Adding New Namelist Items to CLM

What are namelist fields and why would you add them?

Namelist fields are the way to add run-time control variables for the model. They have default values for reasonable settings that are good for most model simulations, but allow the user to override those values. They are the way that we control all run-time behavior of the model. We have a vast array of dozens of namelist settings that control both model physics changing your model results as well as changing behavior without changing results (such as changing the history fields output, or the length of the simulation). The reason to add a namelist field is to add the ability to control a specific aspect of the model behavior at run-time. A way to think about name lists are your list of “preferences” for how CLM should run, like your “preferences” for an application such as a web-browser. Those preferences have reasonable defaults, but you are allowed to change them in order to get the behavior you want out of the application. In the same way the namelist fields have reasonable defaults, but you can override any of the default values as you see fit.

Namelist fields controlled by env_run

Some of the namelist fields are set by “env_run.xml” control variables. Adding a new “env_run” variable at this level, requires working with scripts and code outside of the clm tree, and is more work than we will go into this short description. The assumption we make here is that the variable you are adding is for specific control over CLM itself, rather than the overall CESM driver.

How NOT to add a namelist item?

Many variables in CLM are made public and put into global modules and then actually read in the controlMod.F90 module in models/lnd/clm/src/clm4_5 /main. The problem with this is that these variables get put into several CLM specific modules and are then made public so they can be changed anywhere. This makes it difficult to debug issues and makes CLM specific modules very complex and hard to work with. When a package is removed from CLM it then becomes difficult to remove all these dependencies, because package specific variables are then strewn around CLM global data, subroutines and modules. Keeping data modular by keeping it as local to the subroutines/modules that use it makes it both easier to work with CLM and more robust. As such we are NOT going to tell you how to add an item to controlMod.F90, that method is deprecated, and the following methodology will hopefully gradually replace controlMod over time.

OK, what do I need to do to add a new namelist item then?

Here are all the possible steps needed to add a new namelist item. In most cases the steps up to step 4 will already be done. In some cases a namelist read will already be done for the module you are adding your variable to, which makes the process even simpler.

1. Identify which module this variable needs to be added to.
2. Add the data as local to the module
3. Add an initialization subroutine for that module.
4. Add the call to the initialization subroutine in clm_initializeMod.F90
5. Add a namelist read in the initialization subroutine
6. Broadcast the values and write out the settings
7. Add the namelist/namelist values to the XML database.

Now going through each step in turn. Again in many cases some of these steps would already be done for the module you are adding a namelist variable to.

1.) Identify which module this variable needs to be added to.

The first step is to figure out which subroutines/module(s) the variable will be used in and how to make it as local as possible to those. In the best case the variable will ONLY be used in one subroutine, and hence will just be local to that module. In other cases the variable might be part of an overall package and used by more than one module in the given package. In this case the package should have a module that controls it’s overall behavior and the variable should be added to that level. For example, the “CN” package has the module “CNEcosystemDynMod.F90” that calls the different modules that are part of it. Variables that impact more than one CN package module can be put into this module. If such an overall package module does NOT exist, or this variable impacts wider CLM behavior, contact the CLM-CMT on how to add the variable.

2.) Add the data as local to the module

In most cases the new data will be private module data, and read in with one of the subroutines in the module (the initialization subroutine). To do this you declare the data at the top of the module and give it the descriptor "private", so it will be local to the given module.

3.) Add an initialization subroutine for that module (won't need to do if there already is one)

In most cases modules will already have an initialization subroutine for it so this step won't need to be done. If there isn't one for the given module, declare the interface name at the top of the module as public, and add the subroutine in the module itself. Initially you just need a subroutine declaration and end subroutine. And in general there would be no arguments passed into or out of the subroutine.

4.) Add the call to the initialization subroutine in clm_initializeMod.F90 (won't need to do if there already is one)

If there already was an initialization subroutine, this step is already done. If not a call to the initialization subroutine you made in step 3. needs to be done from clm_iniitializeMod. If the module you are adding to is a submodule of another larger package, the call should be done from the initialization subroutine for that package.

