sea ice model SPMD was turned on using the **`-spmd`** option. But if we try that now we find the following:

```
% $camcfg/configure -dyn fv -hgrid 10x15 -spmd -nosmp
**    ERROR: If CICE decomposition parameters are not specified, then
**    -ntasks must be specified to determine a default decomposition
**    for a pure MPI run.  The setting was:  ntasks=
```

A requirement of the CICE model is that its grid decomposition (which is independent of CAM's decomposition even when the two models are using the same horizontal grid) must be specified at build time. In order for CICE's **configure** to set the decomposition it needs to know how much parallelism is

going to be used. This information is provided by specifying the number of MPI tasks that the job will use via setting the **-ntasks** argument.

NOTE: The default CICE decomposition can be overridden by setting it explicitly using the **configure** options provided for that purpose.

When running CAM in SPMD mode the build procedure must be able to find the MPI include files and library. The recommended method for doing this is to use scripts provided by the MPI installation to invoke the compiler and linker. On Linux systems a common name for this script is **mpif90**. The CAM Makefile does not currently use this script by default on Linux platforms, so the user must explicitly specify it on the **configure** commandline using the **-fc** argument:

```
% $camcfg/configure -fc mpif90 -fc_type pgi -cc mpicc -dyn fv -hgrid 10x15 -ntasks
Issuing command to the CICE configure utility:
  $CAM_ROOT/models/ice/cice/bld/configure -hgrid 10x15 -cice_mode prescribed \
  -ntr_aero 0 -ntasks 6 -nthreads 1 -cache config_cache_cice.xml \
  -cachedir /work/user/cam_test/bld
CICE configure done.
MCT configure is done.
creating /work/user/cam_test/bld/Filepath
creating /work/user/cam_test/bld/Makefile
creating /work/user/cam_test/bld/config.h
creating /work/user/cam_test/bld/config_cache.xml
Looking for a valid GNU make... using gmake
Testing for Fortran 90 compatible compiler... using mpif90
Test linking to NetCDF library... ok
Test linking to MPI library... ok
CAM configure done.
```

Notice that the number of tasks specified to CAM's **configure** is passed through to the commandline that invokes the CICE **configure**. Generally any number of tasks that is appropriate for CAM to use for a particular horizontal grid will also work for CICE. But it is possible to get an error from CICE at this point in which case either the number of tasks requested should be adjusted, or the options that set the CICE decomposition explicitly will need to be used.

NOTE: The use of the **-ntasks** argument to **configure** implies building for SPMD. This means that an MPI library will be required. Hence, the specification **-ntasks 1** is not the same as building for serial execution which is done via the **-nospmd** option and does not require a full MPI library. (Implementation detail: when building for serial mode a special serial MPI library is used which basically provides a complete MPI API, but doesn't do any message passing.)

Next consider configuring CAM to run in pure SMP mode. Similarly to SPMD mode, prior to the introduction of the sea ice component CICE the SMP mode was turned on using the **-smp** option. But with CAM5 that will result in the same error from CICE that we obtained above from attempting to use **-spmd**. If we are going to run the CICE code in parallel, we need to specify up front how much parallelism will be used so that the CICE **configure** utility can set the CPP macros that determine the grid decomposition. We specify the amount of SMP parallelism by setting the **-nthreads** option as follows:

```
% $camcfg/configure -dyn fv -hgrid 10x15 -nospmd -nthreads 6 -test
Issuing command to the CICE configure utility:
  $CAM_ROOT/models/ice/cice/bld/configure -hgrid 10x15 -cice_mode prescribed \
  -ntr_aero 0 -ntasks 1 -nthreads 6 -cache config_cache_cice.xml \
  -cachedir /work/user/cam_test/bld
```

```
CICE configure done.
...
```

We see that the number of threads has been passed through to the CICE **configure** command.

NOTE: The use of the **-nthreads** argument to **configure** implies building for SMP. This means that the OpenMP directives will be compiled. Hence, the specification **-nthreads 1** is not the same as building for serial execution which is done via the **-nosmp** option and does not require a compiler that supports OpenMP.

Finally, to configure CAM for hybrid mode, simply specify both the **-ntasks** and **-nthreads** arguments to **configure**.

Building CAM

Once **configure** is successful, build CAM by issuing the make command:

```
% gmake -j2  >&! make.out
```

The argument **-j2** is given to allow a parallel build using 2 processes. The optimal number of processes to use depends on the compute resource available. There is a lot of available parallelism in the build procedure, so using 16 or even 32 processes may speed things up considerably. Note however that the build happens in shared (not distributed) memory. So specifying more processes than there are processors in a shared memory node is generally not helpful (although the presence of hyperthreading or SMT on a node may provide an advantage to specifying twice the number of processors).

It is useful to redirect the output from **make** to a file for later reference. This file contains the exact commands that were issued to compile each file and the final command which links everything into an executable file. Relevant information from this file should be included when posting a bug report concerning a build failure.

Building the Namelist

The first step in the run procedure is to generate the namelist files. The safest way to generate consistent namelist settings is via the **build-namelist** utility. Even in the case where only a slight modification to the namelist is desired, the best practice is to provide the modified value as an argument to **build-namelist** and allow it to actually generate the namelist files.

NOTE: The default configuration of CAM using the **cam5** physics package requires that about 60 datasets and dozens of parameter values be specified in order to run correctly. Trying to manage namelists of that complexity by hand editing files is extremely error prone and is strongly discouraged. User modifications to the default namelist settings can be made in a number of ways while still letting **build-namelist** actually generate the final namelist. In particular, the **-namelist**, **-infile**, and **-use_case** arguments to **build-namelist** are all mechanisms by which the user can override default values or specify additional namelist variables and still allow build-namelist to do the error and consistency checking which makes the namelist creation process more robust.

The following interactive C shell session builds a default namelist for CAM. We assume that a successful execution of **configure** was performed in the build directory as discussed in the previous sections. This is an essential prerequisite because the `config_cache.xml` file produced by **configure** is a required input file to **build-namelist**. One of the responsibilities of **build-namelist** is to set appropriate default values for many namelist variables, and it can only do this if it knows how the CAM executable was configured.

That information is present in the cache file. As in the previous section the shell variable `camcfg` is set to the CAM configuration directory (`$CAM_ROOT`/models/atm/cam/bld).

We begin by changing into the directory where CAM will be run. It is usually convenient to have the run directory be separate from the build directory. Possibly a number of different runs will be done that each need to have a separate run directory for the output files, but will all use the same executable file from a common build directory. It is, of course, possible to execute **build-namelist** in the build directory since that's where the cache file is and so you don't need to specify to **build-namelist** where to find that file (it looks in the current working directory by default). But then, assuming you plan to run CAM in a different directory, all the files produced by **build-namelist** need to be copied to the run directly. If you're running **configure** and **build-namelist** from a script, then you need to know how to specify the filenames for the files that need to be copied. For this reason it's more robust to change to the run directory and execute **build-namelist** there. That way if there's a change to the files that are produced, your script doesn't break due to the files not all getting copied to the run directory.

Next we set the `CSMDATA` environment variable to point to the root directory of the tree containing the input data files. Note that this is a required input for **build-namelist** (this information may alternatively be provided using the **`-csmdata`** argument). If not provided then **build-namelist** will fail with an informative message. The information is required because many of the namelist variables have values that are absolute filepaths. These filepaths are resolved by **build-namelist** by prepending the `CSMDATA` root to the relative filepaths that are stored in the default values database.

