Comprehensive list of standards

>>Terms of Use
>>Go BACK to previous page.

These conventions cover some of the more detailed aspects of CLM code design. It is intended to compliment and extend the information found on the previous page. If you have questions please contact us at CLM-CMT@cgd.ucar.edu.

General Guidelines

1. We support Fortran 2003 language features with a few exceptions. Information regarding compiler support for Fortran 2003 features may be found at the fortran wiki . The three Fortran 2003 feature that we can't support due to lack of compiler implementation are:
    a. finalization
    b. parameterized derived types
    c. derived type I/O
2. Use **implicit none**. All variables must be explicitly typed.
3. **Don't repeat yourself**. Duplicated code is difficult to maintain. If you find yourself using copy and paste more than once, create a new function or subroutine that encapsulates the desired functionality.
4. Try to implement subroutine and function interface with argument lists of 9 or less. If you have lists longer than this, break up the functionality into multiple methods and/or use derived types for higher level interfaces. Try to have argument lists short enough that you can make unit-tests for them.
5. If you have argument lists greater than 8 -- you must call such routines using the keyword-pair form for all arguments (but see 3 above).

Performance, debugging and optimization

1. Don't mix up C/C++ and Fortran memory access order; it will slow your code down drastically. If you have question contact a CLM-CMT.
2. Debug with a debugger (totalview, gdb, idb, pgdbg, etc...) it will save you a lot of time finding errors. For an example see this CISL example
    a. Adding Write statements to your code and dumping variable values also can work very well in finding problems.
3. Be aware of the memory footprint of additional code. Try to keep increases in memory use low. Use Valgrind or similar tools often during your development process to track and monitor memory increases, changes, leaks, etc....
4. Performance
    a. Regarding vectorization. If you feel your code performance is suffering there are some ideas here under the Fortran section.
    b. How you access derived type members can change performance. See for example some ideas regarding derived types and a discussion on alignment

Modularity

1. Data types, data and methods should be encapsulated within a module if they are used in that module. If data types, data and/or methods are used outside of the module they are declared in, they must be made publicly available.
2. Data should also be accessible at the most private level possible. In other words:
    a. If a variable is only used in one subroutine it should be local to that subroutine
    b. If a variable is only local to one module then it is declared private to that module.
    c. Public data for modules should be used sparingly.
3. Only data required for output to history files should be in clmtype.F90.
4. A separate restart module/subroutine called from CLM restFileMod.F90 should be used to facilitate reading/writing of information to restart files.
5. In general data initialized in clmtype should have a separate module/subroutine to do so.
6. The number of variables added to clmtype needs to be limited.

Guidelines for Fortran statements

1. Fortran2003 deprecated features will not be accepted in new code (equivalence, common blocks, block data, etc.).
2. Use statements – should nearly always use the **only** construct to define what variables are being used from a given module. The only case when this is not to be done, is when the list of variables is so long to make this unmanageable taking up more than 6 continue lines (for example with use of the clmtype module).
3. The "intent" attribute of subroutine input/output variables is used. Argument lists should ALWAYS use the INTENT attribute (try to have data be either IN or OUT and avoid INOUT if possible/reasonable).
4. Try to limit the use of the "use statement" to other modules to subroutines/functions and read-only parameters.
5. Data that is read-only, or output from a subroutine/function should be passed down using argument lists.
6. Modules explicitly declare public subroutines/functions at the module level.
7. By default all modules should have variables/methods/types declared to be private.
8. Print statements are not allowed, use write instead.
9. Stop statements will not be accepted in new code – use the endrun subroutine part of the CLM abortutils module.
10. Debugging statements should be removed prior to check in. If there is a need to leave them in, then they will be controlled by a logical or integer control variable.

11. Don't hard-code "magic numbers". For example, hard-coding a variable to "5" in order to make sure a certain path of an if branch is taken. Use the parameter statement at the top of the subroutine to set such variables.
12. Use symbolic logical operators (<, >, == etc.) rather than word (.lt., .gt. .eq.).

Guidelines for working within the CLM codebase

1. We no longer accept new code with preprocessor macros (CPP tokens).
2. Namelist items should be defined in the XML files relevant to clm4_5 and clm4_0:
    a. models/lnd/clm/bld/namelist_files/namelist_definition_clm4_5.xml
    b. models/lnd/clm/bld/namelist_files/namelist_definition_clm4_0.xml
3. Type declarations make use of the shr_kinds_mod module.
4. All write statements are to iulog from clm_varctl.F90 or to another appropriate logical unit.
5. Write statements should be within "if (masterproc)" if statement unless output by all tasks is really desired (such as for error statements).
6. Unit numbers are to be controlled using the CLM module fileutils.F90.
7. The module spmdMod should be used to get MPI information such as masterproc, iam etc.
8. The ncdio_pio module should be used to read/write/define NetCDF input/output data.
9. Modules clmvarcon or shr_const_mod should be used to define parameters in common with CLM and/or CESM.
10. Module clm_time_manager should be used to get time/date information. In particular length of day, month, or year information should be queried from clm_time_manager.
11. CLM parameters should be retrieved from the clm_varcon, clm_varpar and clm_varctl modules.
12. If writing is done in an initialization subroutine iulog should be flushed using shr_sys_flush at the end of the subroutine.
13. Evaluate new history fields being added. If they are not critical leave out of the code base; if they are only needed rarely, make them "inactive" by default.

Guidelines for adding new datasets

1. Input datasets or raw datasets used to create input datasets (such as those that require the mksurfdata tool) need to be imported into the Subversion inputdata repository by one of the CLM-CMTs. CLM-CMTs have the necessary permissions to do this and can also stage data on DIN_LOC_ROOT for NCAR internal machines.
2. Input datasets should have names that describe what they are, contain their resolution, and a have creation date at the end (i.e. _cYYMMDD). Data should be in NetCDF format ending in a .nc extension.
3. All input datasets will be entered into the CLM XML database in the models/lnd/clm/bld/namelist_files directory.
4. CF-metadata conventions should be followed for the files.
5. The attributes of "units" and "long_name" should be set for all variables. Variables with missing data should have set the "missing_value" and "_FillValue" attributes.
6. The attributes "Conventions", "source" and/or "history" should be set as global attributes on the file (references, institution, title and date should be considered as well).
7. Filenames should not be hardcoded into CLM code, they should be entered in by a namelist.
8. File sizes or number of time samples or records, should not be hardcoded into CLM code, but read in from the file.