The **build-namelist** commandline contains the **`-config`** argument which is used to point to the cache file which was produced in the build directory. It also contains the **`-test`** argument, explained further below.

```
% cd /work/user/cam_test
% setenv CSMDATA /fs/cgd/csm/inputdata
% $camcfg/build-namelist -test -config /work/user/cam_test/bld/config_cache.xml
Writing CICE namelist to ./ice_in
Writing RTM namelist to ./rof_in
Writing DOCN namelist to ./docn_ocn_in
Writing DOCN stream file to ./docn.stream.txt
Writing CLM namelist to ./lnd_in
Writing driver namelist to ./drv_in
CAM writing dry deposition namelist to drv_flds_in
Writing ocean component namelist to ./docn_in
CAM writing namelist to atm_in
Checking whether input datasets exist locally...
OK -- found depvel_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart/dvel/depv
OK -- found tracer_cnst_filelist = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart_
OK -- found tracer_cnst_datapath = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart_
OK -- found depvel_lnd_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart/dvel/
OK -- found xs_long_file = /fs/cgd/csm/inputdata/atm/waccm/phot/temp_prs_GT200nm_j
OK -- found rsf_file = /fs/cgd/csm/inputdata/atm/waccm/phot/RSF_GT200nm_v3.0_c0804
OK -- found clim_soilw_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart/dvel/
OK -- found exo_coldens_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart/phot
OK -- found tracer_cnst_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart_aero
OK -- found season_wes_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart/dvel/
OK -- found solar_data_file = /fs/cgd/csm/inputdata/atm/cam/solar/solar_ave_sc19-s
OK -- found soil_erod = /fs/cgd/csm/inputdata/atm/cam/dst/dst_10x15_c090203.nc
OK -- found bndtvs = /fs/cgd/csm/inputdata/atm/cam/sst/sst_HadOIBl_bc_10x15_clim_c
OK -- found focndomain = /fs/cgd/csm/inputdata/atm/cam/ocnfrac/domain.camocn.10x15
OK -- found tropopause_climo_file = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mozart
OK -- found fpftcon = /fs/cgd/csm/inputdata/lnd/clm2/pftdata/pft-physiology.c11042
```

```
OK -- found fsnowaging = /fs/cgd/csm/inputdata/lnd/clm2/snicardata/snicar_drdt_bst
OK -- found fatmlndfrc = /fs/cgd/csm/inputdata/share/domains/domain.lnd.fv10x15_US
OK -- found fsnowoptics = /fs/cgd/csm/inputdata/lnd/clm2/snicardata/snicar_optics_
OK -- found fsurdat = /fs/cgd/csm/inputdata/lnd/clm2/surfdata/surfdata_10x15_simyr
OK -- found prescribed_ozone_datapath = /fs/cgd/csm/inputdata/atm/cam/ozone
OK -- found prescribed_ozone_file = /fs/cgd/csm/inputdata/atm/cam/ozone/ozone_1.9x
OK -- found liqopticsfile = /fs/cgd/csm/inputdata/atm/cam/physprops/F_nwvl200_mu20
OK -- found iceopticsfile = /fs/cgd/csm/inputdata/atm/cam/physprops/iceoptics_c080
OK -- found water_refindex_file = /fs/cgd/csm/inputdata/atm/cam/physprops/water_re
OK -- found ncdata = /fs/cgd/csm/inputdata/atm/cam/inic/fv/cami_0000-01-01_10x15_L
OK -- found bnd_topo = /fs/cgd/csm/inputdata/atm/cam/topo/USGS-gtopo30_10x15_remap
OK -- found ext_frc_specifier for SO2 = /fs/cgd/csm/inputdata/atm/cam/chem/trop_mo
OK -- found ext_frc_specifier for bc_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/trop_
OK -- found ext_frc_specifier for num_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/trop
OK -- found ext_frc_specifier for num_a2 = /fs/cgd/csm/inputdata/atm/cam/chem/trop
OK -- found ext_frc_specifier for pom_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/trop
OK -- found ext_frc_specifier for so4_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/trop
OK -- found ext_frc_specifier for so4_a2 = /fs/cgd/csm/inputdata/atm/cam/chem/trop
OK -- found srf_emis_specifier for DMS = /fs/cgd/csm/inputdata/atm/cam/chem/trop_m
OK -- found srf_emis_specifier for SO2 = /fs/cgd/csm/inputdata/atm/cam/chem/trop_m
OK -- found srf_emis_specifier for SOAG = /fs/cgd/csm/inputdata/atm/cam/chem/trop_
OK -- found srf_emis_specifier for bc_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/trop
OK -- found srf_emis_specifier for num_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/tro
OK -- found srf_emis_specifier for num_a2 = /fs/cgd/csm/inputdata/atm/cam/chem/tro
OK -- found srf_emis_specifier for pom_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/tro
OK -- found srf_emis_specifier for so4_a1 = /fs/cgd/csm/inputdata/atm/cam/chem/tro
OK -- found srf_emis_specifier for so4_a2 = /fs/cgd/csm/inputdata/atm/cam/chem/tro
OK -- found mode_defs for so4_a1 = /fs/cgd/csm/inputdata/atm/cam/physprops/sulfate
OK -- found mode_defs for pom_a1 = /fs/cgd/csm/inputdata/atm/cam/physprops/ocpho_r
OK -- found mode_defs for soa_a1 = /fs/cgd/csm/inputdata/atm/cam/physprops/ocphi_r
OK -- found mode_defs for bc_a1 = /fs/cgd/csm/inputdata/atm/cam/physprops/bcpho_rr
OK -- found mode_defs for dst_a1 = /fs/cgd/csm/inputdata/atm/cam/physprops/dust4_r
OK -- found mode_defs for ncl_a1 = /fs/cgd/csm/inputdata/atm/cam/physprops/ssam_rr
OK -- found mode_defs for so4_a2 = /fs/cgd/csm/inputdata/atm/cam/physprops/sulfate
OK -- found mode_defs for soa_a2 = /fs/cgd/csm/inputdata/atm/cam/physprops/ocphi_r
OK -- found mode_defs for ncl_a2 = /fs/cgd/csm/inputdata/atm/cam/physprops/ssam_rr
OK -- found mode_defs for dst_a3 = /fs/cgd/csm/inputdata/atm/cam/physprops/dust4_r
OK -- found mode_defs for ncl_a3 = /fs/cgd/csm/inputdata/atm/cam/physprops/ssam_rr
OK -- found mode_defs for so4_a3 = /fs/cgd/csm/inputdata/atm/cam/physprops/sulfate
OK -- found rad_climate for mam3_mode1 = /fs/cgd/csm/inputdata/atm/cam/physprops/m
OK -- found rad_climate for mam3_mode2 = /fs/cgd/csm/inputdata/atm/cam/physprops/m
OK -- found rad_climate for mam3_mode3 = /fs/cgd/csm/inputdata/atm/cam/physprops/m
```

The first nine lines of output from **build-namelist** inform the user about the files that have been created. There are namelist files for the ice component (`ice_in`), the river runoff component (`rof_in`), the land component (`lnd_in`), the data ocean component (`docn_in`, `docn_ocn_in`), the atmosphere component (`atm_in`), the driver (`drv_in`), and a file that is read by both the atmosphere and land components (`drv_flds_in`). There is also a "stream file" (`docn.stream.txt`) which is read by the data ocean component. Note that these filenames are hardcoded in the components and cannot be changed without source code modifications.

The next section of output is the result of using the **`-test`** argument to **build-namelist**. As with **configure** we recommend using this argument whenever a model configuration is being run for the first time. It checks that each of the files that are present in the generated namelists can be found in the input data tree

whose root is given by the CSMDATA environment variable. If a file is not found then the user will need to take steps to make that file accessible to the executing model before a successful run will be possible. The following is a list of possible actions:

- Acquire the missing file. If this is a default file supplied by the CESM project then you will be able to download the file from the project's svn data repository (see the section called “Acquiring Input Datasets”).
- If you have write permissions in the directory under $CSMDATA then add the missing file to the appropriate location there.
- If you don't have write permissions under $CSMDATA then put the file in a place where you can (for example, your run directory) and rerun **build-namelist** with an explicit setting for the file using your specific filepath.

Example 2.1. Use build-namelist to specify a dataset in a non-default location.

Suppose that the **-test** option informed you that the ncdata file cami_0000-01-01_10x15_L30_c081013.nc was not found. You acquire the file from the data repository, but don't have permissions to write in the $CSMDATA tree. So you put the file in your run directory and issue a **build-namelist** command that looks like this:

```
% $camcfg/build-namelist -config /work/user/cam_test/bld/config_cache.xml \
  -namelist "&atm ncdata='/work/user/cam_test/cami_0000-01-01_10x15_L30_c081013.nc
```

Now the namelist in atm_in will contain an initial file (specified by namelist variable ncdata) which will be found by the executing CAM model.

Acquiring Input Datasets

If you are doing a standard production run that is supported in the CESM scripts, then using those scripts will automatically invoke a utility to acquire needed input datasets. The information in this section is to aid developers using CAM standalone scripts.

The input datasets required to run CAM are available from a Subversion repository located here: https://svn-ccsm-inputdata.cgd.ucar.edu/trunk/inputdata/. The user name and password for the input data repository will be the same as for the code repository (which are provided to users when they register to acquire access to the CESM source code repository).

Example 2.2. Acquire missing dataset

If you have a list of files that you need to acquire before running CAM, then you can either just issue commands interactively, or if your list is rather long then you may want to put the commands into a shell script. For example, suppose after running **build-namelist** with the **-test** option you find that you need to acquire the file /fs/cgd/csm/inputdata/atm/cam/inic/fv/cami_0000-01-01_10x15_L26_c030918.nc. And let's assume that /fs/cgd/csm/inputdata/ is the root directory of the inputdata tree, and that you have permissions to write there. If the subdirectory atm/cam/inic/fv/ doesn't already exist, then create it. Finally, issue the following commands at an interactive C shell prompt:

```
% set svnrepo='https://svn-ccsm-inputdata.cgd.ucar.edu/trunk/inputdata'
% cd /fs/cgd/csm/inputdata/atm/cam/inic/fv
% svn export $svnrepo/atm/cam/inic/fv/cami_0000-01-01_10x15_L26_c030918.nc
```

```
Error validating server certificate for 'https://svn-ccsm-inputdata.cgd.ucar.edu:4
 - 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? p
A    cami_0000-01-01_10x15_L26_c030918.nc
Export complete.
```

The messages about validating the server certificate will only occur for the first file that you export if you answer "p" to the question as in the example above.

Running CAM

Once the namelist files have successfully been produced, and the necessary input datasets are available, the model is ready to run. Usually CAM will be run with SPMD parallelization enabled, and this requires setting up MPI resources and possibly dealing with batch queues. These issues will be addressed briefly in the section called “Sample Run Scripts”. But for a simple test in serial mode executed from an interactive shell, we only need to issue the following command:

```
% /work/user/cam_test/bld/cam >&! cam.log
```

The commandline above redirects STDOUT and STDERR to the file `cam.log`. The CAM logfile contains a substantial amount of information from all components that can be used to verify that the model is running as expected. Things like namelist variable settings, input datasets used, and output datasets created are all echoed to the log file. This is the first place to look for problems when a model run is unsuccessful. It is also very useful to include relevant information from the logfile when submitting bug reports.

Sample Run Scripts

Chapter 3. Input datasets

The minimal CAM configuration requires an initial conditions dataset. But most configurations require in addition to initial conditions a variety of boundary condition files. This chapter will provide an overview of CAM's dataset requirements and some information on the provenance of the default datasets.

SST and Sea Ice Boundary Files

The standard CAM standalone configuration (An F compset when using CESM scripts) uses prescribed sea surface temperatures (SST) and sea ice fractions from datasets containing either climatological or time series data. The source of this data for CAM's default datasets is Hurrell et al. [2008].

The default CAM datasets have been preconditioned to comply with the AMIP II requirement as described in Taylor et al. [2000]. The requirement is that the SST and sea-ice concentration boundary conditions should be specified such that the monthly means computed from model output precisely agree with the monthly means in the input dataset.

The original Hurrell datasets are on a 1 degree grid. The AMIP II versions of these dataset are available as 1 degree datasets and have also been spatially interpolated to several spectral and finite volume grid resolutions. The currently available datasets are:

Pre-industrial climatology (1870 - 1890):

```
atm/cam/sst/sst_HadOIBl_bc_1x1_clim_pi_c101029.nc
atm/cam/sst/sst_HadOIBl_bc_0.23x0.31_clim_pi_c091020.nc
atm/cam/sst/sst_HadOIBl_bc_0.47x0.63_clim_pi_c100128.nc
atm/cam/sst/sst_HadOIBl_bc_0.9x1.25_clim_pi_c100127.nc
atm/cam/sst/sst_HadOIBl_bc_1.9x2.5_clim_pi_c100127.nc
atm/cam/sst/sst_HadOIBl_bc_4x5_clim_pi_c100127.nc
atm/cam/sst/sst_HadOIBl_bc_10x15_clim_pi_c100127.nc
atm/cam/sst/sst_HadOIBl_bc_128x256_clim_pi_c100128.nc
atm/cam/sst/sst_HadOIBl_bc_64x128_clim_pi_c100128.nc
atm/cam/sst/sst_HadOIBl_bc_48x96_clim_pi_c100128.nc
atm/cam/sst/sst_HadOIBl_bc_32x64_clim_pi_c100128.nc
atm/cam/sst/sst_HadOIBl_bc_8x16_clim_pi_c100128.nc
```

Historical Time Series

```
atm/cam/sst/sst_HadOIBl_bc_1x1_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_0.23x0.31_1850_2010_c110526.nc
atm/cam/sst/sst_HadOIBl_bc_0.47x0.63_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_0.9x1.25_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_1.9x2.5_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_4x5_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_10x15_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_128x256_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_64x128_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_48x96_1850_2008_c100128.nc
atm/cam/sst/sst_HadOIBl_bc_32x64_1850_2012_c130411.nc
atm/cam/sst/sst_HadOIBl_bc_8x16_1850_2012_c130411.nc
```

Present day climatology (1982 - 2001):

```
atm/cam/sst/sst_HadOIBl_bc_1x1_clim_c101029.nc
atm/cam/sst/sst_HadOIBl_bc_0.23x0.31_clim_c061106.nc
atm/cam/sst/sst_HadOIBl_bc_0.47x0.63_clim_c061106.nc
atm/cam/sst/sst_HadOIBl_bc_0.9x1.25_clim_c040926a.nc
atm/cam/sst/sst_HadOIBl_bc_1.9x2.5_clim_c061031.nc
atm/cam/sst/sst_HadOIBl_bc_4x5_clim_c061031.nc
atm/cam/sst/sst_HadOIBl_bc_10x15_clim_c050526.nc
atm/cam/sst/sst_HadOIBl_bc_256x512_clim_c031031.nc
atm/cam/sst/sst_HadOIBl_bc_128x256_clim_c050526.nc
atm/cam/sst/sst_HadOIBl_bc_64x128_clim_c050526.nc
atm/cam/sst/sst_HadOIBl_bc_48x96_clim_c050526.nc
atm/cam/sst/sst_HadOIBl_bc_32x64_clim_c050526.nc
atm/cam/sst/sst_HadOIBl_bc_8x16_clim_c050526.nc
```

Chapter 4. Model Output

CAM produces a series of NetCDF format history files containing atmospheric gridpoint data generated during the course of a run. It also produces a series of NetCDF format restart files necessary to continue a run once it has terminated successfully and a series of initial conditions files that may be used to initialize new simulations. The contents of these datasets are described below.

Model History Files

History files contain model data values written at specified frequencies during a run. Options are also available to record averaged, instantaneous, maximum, or minimum values on a field-by-field basis. If the user wishes to see a field written at more than one time frequency (e.g. daily, hourly), additional history files must be declared. This functionality is available via setting namelist variables.

History files may be visualized using various commercial or freely available tools. Examples include the the NCAR Graphics package (via NCL), CDAT, FERRET, ncview, MATLAB, and IDL. For a list of software tools for interacting with NetCDF files, view the UNIDATA maintained link Software for Manipulating or Displaying NetCDF Data [http://www.unidata.ucar.edu/software/netcdf/software.html].

General Features of History Files

CAM writes a sequence of time samples to each of its specified history files. There can currently be from one to six history file streams, and each stream has its own set of the following attributes:

- fields
- output frequency
- maximun number of time samples in a file
- output precision (4-byte or 8-byte floats)
- output domain (global or rectangular subdomains)

Each time sample in a history file has an associated timestamp which conforms to the CF metadata conventions [http://cfconventions.org/]. The time unit used in CAM's output files is "days since reference date" where the reference date is the run start date by default, but can be customized via the `ref_ymd` and `ref_tod` namelist variables. The variables relevant to the timestamps are the following (from the output of the NetCDF **ncdump** utility):

```
double time(time) ;
        time:long_name = "time" ;
        time:units = "days since 0000-01-01 00:00:00" ;
        time:calendar = "noleap" ;
        time:bounds = "time_bnds" ;

double time_bnds(time, nbnd) ;
        time_bnds:long_name = "time interval endpoints" ;

int date(time) ;
        date:long_name = "current date (YYYYMMDD)" ;

int datesec(time) ;
```

```
datesec:long_name = "current seconds of current date" ;
```

The variable names, `time`, `time_bnds`, `date`, and `datesec` are all local conventions. What makes the history files CF compliant is that the time coordinate, `time`, can be identified by it's units attribute "days since 0000-01-01 00:00:00". The reference date is in the form YYYY-MM-DD HH:MM:SS where YYYY, MM, DD, HH, MM, SS are year, month, day, hour, minute, second respectively, and a missing timezone defaults to UTC. The `calendar` and `bounds` attributes are also part of CF. The `calendar` value "`noleap`" denotes the Gregorian calendar with no leap years. The `bounds` value `time_bnds` denotes that the variable with the name `time_bnds` contains the timestamps that bound the time intervals over which an operation such as computing an averager or a minimum or maximum value has been applied. Whether or not the interval specified by `time_bnds` is relevent depends on the individual variables, e.g., a single file can contain both instantaneous and time averaged fields. The type of the time operation that has been applied is contained in the `cell_methods` attribute of each variable, e.g.,

```
float T(time, lev, lat, lon) ;
        T:mdims = 1 ;
        T:units = "K" ;
        T:long_name = "Temperature" ;
        T:cell_methods = "time: mean" ;
```

The `cell_methods` attribute for the temperature variable indicates that it is being output as a time averaged field. If temperature was instantaneous then the `cell_methods` attribute would not be present since instantaneous is the default.

The variables `date` and `datesec` are for convenience only; they don't play any role in terms of CF compliance. The `date` variable is an integer which is encoded to contain the digits YYYYMMDD where YYYY, MM, and DD are the year, month, and day of month respectively. `datesec` is the integer number of seconds past 0Z in the current day. The variables `date` and `datesec` are redundant in the sense that they can be recovered from the `time` variable via a date calculation using the specified calendar.

Timestamps and time intervals

The timestamp associated with each time sample in a history file is the model time at the end of the timestep during which the model writes data to the disk. In the case of instantaneous data the meaning is clear. However when the data is representative of a time interval, the timestamp corresponds to *the end of the interval*.

This is often a point of confusion when looking at history files. Since the endpoint of one interval is the same as the begining of the next interval, when looking at a monthly average for January, which has a timestamp of 0Z on Feb 01, at first glance the timestamp would seem to correspond to a February average. Hence it's important for post processing tools to make use of the data in the `time_bnds` variable so that the time interval endpoints can be used to compute an interval midpoint which is the more appropriate timestamp to associate with the interval.

Example 4.1. Timestamps for a year of monthly averages

Here are the timestamps and corresponding time interval bounds for a one year sequence of monthly averages starting at 0000-01-01 00:00:00.

```
Month    time    date   datesec   time_bnds
Jan       31      201      0        0,  31
Feb       59      301      0       31,  59
Mar       90      401      0       59,  90
```

Apr	120	501	0	90, 120
May	151	601	0	120, 151
Jun	181	701	0	151, 181
Jul	212	801	0	181, 212
Aug	243	901	0	212, 243
Sep	273	1001	0	243, 273
Oct	304	1101	0	273, 304
Nov	334	1201	0	304, 334
Dec	365	10101	0	334, 365

Multiple time samples in a single file

CAM's default history output is a sequence of monthly averaged fields, written with one time sample per file. This restriction is related to the default file naming scheme which uses the string "YYYY-MM" to indicate the year and month of the average contained in the file. However in general it is possible to write multiple time samples in any of the history file streams that don't contain monthly time intervals. However there is one somewhat unexpected "feature" of multiple time sample files that we wish to point out here.

Example 4.2. Timestamps for five daily averages

Here are the timestamps and corresponding time interval bounds for all time samples written to a single file from a 5 day run starting at 0000-01-01 00:00:00.

Sample	time	date	datesec	time_bnds
1	0	101	0	0, 0
2	1	102	0	0, 1
3	2	103	0	1, 2
4	3	104	0	2, 3
5	4	105	0	4, 5
6	5	106	0	5, 6

Instead of ending up with a file containing five time samples, i.e., a daily average for each of the first five days of January, we get six time samples. The first one looks a bit strange since the time bounds are indicating an interval of zero duration. But in fact that's correct for the first time sample which is instantaneous data representing the initial conditions which have only been modify by a partial first step up to the point of the radiation calculation. This "extra" time sample from the initialization phase is included in every history file except for the monthly average file. An unfortunate consequence of this extra time sample is that it's not possible to create a sequence of files with the same number of time intervals since the first file in the sequence will always have one fewer time interval than the rest due to the inclusion of the time zero sample.

Default History Fields and Master Field Lists

CAM is set up by default to output a set of fields to a single monthly average history file. There is a much larger set of available fields, known as the "master field list," from which the user can choose fields of interest to add to the history file via namelist settings. Both the set of default fields and the master field list depend on how CAM is configured. Due to the large number of fields we have chosen to make lists of fields for some standard configuration available via linked documents rather than to inline the lists here. Each of the field list documents is comprised of tables containing the lists of fields that are output by default as well as the master field list.

NOTE: The master field list tables may contain some fields that are not actually available for output. The presence of a field in the master field list is a necessary, but not sufficient condition that the

corresponding field in the history file will contain valid data. This is because in some instances fields are added to the master field list (this is done in the source code) even though that field may not be computed in the configuration that is built (specified via the arguments to **configure**). When adding non-default fields to the history file it's important to check that the fields contain reasonable data before doing a long run.

The following links provide tables of default and master field lists for some standard model configurations which are characterized by the values of the `-dyn`, `-phys`, and `-chem` arguments to **configure**. The source of the information in these tables is CAM's default log file, so you can always look there for any configuration not included in the list below.

- fv, cam4, none [hist_flds_fv_cam4.html]
- fv, cam4, trop_bam [hist_flds_fv_cam4_trop_bam.html]
- fv, cam5, trop_mam3 [hist_flds_fv_cam5.html]
- fv, cam4, waccm_mozart [hist_flds_fv_cam4_waccm.html] (use_case: waccm_2000_cam4)
- fv, cam4, super_fast_llnl [hist_flds_fv_cam4_super_fast_llnl.html] (use_case: 2000_cam4_super_fast_llnl)

Chapter 5. Physics modifications via the namelist

This chapter is comprised of sections that explore how to customize various aspects of CAM's run time configuration. General instructions for building namelists using the **build-namelist** utility were given in the section called “Building the Namelist”, and details of the **build-namelist** utility are in Appendix B, *The **build-namelist** utility*.

Radiative Constituents

The atmospheric constituents which impact the calculation of radiative fluxes and heating rates are referred to as radiative constituents. A single CAM run may potentially contain multiple sources of any given constituent, for example, a prognostic version of ozone from a chemistry scheme and a prescribed version of ozone from a dataset. The radiative constituent module was designed to

- provide an explicit specification of the gas and aerosol constituents that impact the radiation calculations, and
- allow this specification to be modified via the namelist.

A detailed description of the radiative constituent module is found in the Reference Manual [../rm_tr/rm.html#rad_cnst_intro].

Putting the entire specification of the radiative constituents into the namelist results in a certain amount of complexity which is hard to avoid. This sections begins with a description of what's in the default specifications for both the **cam4** and **cam5** physics packages. Following that are some examples of how to modify the default namelist settings.

Default `rad_climate` for `cam4` physics

The `cam4` physics package uses prescribed gases (except for water vapor), and prescribed bulk aerosols. `rad_climate` is the namelist variable which holds the specification of radiatively active constituents. The default value of `rad_climate` generated by **build-namelist** is:

```
rad_climate =
 'A:Q:H2O', 'N:O2:O2', 'N:CO2:CO2', 'N:ozone:O3',
 'N:N2O:N2O', 'N:CH4:CH4', 'N:CFC11:CFC11', 'N:CFC12:CFC12',
 'N:sulf:/CSMDATA/atm/cam/physprops/sulfate_camrt_c080918.nc',
 'N:dust1:/CSMDATA/atm/cam/physprops/dust1_camrt_c080918.nc',
 'N:dust2:/CSMDATA/atm/cam/physprops/dust2_camrt_c080918.nc',
 'N:dust3:/CSMDATA/atm/cam/physprops/dust3_camrt_c080918.nc',
 'N:dust4:/CSMDATA/atm/cam/physprops/dust4_camrt_c080918.nc',
 'N:bcar1:/CSMDATA/atm/cam/physprops/bcpho_camrt_c080918.nc',
 'N:bcar2:/CSMDATA/atm/cam/physprops/bcphi_camrt_c080918.nc',
 'N:ocar1:/CSMDATA/atm/cam/physprops/ocpho_camrt_c080918.nc',
 'N:ocar2:/CSMDATA/atm/cam/physprops/ocphi_camrt_c080918.nc',
 'N:SSLTA:/CSMDATA/atm/cam/physprops/ssam_camrt_c080918.nc',
 'N:SSLTC:/CSMDATA/atm/cam/physprops/sscm_camrt_c080918.nc'
```

The `rad_climate` variable takes an array of string values. Each of the strings has three fields separated by colons. In this example the first field of each string is either an `A` or an `N`. An `A` indicates the constituent

is advected and an N indicates the constituent is not advected. Generally a non-advected constituent is one whose value is prescribed from a dataset but that's not always the case. It's also possible that a non-advected constituent is one that has been prognosed by a chemistry scheme (e.g. the cloud borne species in the modal aerosol models) or diagnosed from other prognostic species. The second field in each string is a name that is used to identify the constituent in the appropriate internal data structure (there are separate data structures for the advected and the non-advected constituents). The third field is either a name from the set of gas specie names recognized by the radiation code, or it is an absolute pathname of a dataset that contains physical and optical properties of an aerosol. This third field is how CAM distinquishes the gas from the aerosol species.

The names used for the prescribed gas species except ozone in both the cam4 and cam5 physics packages, i.e., O2, CO2, N2O, CH4, CFC11, and CFC12, are hardcoded in the module ghg_data which is responsible for setting the values of these species in the physics buffer. The name for water vapor, Q, is hardcoded in a cnst_add subroutine call made from subroutine phys_register. The name for ozone, ozone, is hardcoded in the prescribed_ozone module which is responsible for reading ozone datasets and setting the values for ozone in the physics buffer.

The names used to identify the gas species which must be provided to the cam4 radiation code are H2O, O2, CO2, O3, N2O, CH4, CFC11, and CFC12. These names are hardcoded in the module radconstants. There are no datasets associated with the gas specie names because the optical properties of the gases are handled by the radiation code directly.

The names used to identify the bulk aerosol species are hardcoded in the **build-namelist** utility and are specified to the prescribed_aero module by the namelist variable prescribed_aero_specifier as follows:

```
prescribed_aero_specifier =
 'sulf:SO4', 'bcar1:CB1', 'bcar2:CB2', 'ocar1:OC1', 'ocar2:OC2',
 'sslt1:SSLT01', 'sslt2:SSLT02', 'sslt3:SSLT03', 'sslt4:SSLT04',
 'dust1:DST01', 'dust2:DST02', 'dust3:DST03', 'dust4:DST04'
```

The first name in each of these colon separated pairs is the one the prescribed_aero module adds to the physics buffer, while the second name is the variable name in the dataset. The first names for all the species except the sea salt bins (sslt1, ..., sslt4) are the ones that appear in the rad_climate specifier. Sea salt is treated specially by repartitioning the total mass in the four bins into a coarse and an accumulation mode with the names SSLTC and SSLTA respectively. The repartitioning is done by the sslt_rebin module.

Each of the aerosol species has an associated file which contains physical and optical properties.

Default rad_climate for cam5 physics

The cam5 physics package uses the same prescribed gases as the cam4 package, but uses prognostic modal aerosols from the trop_mam3 chemistry package. The default value of rad_climate generated by **build-namelist** is:

```
rad_climate =
 'A:Q:H2O', 'N:O2:O2', 'N:CO2:CO2', 'N:ozone:O3',
 'N:N2O:N2O', 'N:CH4:CH4', 'N:CFC11:CFC11', 'N:CFC12:CFC12',
 'M:mam3_mode1:/CSMDATA/atm/cam/physprops/mam3_mode1_rrtmg_c110318.nc',
 'M:mam3_mode2:/CSMDATA/atm/cam/physprops/mam3_mode2_rrtmg_c110318.nc',
 'M:mam3_mode3:/CSMDATA/atm/cam/physprops/mam3_mode3_rrtmg_c110318.nc'
```

The gas species mass mixing ratios come from the same constituents in `cam5` as they did in `cam4` (but the radiative treatment is different since the `rrtmg` radiation package replaces `camrt`). Hence the `rad_climate` strings for the gasses are the same as they were in the `cam4` physics example.

The aerosol constituents in this `rad_climate` specification are all in the form of modes. The first field is an `M` rather than an `A` or an `N` to indicate that the aerosol constituents are modes. Roughly, the `rad_climate` variable lists the aerosol constituents whose contributions are added together to compute the total aerosol optical depth. In the case of the bulk aerosols the optical depths due to the individual aerosol species are summed to find the total aerosol optical depth. In the case of the modal aerosol model it is the modes that are summed. Hence each mode has an entry in the `rad_climate` list, along with a file that contains physical and optical properties of the mode as a whole. In the example above there are three modes identified by the names `mam3_mode1`, `mam3_mode2`, and `mam3_mode3`. These names are hardwired in the **build-namelist** utility and are only used to connect each mode with more detailed specification of the constituents that comprise it. That specification is given by the namelist variable `mode_defs` and looks as follows for the default `trop_mam3` chemistry scheme.

```
mode_defs =
 'mam3_mode1:accum:=',
   'A:num_a1:N:num_c1:num_mr:+',
   'A:so4_a1:N:so4_c1:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc:
   'A:pom_a1:N:pom_c1:p-organic:/CSMDATA/atm/cam/physprops/ocpho_rrtmg_c101112.nc:
   'A:soa_a1:N:soa_c1:s-organic:/CSMDATA/atm/cam/physprops/ocphi_rrtmg_c100508.nc:
   'A:bc_a1:N:bc_c1:black-c:/CSMDATA/atm/cam/physprops/bcpho_rrtmg_c100508.nc:+',
   'A:dst_a1:N:dst_c1:dust:/CSMDATA/atm/cam/physprops/dust4_rrtmg_c090521.nc:+',
   'A:ncl_a1:N:ncl_c1:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc',
 'mam3_mode2:aitken:=',
   'A:num_a2:N:num_c2:num_mr:+',
   'A:so4_a2:N:so4_c2:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc:
   'A:soa_a2:N:soa_c2:s-organic:/CSMDATA/atm/cam/physprops/ocphi_rrtmg_c100508.nc:
   'A:ncl_a2:N:ncl_c2:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc',
 'mam3_mode3:coarse:=',
   'A:num_a3:N:num_c3:num_mr:+',
   'A:dst_a3:N:dst_c3:dust:/CSMDATA/atm/cam/physprops/dust4_rrtmg_c090521.nc:+',
   'A:ncl_a3:N:ncl_c3:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc:+',
   'A:so4_a3:N:so4_c3:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc'
```

Similarly to the `rad_climate` variable, the `mode_defs` variable is an array of strings which provide a definition for all the modes that may be used in a single run. The modes don't all need to appear in the `rad_climate` variable; some may only be needed for diagnostic radiation calculations which will be discussed in more detail later.

There are three different types of strings in `mode_defs`:

- The initial string in each mode specification contains three fields. The first is a name that identifies the mode, the second is a name that identifies the type of the mode, and the final is the token "=".
- One string in each mode specification must contain the names for the mode number concentrations in both the interstitial and cloud borne phases.
- One or more strings in each mode specification must contain the names for the mass mixing ratios in both the interstitial and cloud borne phases of the individual constituents that comprise the mode.

The example of `mode_defs` above has been formatted in a way that makes the individual parts of each mode definition stand out. The actual output from the **build-namelist** utility is not formatted like this and is a bit harder to decipher.

What follows is an detailed explanation of the mode definitions in the example above.

- There are three modes defined, i.e., `mam3_mode1`, `mam3_mode2`, and `mam3_mode3`. The name of a mode is arbitrary, the only requirement being that the same name is used in the `rad_climate` (or `rad_diag_N`) and the `mode_defs` variables. These default mode names for `trop_mam3` are hardcoded in the **build-namelist** utility. The three modes are of type `accum` (accumulation), `aitken`, and `coarse` respectively. The names for the mode types are hardcoded in the `modal_aero_data` module.

- The second line in the definition of each mode contains the names of the number concentrations for the interstitial and cloud borne phases. Looking specifically at the definition for `mam3_mode1`, the first two fields are for the interstitial phase and specify that the name `num_a1` is an advected constituent (`A`), while the third and fourth fields are for the cloud borne phase and specify that the name `num_c1` is a non-advected constituent (`N`). The names of the number concentration constituents are hardcoded in the `modal_aero_initialize_data` module. The fifth field, `num_mr`, is a fixed token recognized by the parser of the `mode_defs` strings (in the `rad_constituents` module) as an indicator that the string contains the number concentration names. The final token in the string, a "+", signals to the parser that the definition of the current mode continues in the next string.

- The third through final strings in each mode definition contain specifications for each specie in the mode. Looking again at the definition of `mam3_mode1`, the first specie is of type `sulfate` which is indicated by the fifth field in that string. The specie type names are hardcoded in the `modal_aero_data` module. The first two fields in the string provide the name for the mass mixing ratio of the specie in the interstitial phase (`so4_a1`), and indicate that it is an advected constituent (`A`). Fields three and four specify that the name of the mass mixing ratio for the cloud borne phase is `so4_c1`, and that this is a non-advected constituent (`N`). The names of the mass mixing ratio constituents are hardcoded in the `modal_aero_initialize_data` module. The sixth field in the string is the absolute pathname of the file containing physical and optical properties of the specie. The last field in the string contains the token "+" which again indicates that the definition of the mode continues in the next string.

Example 5.1. Modify a radiatively active gas

Suppose that we wish to modify the distribution of water vapor that is seen by the radiation calculations. More specifically, consider modifying just the stratospheric part of the water vapor distribution while leaving the troposheric distribution unchanged. To modify a radiatively active gas two things must be done.

- Change the name (and possibly the type) of the constituent which is providing the mass mixing ratios to the radiation code. This is a simple modification to the `rad_climate` value.

- Make the necessary modifications to CAM to provide the new constituent mixing ratios. A likely scenario for this example would be to create a new module which is responsible for adding the modified water vapor field to the physics buffer. This module could leverage the existing `tropopause` module to determine the vertical levels where changes need to be made. It could also leverage existing modules for reading and interpolating prescribed constituents, for example the `prescribed_ozone` module. Details of how to make this type of source code modification won't be covered here.

Now suppose the source code modifications have been made and the new water vapor constituent is in the physics buffer with the name `Q_fixstrat`. The best way to modify the `rad_climate` variable is to start from a value that was generated by **build-namelist** for the configuration of interest but with the default water vapor, and then to modify that version of `rad_climate` and add the modified version to the **build-namelist** command in our run script. Note that the entire value of `rad_climate` must be specified. There is no way to just modify one individual string in the array of string values. If we are running with a default `cam5` configuration then the customized namelist would be generated by the following command.

```
$camcfg/build-namelist ... \
 -namelist "&cam ...
 rad_climate =
 'N:Q_fixstrat:H2O', 'N:O2:O2', 'N:CO2:CO2', 'N:ozone:O3',
 'N:N2O:N2O', 'N:CH4:CH4', 'N:CFC11:CFC11', 'N:CFC12:CFC12',
 'M:mam3_mode1:/CSMDATA/atm/cam/physprops/mam3_mode1_rrtmg_c110318.nc',
 'M:mam3_mode2:/CSMDATA/atm/cam/physprops/mam3_mode2_rrtmg_c110318.nc',
 'M:mam3_mode3:/CSMDATA/atm/cam/physprops/mam3_mode3_rrtmg_c110318.nc'  /"
```

The only difference between this version of `rad_climate` and the default is that the string for water vapor:

```
'A:Q:H2O'
```

has been replaced by

```
'N:Q_fixstrat:H2O'
```

In addition to specifying the new name for the constituent (`Q_fixstrat`), it was necessary to replace the `A` by an `N` since the new constituent is not advected, even though it is derived in part for the constituent `Q` which is advected.

Diagnostic radiative forcing

There are several namelist variables available for direct radiative forcing calculations in the `cam5` physics package. But note that these online calculations are enabled for RRTMG only and not for the CAM_RT radiation code used in the `cam4` and earlier physics packages. The ability to do radiative forcing calculations with CAM_RT is provided by using the offline tool PORT which is documented here [https://wiki.ucar.edu/display/port/PORT], and described in the paper Conley et al. [2013]. The PORT functionality is included in the CESM release code.

Namelist variables are available for ten radiative forcing calculations; `rad_diag_1`, ..., `rad_diag_10`. The values of these variables use the exact same format as the `rad_climate` variable. When a diagnostic calculation is requested, for example by setting the variable `rad_diag_1`, then the default history output variables for the radiative heating rates and fluxes will be output for the diagnostic calculation as well. The names of the variables for the diagnostic calculation will be distinquished from those that affect the climate simulation by appending the strings '`_d1`', ..., '`_d10`' for diagnostic calculations specified by `rad_diag_1` through `rad_diag_10` respectively.

Example 5.2. Aerosol radiative forcing

To compute the total aerosol radiative forcing we need a diagnostic calculation in which all the aerosols have been removed. To do this we start from the default setting for the `rad_climate` variable, use that as the initial setting for `rad_diag_1`, and then edit that initial setting to remove the aerosols. In the `cam5` physics this involves removing the specification of the three modes, so we end up with a setting in our **build-namelist** command that looks like this

```
$camcfg/build-namelist ... \
 -namelist "&cam ...
 rad_diag_1 =
 'A:Q:H2O', 'N:O2:O2', 'N:CO2:CO2', 'N:ozone:O3',
 'N:N2O:N2O', 'N:CH4:CH4', 'N:CFC11:CFC11', 'N:CFC12:CFC12' /"
```

Example 5.3. Black carbon radiative forcing

To compute the radiative forcing of a single aerosol specie we need a diagnostic calculation in which that specie has been removed from all modes that contain it. This is a bit more complicated that the previous example where we were able to remove entire modes from the value of `rad_diag_1`. Removing species from modes requires us to create new mode definitions. Using black carbon as a specific example, we see from the default definitions of the `trop_mam3` modes (the section called "Default `rad_climate` for `cam5` physics") that black carbon is only contained in `mam3_mode1`. The best way to create the definition of a new mode which doesn't contain black carbon is to copy the definition of `mam3_mode1`, change its name, and remove the black carbon from the definition. Then use this new mode in place of `mam3_mode1` in the specifier for `rad_diag_1`. Below is an outline of our **build-namelist** command with just the `mode_defs` and `rad_diag_1` variables listed.

```
$camcfg/build-namelist ... \
 -namelist "&cam ...
mode_defs =
 'mam3_mode1:accum:=',
   'A:num_a1:N:num_c1:num_mr:+',
   'A:so4_a1:N:so4_c1:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc:
   'A:pom_a1:N:pom_c1:p-organic:/CSMDATA/atm/cam/physprops/ocpho_rrtmg_c101112.nc:
   'A:soa_a1:N:soa_c1:s-organic:/CSMDATA/atm/cam/physprops/ocphi_rrtmg_c100508.nc:
   'A:bc_a1:N:bc_c1:black-c:/CSMDATA/atm/cam/physprops/bcpho_rrtmg_c100508.nc:+',
   'A:dst_a1:N:dst_c1:dust:/CSMDATA/atm/cam/physprops/dust4_rrtmg_c090521.nc:+',
   'A:ncl_a1:N:ncl_c1:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc',
 'mam3_mode2:aitken:=',
   'A:num_a2:N:num_c2:num_mr:+',
   'A:so4_a2:N:so4_c2:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc:
   'A:soa_a2:N:soa_c2:s-organic:/CSMDATA/atm/cam/physprops/ocphi_rrtmg_c100508.nc:
   'A:ncl_a2:N:ncl_c2:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc',
 'mam3_mode3:coarse:=',
   'A:num_a3:N:num_c3:num_mr:+',
   'A:dst_a3:N:dst_c3:dust:/CSMDATA/atm/cam/physprops/dust4_rrtmg_c090521.nc:+',
   'A:ncl_a3:N:ncl_c3:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc:+',
   'A:so4_a3:N:so4_c3:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc'
 'mam3_mode1_noBC:accum:=',
   'A:num_a1:N:num_c1:num_mr:+',
   'A:so4_a1:N:so4_c1:sulfate:/CSMDATA/atm/cam/physprops/sulfate_rrtmg_c080918.nc:
   'A:pom_a1:N:pom_c1:p-organic:/CSMDATA/atm/cam/physprops/ocpho_rrtmg_c101112.nc:
   'A:soa_a1:N:soa_c1:s-organic:/CSMDATA/atm/cam/physprops/ocphi_rrtmg_c100508.nc:
   'A:dst_a1:N:dst_c1:dust:/CSMDATA/atm/cam/physprops/dust4_rrtmg_c090521.nc:+',
   'A:ncl_a1:N:ncl_c1:seasalt:/CSMDATA/atm/cam/physprops/ssam_rrtmg_c100508.nc'

 rad_diag_1 =
 'A:Q:H2O', 'N:O2:O2', 'N:CO2:CO2', 'N:ozone:O3',
 'N:N2O:N2O', 'N:CH4:CH4', 'N:CFC11:CFC11', 'N:CFC12:CFC12',
 'M:mam3_mode1_noBC:/CSMDATA/atm/cam/physprops/mam3_mode1_rrtmg_c110318.nc',
 'M:mam3_mode2:/CSMDATA/atm/cam/physprops/mam3_mode2_rrtmg_c110318.nc',
 'M:mam3_mode3:/CSMDATA/atm/cam/physprops/mam3_mode3_rrtmg_c110318.nc'  /"
```

Note that we just appended the new mode definition, `mam3_mode1_noBC`, to the end of the modes used in the climate calculation, and then used that mode in place of `mam3_mode1` in the `rad_diag_1` value.

Appendix A. The configure utility

The **configure** utility provides a flexible way to specify any configuration of CAM. The best way to communicate to another user how you built CAM is to simply supply them with the **configure** commandline that was used (along with the source code version).

configure has two distinct operating modes which correspond to the two distinct ways of building CAM, i.e., either using the CESM scripts, or using CAM standalone scripts. By default **configure** runs in the mode used by the standalone scripts. In this mode **configure** is responsible for setting the filepaths and CPP macros needed to build not only CAM, but all the components of the standalone configuration including the land, sea ice, data ocean, and driver. In the mode used when building CAM from the CESM scripts **configure** is only responsible for setting the filepaths and CPP macros needed to build a library containing just the CAM component.

When configuring a build of standalone CAM, **configure** produces the files `Filepath` and `Makefile`. In addition, a configuration cache file (`config_cache.xml` by default) is written which contains the values of all the configuration parameters set by **configure**. The files produced by **configure** are written to the directory where CAM will be built, which by default is the directory from which **configure** is executed, but can be specified to be elsewhere (see the `-cam_bld` option).

When configuring CAM for a build using the CESM scripts, **configure** doesn't write a Makefile, but instead writes a file `CCSM_cppdefs` which is used by the CESM Makefile. Also, the `Filepath` file only contains paths for the CAM component.

In both modes **configure** is responsible for setting the correct filepaths and CPP macros to produce the desired configuration of CAM's dynamical core, physics parameterizations and chemistry scheme. The options that are involved in making these choices are described in the section called “CAM configuration” below. The subsequent sections describe options used by the CAM standalone scripts.

configure will optionally perform tests to validate that the Fortran compiler is operational and Fortran 90 compliant, and that the linker can resolve references to required external libraries (NetCDF and possibly MPI). These tests will point out problems with the user environment in a way that is much easier to understand than looking at the output from a failed build of CAM. We strongly recommend that the first time CAM is built on any new machine, **configure** should be invoked to execute these tests (see the `-test` option).

How configure is called from the CESM scripts

The CESM scripts access CAM's **configure** via the script `$CAM_ROOT/models/atm/cam/bld/cam.buildnml.csh`. The `cam.buildnml.csh` script acts as the interface between the CESM scripts and CAM's **configure** and **build-namelist** utilities.

Arguments to configure

All configuration options can be specified using command line arguments to **configure** and this is the recommended practice. Options specified via command line arguments take precedence over options specified any other way.

At the next level of precedence a few options can be specified by setting environment variables. And finally, at the lowest precedence, many options have hard-coded defaults. Most of these are located in the files `$CAM_ROOT/models/atm/cam/bld/config_files/defaults_*.xml`. A few that depend on the values of other options are set by logic contained in the **configure** script (a Perl script). The hard-coded defaults are designed to produce the standard production configurations of CAM.

The configure script allows the user to specify compile time options such as model resolution, dynamical core type, additional compiler flags, and many other aspects. The user can type **`configure --help`** for a complete list of available options.

The options may all be specified with either one or two leading dashes, e.g., `-help` or `--help`. The few options that can be expressed as single letter switches may not be clumped, e.g., `-h -s -v` may *NOT* be expressed as `-hsv`. When multiple options are listed separated by a vertical bar either version may be used.

CAM configuration

These options will have an effect whether running CAM as part of CESM or running in a CAM standalone mode:

`-[no]age_of_air_trcs`	Switch on [off] age of air tracers. Default: on for waccm_phys, otherwise off.
`-carma` **`<name>`**	Build CAM with specified CARMA microphysics model [`none` | `bc_strat` | `cirrus` | `dust` | `meteor_smoke` | `pmc` | `sea_salt` | `sulfate` | `test_detrain` | `test_growth` | `test_passive` | `test_radiative` | `test_swelling` | `test_tracers`]. Default: `none`.
`-chem` **`<name>`**	Build CAM with specified prognostic chemistry package [`waccm_mozart` | `waccm_mozart_sulfur` | `waccm_ghg` | `trop_mozart` | `trop_mozart_mam3` | `trop_mozart_soa` | `trop_ghg` | `trop_bam` | `trop_mam3` | `trop_mam7` | `super_fast_llnl` | `super_fast_llnl_mam3` | `trop_strat_soa` | `trop_strat_mam3` | `trop_strat_mam7` | `none`]. Default: `trop_mam3` if the physics package is `cam5`, otherwise default is `none`.
`-clubb_sgs`	Switch to turn on the CLUBB_SGS package. Default: Off.
`-co2_cycle`	This option is usually used with the `-ccsm_seq` option as part of the configuration for running biogeochemistry (BGC) compsets. It modifies the CAM configuration by increasing the number of advected constituents by 4. Default: not set.
`-comp_intf` [`mct` | `esmf`]	Specify the component interfaces Default: `mct`.
`-cosp`	Enable the COSP simulator package. Default: not set.
`-cppdefs` **`<string>`**	A string of user specified CPP defines appended to Makefile defaults. E.g. `-`

	`cppdefs '-DVAR1 -DVAR2'`. Note that a string containing whitespace will need to be quoted.
`-dyn` [`eul` | `sld` | `fv` | `se`]	Build CAM with specified dynamical core. Default: `fv`.
`-edit_chem_mech`	Invokes `CAMCHEM_EDITOR` to allow the user to edit the chemistry mechanism file.
`-hgrid` **<name>**	Specify horizontal grid. For spectral grids use `nlatxnlon` where `nlat` and `nlon` are the number of latitude and longitude grid points respectively in the global Gaussian grid (e.g., `64x128`). For FV grids use `dlatxdlon` where `dlat` and `dlon` are the grid cell size in degrees for latitude and longitude respectively (e.g., `1.9x2.5`). For SE grids (cubed sphere) use `neNnpM` where `N` is the number of elements on an edge of the cube, and `M` is the number of Gauss points on the edge of an element (e.g., `ne30np4`).
`-microphys` [`mg1` | `mg1.5` | `rk`]	Specify the microphysics package. Default: `mg1` if the physics package is `cam5`, otherwise `rk`.
`-nadv` **<n>**	Set total number of advected species to **<n>**. If `-nadv` is set to a larger number than is required by the selected physics and chemistry schemes, then the remainder will automatically be used for test tracers (N.B. the namelist variable `tracers_flag` must be set to `.true.` to enable the test tracer code.) Default: set to the number required by the selected physics and chemistry schemes.
`-nadv_tt` **<n>**	Set number of advected test tracers to **<n>**. Setting the number of test tracers explicitly with this option allows **build-namelist** to automatically enable the test tracer code by setting the `tracers_flag` namelist variable. Default: 0.
`-nlev` **<n>**	Set number of vertical layers to **<n>**. Default: `30` if the physics package is `cam5`, `ideal`, or `adiabatic`. `26` if the physics package is `cam4`. `66` if the chemistry package is `waccm_*`. `81` if the `-waccmx` is used.
`-offline_dyn`	Switch enables the use of offline driver for FV dycore. Default: not set.
`-pbl` [`uw` | `hb` | `hbr` | `clubb_sgs`]	PBL package. Default: `uw` if the physics package is `cam5`; `clubb_sgs` if the `-clubb_sgs` switch is set; otherwise `hb`.

`-pcols` **<n>**	Set maximum number of grid columns in a chunk to **<n>**. Default: 16.
`-pergro`	Switch enables building CAM for perturbation growth tests. Only valid with `cam3` and `cam4` physics packages.
`-phys` [`cam3` | `cam4` | `cam5` | `ideal` | `adiabatic`]	Physics package. Default: `cam5`.
`-prog_species` **<list>**	Comma separated list of prognostic mozart species packages. Currently available: `DST,SSLT,SO4,GHG,OC,BC,CARBON16`
`-psubcols` **<n>**	Set maximum number of subcolumns in a grid column to **<n>**. Default: 1.
`-rad` [`rrtmg` | `camrt`]	Radiation package. Default: `rrtmg` if the physics package is `cam5`, otherwise `camrt`.
`-usr_mech_infile` **<name>**	Pathname of the user supplied chemistry mechanism file.
`-waccm_phys`	Switch enables the use of WACCM physics in any chemistry configuration. Default: Off unless one of the waccm chemistry options is chosen then it's automatically turned on.
`-waccmx`	Build CAM/WACCM with WACCM upper Thermosphere/Ionosphere extended package.

SCAM configuration

`-camiop`	Configure CAM to generate an IOP file that can be used to drive SCAM. This switch only works with the Eulerian dycore.
`-scam`	Compiles model in single column mode. Only works with Eulerian dycore.

CAM parallelization

`-[no]smp`	Switch on [off] SMP parallelism (OpenMP). This option can be used when building a model that doesn't contain CICE. It allows building an executable whose thread count can be set at run time.
`-[no]spmd`	Switch on [off] SPMD parallelism (MPI). This option can be used when building a model that doesn't contain CICE. It allows building an executable whose task count can be set at run time.

CAM parallelization when running standalone with CICE

`-ntasks` **<n>**	This option must be used to specify SPMD parallelism when the CICE component is present. **<n>** is the number of MPI tasks. Setting ntasks > 0 implies `-spmd`. Use `-nospmd` to turn off linking with an MPI library. To configure for pure MPI specify "`-ntasks N -nosmp`". ntasks is

used by CICE to determine default grid decompositions which must be specified at build time.

`-nthreads` **<n>** This option must be used to specify SMP parallelism when the CICE component is present. **<n>** is the number of OpenMP threads per process. Setting nthreads > 0 implies `-smp`. Use `-nosmp` to turn off compilation of OMP directives. For pure OpenMP set "`-nthreads N -nospmd`". nthreads is used by CICE to determine default grid decomposition which must be specified at build time.

NOTE: When CAM is running standalone with CICE the default CICE decomposition is determined from the values of the `-ntasks` and `-nthreads` arguments. The user also has the ability to explicitly set the CICE decomposition using the following four arguments. If any of these arguments is set then *ALL FOUR* must be set.

`-cice_bsizex` **<n>** CICE block size in longitude dimension. This size must evenly divide the number of longitude points in the global grid.

`-cice_bsizey` **<n>** CICE block size in latitude dimension. This size must evenly divide the number of latitude points in the global grid.

`-cice_maxblocks` **<n>** Maximum number of CICE blocks per process.

`-cice_decomptype` **<name>** CICE decomposition type [`cartesian` | `spacecurve` | `roundrobin`].

General options

`-cache` **<name>** Name of output cache file. Default: `config_cache.xml`.

`-cachedir` **<dir>** Name of directory where output cache file is written. Default: CAM build directory.

`-ccsm_seq` Switch to specify that CAM is being built from within the CESM scripts. This produces Filepath and CCSM_cppdefs files that contains only the paths and CPP macros needed to build a library for the CAM component.

`-defaults` **<name>** Specify a configuration file which will be used to supply defaults instead of one of the `config_files/defaults_*.xml` files. This file is used to specify model configuration parameters only. Parameters relating to the build which are system dependent will be ignored.

`-help | -h` Print usage to STDOUT.

`-silent | -s` Turns on silent mode - only fatal messages printed to STDOUT.

`-test` Switch on testing of Fortran compiler and linking to external libraries.

`-verbose | -v` Turn on verbose echoing of settings made by configure.

`-version` Echo the repository tag name used to check out this CAM source tree.

Surface components

Options for surface components used in standalone CAM mode:

`-ice` [`cice` | `sice`]	Specify the sea ice component. Default: `cice`.
`-lnd` [`clm` | `slnd`]	Specify the land component. Default: `clm`.
`-ocn` [`docn` | `socn` | `dom` | `aquaplanet`]	Specify ocean component. If set to `aquaplanet` then the stub ice (`sice`) and stubb land (`slnd`) components are implied. Default: `docn`.
`-rof` [`rtm` | `srof`]	Specify the river runoff component. Default: `rtm`.

CAM standalone build

Options for building CAM via standalone scripts:

`-cam_bld` **<dir>**	Directory where CAM will be built. This is where configure will write the output files it generates (Makefile, Filepath, etc...). Default: ./
`-cam_exe` **<name>**	Name of the CAM executable. Default: `cam`.
`-cam_exedir` **<dir>**	Directory where CAM executable will be created. Default: CAM build directory.
`-cc` **<name>**	User specified C compiler. Default: Depends on the OS and the Fortran compiler.
`-cflags` **<string>**	A string of user specified C compiler options appended to the default options set in Makefile.
`-debug`	Switch to turn on building CAM with compiler options for debugging. The specific options are compiler dependent. These flags are set in the `Makefile.in` template file.
`-esmf_libdir` **<dir>**	Directory containing ESMF library and the `esmf.mk` file. If this option is specified then the external ESMF library will be used in place of the ESMF-WRF time manager code which is provided in the CESM source distribution.
`-fc` **<name>**	User specified Fortran compiler. Default: Depends on the OS and whether MPI is enabled.
`-fc_type` [`pgi` | `lahey` | `intel` | `pathscale` | `gnu` | `xlf`]	Type of the Fortran compiler. This argument is used in conjunction with the `-fc` argument

	when the name of the fortran compiler refers to a wrapper script (e.g., mpif90 or ftn). In this case the user needs to specify the type of Fortran compiler that is being invoked by the wrapper script. Default: Depends on the name of the Fortran compiler.
`-fflags` **<string>**	A string of user specified Fortran compiler options appended to the default options set in the Makefile. See `-fopt` to override optimization flags.
`-fopt` **<string>**	A string of user specified Fortran compiler optimization flags. Overrides Makefile defaults.
`-gmake` **<name>**	Name of the GNU make program on your system. Supply the absolute pathname if the program is not in your path (or fix your path). This is only needed by **configure** for running tests via the `-test` option.
`-lapack_libdir` **<dir>**	Directory containing LAPACK library.
`-ldflags` **<string>**	A string of user specified load options. Appended to Makefile defaults.
`-linker` **<name>**	User specified linker. Default: use the Fortran compiler.
`-mpi_inc` **<dir>**	Directory containing MPI include files.
`-mpi_lib` **<dir>**	Directory containing MPI library.
`-nc_inc` **<dir>**	Directory containing NetCDF include files.
`-nc_lib` **<dir>**	Directory containing NetCDF library.
`-nc_mod` **<dir>**	Directory containing NetCDF module files.
`-pnc_inc` **<dir>**	Directory containing PnetCDF include files.
`-pnc_lib` **<dir>**	Directory containing PnetCDF library.
`-rad_driver`	Build CAM with the offline radiation driver. This produces an executable that can only be used for offline radiation calculations.
`-target_os` **<name>**	Override the OS setting for cross platform compilation from the following list [`aix`|`irix`|`linux`| `bgl`|`bgp`]. Default: OS on which configure is executed as defined by the Perl $OSNAME variable.
`-usr_src` **<dir1>[,<dir2>[,<dir3>[...]]]**	Directories containing user source code. Note that these directories will also be searched for modified versions of the files needed by

the **build-namelist** script, e.g., the namelist definition and use case files.

Environment variables recognized by configure

The following environment variables are recognized by **configure**. Note that the command line arguments for specifying this information always takes precedence over the environment variables.

`CASEROOT`	Directory where a CESM case is set up. This is only used when building from the CESM scripts to add the SourceMods directory for CAM to the `Filepath` file.
`ESMF_LIBDIR`	Directory containing the ESMF library.
`INC_MPI`	Directory containing the MPI include files.
`INC_NETCDF`	Directory containing the NetCDF include files.
`INC_PNETCDF`	Directory containing the PnetCDF include files.
`LAPACK_LIBDIR`	Directory containing the LAPACK library.
`LIB_MPI`	Directory containing the MPI library.
`LIB_NETCDF`	Directory containing the NetCDF library.
`LIB_PNETCDF`	Directory containing the PnetCDF library.
`MCT_LIBDIR`	Directory containing the MCT libraries.
`MOD_NETCDF`	Directory containing the NetCDF module files.

Appendix B. The build-namelist utility

The **build-namelist** utility builds namelists (and on occasion other types of input files) which specify run-time details for CAM and the components it's running with in standalone mode. When executed from the CESM scripts it only produces a namelist file for the CAM component (in the file `atm_in`), and a namelist file for control of dry deposition which is shared by CAM and CLM (in the file `drv_flds_in`).

The task of constructing a correct namelist has become extremely complex due to the large number of configurations supported by CAM. Editing namelists by hand is an extremely fragile process due to the number of variables that need to be set, and to the many interdependencies among them. *We stronly discourage editing namelists by hand.* All customizations of the CAM namelist are possible by making use of the **build-namelist** command line options.

Some of the important features of **build-namelist** are:

- All valid namelist variables are known to **build-namelist**. So an invalid variable specified by the user (supplied either by the `-infile` or `-namelist` options) will cause **build-namelist** to fail with an error message telling which namelist variable is invalid. This is a big improvement over a runtime failure caused by an invalid variable which typically gives no hint as to which variable caused the problem.

- In addition to knowing all valid variable names and their types, **build-namelist** also knows which namelist group each variable belongs to. This means that the user only needs to specify variable names to **build-namelist** and not the group names. The `-infile` and `-namelist` options still require valid namelist syntax as input, but the group name is ignored. So all variables can be put in a single group with an arbitrary name, for example, "&xxx ... /" where "xxx" is the namelist group name.

- Since **build-namelist** knows all namelist variables specified by the user it is able to do consistency checking. In general however, **build-namelist** assumes that the user is the expert and will not override a user specification unless there is a major inconsistency, for example if variables have been set to use parameterizations which can not be run at the same time.

- All configurations have namelist variables that must be specified, and **build-namelist** has a mechanism to provide default values for these variables. When an appropriate default value cannot be found then **build-namelist** will fail with an informative message.

- When running a configuration for the first time there are often many input datasets that may not be in the local input data directory. In order to facilitate getting the required datasets **build-namelist** has an option, `-test`, that can be used to produce a complete list of required datasets and report status of whether or not they are present in the local directory. This list can then be used to obtain the needed datasets from the CESM SVN input data repository.

One required input for **build-namelist** is a configuration cache file produced by a previous invocation of **configure** (`config_cache.xml` by default). **build-namelist** looks at this file to determine the features of the CAM executable, such as the dynamical core and horizontal resolution, that affect the default specifications for namelist variables. The default values themselves are specified in the file `$CAM_ROOT/models/atm/cam/bld/namelist_files/namelist_defaults_cam.xml`, and in the use case files located in the directory `$CAM_ROOT/models/atm/cam/bld/namelist_files/use_cases/`.

The other required input for **build-namelist** is the root directory for the input datasets. This is required since nearly all input files must be specified using absolute filepaths, but the defaults are stored as filepaths which are relative to the root directory. It is expected that the actual location of the root directory is something that will be resolved at runtime. The way this is done is to either specify it using the `-csmdata` argument, or to set the environment variable `CSMDATA`.

The methods for setting the values of namelist variables, listed from highest to lowest precedence, are:

1. using specific command-line options, e.g., `-case` and `-runtype`,
2. using the `-namelist` option,
3. setting values in a file specified by `-infile`,
4. specifying a `-use_case` option,
5. setting values in the namelist defaults file.

The first four of these methods for specifying namelist variables are the ones available to the user without requiring code modification. Any namelist variable recognized by CAM can be modified using method 2 or 3. The final two methods represent defaults that are hard coded as part of the code base.

Options to build-namelist

To get a list of all available options, type **`build-namelist --help`**. Available options are also listed just below.

The following options may all be specified with either one or two leading dashes, e.g., `-help` or `--help`. The few options that can be expressed as single letter switches may not be clumped, e.g., `-h -s -v` may *NOT* be expressed as `-hsv`. When multiple options are listed separated by a vertical bar either version may be used.

`-case` **`<name>`**
: Case identifier up to 80 characters. This value is used to set the case_name variable in the driver namelist. Default: `camrun`

`-cice_nl` **`<namelist>`**
: Specify namelist settings for CICE directly on the commandline by supplying a string containing FORTRAN namelist syntax, e.g., **`-cice_nl "&ice histfreq=1 /"`**. This namelist will be passed to the invocation of the CICE build-namelist via its `-namelist` argument.

`-config` **`<filepath>`**
: Read the specified configuration cache file to determine the configuration of the CAM executable. Default: `config_cache.xml`.

`-config_cice` **`<filepath>`**
: Filepath of the CICE config_cache file. This filepath is passed to the invocation of the CICE **build-namelist**. Only specify this to override the default filepath which was set when the CICE **configure** was invoked by the CAM **configure**.

`-csmdata` **`<dir>`**
: Root directory of CESM input data. Can also be set by using the `CSMDATA` environment variable.

`-dir` **`<dir>`**
: Directory where output namelist files for each component will be written, i.e., `atm_in`,

	`drv_in`, `ice_in`, `lnd_in` and `ocn_in`. Default: current working directory.
`-help | -h`	Print usage to STDOUT.
`-ignore_ic_date`	Ignore the date attribute of the initial condition files when determining the default.
`-ignore_ic_year`	Ignore just the year part of the date attribute of the initial condition files when determining the default.
`-infile` **<filepath>**	Specify a file containing namelists to read values from.
`-inputdata` **<filepath>**	Writes out a list of pathnames for required input datasets to the specified file.
`-namelist` **<namelist>**	Specify namelist settings directly on the commandline by supplying a string containing FORTRAN namelist syntax, e.g., **-namelist "&atm stop_option='ndays' stop_n=10 /"**
`-ntasks` **<n>**	Specify the number of MPI tasks to be used by the run. This is only used to set a default decomposition for the FV dycore, i.e., the npr_yz variable.
`-runtype` [`startup`|`continue`|`branch`]	Type of simulation. Default: `startup`.
`-silent | -s`	Turns on silent mode - only fatal messages issued.
`-test`	Enable checking that input datasets exist on local filesystem. This is also a convenient way to generate a list of the required input datasets for a model run.
`-use_case` **<name>**	Specify a use case.
`-verbose | -v`	Turn on verbose echoing of informational messages.
`-version`	Echo the source code repository tag name used to check out this CAM distribution.

Environment variables used by build-namelist

The environment variables recognized by **build-namelist** are presented below.

`CSMDATA`	Root directory of CESM input data. Note that the commandline argument `-csmdata` takes precedence over the environment variable.

`OMP_NUM_THREADS`

If values of the specific variables that set the thread count for each component, i.e., **`atm_nthreads`**, **`cpl_nthreads`**, **`ice_nthreads`**, **`lnd_nthreads`**, or **`ocn_nthreads`**, are set via the `-namelist`, or `-infile` options, then these values have highest precedence. The `OMP_NUM_THREADS` environment variable has next highest precedence for setting any of the component specific thread count variables. Lowest precedence for setting these variables is the value of `nthreads` from the configure cache file.

CAM Namelist variables

A searchable (or browsable) page containing all CAM namelist variables is here [/cgi-bin/eaton/namelist/nldef2html-cam5_3].

References

[Conley et al. [2013]] Conley, A.J., J.-F. Lamarque, F. Vitt, W.D. Collins, and J. Kiehl. PORT, a CESM tool for the diagnosis of radiative forcing , Geoscientific Model Development, 6, 469-476, doi:10.5194/gmd-6-469-2013 [http://dx.doi.org/10.5194/gmd-6-469-2013] , 2013.

[Hurrell et al. [2008]] Hurrell, J.W., James J. Hack, Dennis Shea, Julie M. Caron, and James Rosinski. A New Sea Surface Temperature and Sea Ice Boundary Dataset for the Community Atmosphere Model , Journal of Climate, 21, 5145-5153, doi:10.1175/2008JCLI2292.1 [http://dx.doi.org/10.1175/2008JCLI2292.1] , 2008.

[Taylor et al. [2000]] Taylor, K.E., David Williamson, and Francis Zwiers. The Sea Surface Temperature and Sea-Ice Concentration Boundary Conditions for AMIP II Simulations , PCMDI Report No. 60, UCRL-JC-125597, http://www-pcmdi.llnl.gov/publications/pdf/60.pdf [http://www-pcmdi.llnl.gov/publications/pdf/60.pdf] , 2000.