5.) Add a namelist read in the initialization subroutine (won't need to do if there already is one)

If the module already had an initialization subroutine that already reads in a namelist you merely need to add your new variables to the namelist by adding them to the namelist declaration. If not you'll need to both add the namelist declaration to the initialization subroutine as well as reading in the namelist. There are some CLM global modules and global data that help you do this. The CLM utility "clm_nlUtilsMod" module contains a subroutine to find the beginning of a namelist group and "fileutils" has subroutines to give you a file logical unit to use to read your data in on. "controlMod" has the name of the CLM namelist file, "spmdMod" identifies if this is the master processor to actually do the read on, and clm_varctl has the logical unit to write log output to.

```
  use clm_varctl     , only: iulog                  ! Logical unit to write log output to

  use spmdMod        , only: masterproc             ! If this is the processor to read the namelist in on

  use fileutils      , only: getavu, relavu         ! Get and release an available logical unit to read data to

  use controlMod     , only: NLFilename             ! The name of the CLM namelist file "lnd_in"

   use clm_nlUtilsMod   , only : find_nlgroup_name  ! Find the input namelist group in the file
```

6.) Broadcast the values and write out the settings

After the namelist is read, the values need to be broadcast to each processor using "shr_mpi_bcast" which is a subroutine call to the "csm_share" utility "shr_mpi_mod.F90". Then it's useful to write out the settings to the log file for the user to see what is happening.

7.) Add the namelist/namelist values to the XML database.

The namelist items need to be added to the namelist_definition_clm4_5.xml file in models/lnd/clm/bld/namelist_files. If there is a well-defined list of valid values, that field should be given for them. If this is a new namelist, the namelist itself will need to be added to the CLM script "build-namelist" in models/lnd /clm/bld.

For example, here is the definition of the namelist value "popdensmapalgo" to the XML database which is part of the "popd_streams" namelist.

```
<entry id="popdensmapalgo" type="char*256" category="datasets"

       group="popd_streams" valid_values="bilinear,nn,nnoni,nnonj,spval,copy" >

Mapping method from human population density input file to the model resolution

    bilinear = bilinear interpolation

    nn       = nearest neighbor

    nnoni    = nearest neighbor on the "i" (longitude) axis

    nnonj    = nearest neighbor on the "j" (latitude) axis

    spval    = set to special value

    copy     = copy using the same indices

</entry>
```

The place where a new namelist is added into CLM build-namelist is in the "write_output_files" subroutine where the "groups" are listed such as in the following line where namelist groups are listed. There is also a subroutine to setup any logic needed for that namelist in the subroutine "setup_logic_popd_streams".

```
    @groups = qw(clm_inparm ndepdyn_nml popd_streams light_streams clm_hydrology1_inparm
clm_soilhydrology_inparm);
```

If you need to define default values for the new namelist item see the section Adding New Resolutions or New Files to the build-namelist Database in the CLM User's Guide about how to do this. In general you would NOT need to do this, unless you need a default value to be set in every namelist. In general when adding a new control switch setting the default in the code and allowing the user to override the default setting if they know about the setting is fine.

What is an example of an existing namelist read?

A good example of an existing namelist item read is for CNFireMod.F90 in models/lnd/clm/src/clm4_5/biogeochem.

```
 ! Define the namelist variables
 integer            :: stream_year_first_popdens   ! first year...

….

 ! Define the namelist with the other
 ! Set the default values for namelist

 stream_year_first_popdens  = 1       ! first year in stream to use

 ….

 ! Read popd_streams namelist

 if (masterproc) then

    nu_nml = getavu()

    open( nu_nml, file=trim(NLFilename), status='old', iostat=nml_error )

    call find_nlgroup_name(nu_nml, 'popd_streams', status=nml_error)

    if (nml_error == 0) then

       read(nu_nml, nml=popd_streams,iostat=nml_error)

       if (nml_error /= 0) then

          call endrun(subname // ':: ERROR reading popd_streams namelist')

       end if

    end if

    close(nu_nml)

    call relavu( nu_nml )

 end if

 ! Broadcast the variables read in...

 call shr_mpi_bcast(stream_year_first_popdens, mpicom)
```

```
...

   ! Write out what the settings are..

   if (masterproc) then

      write(iulog,*) ' '

      write(iulog,*) 'popdens_streams settings:'

….

    end if
```

Add the call to the initialization subroutine in clm_iniitializeMod.F90

In this case CNFireInit is part of CN initialization and hence the call is added to CNEcosystemDynInit. Furthermore, CNFireInit is actually split into two subroutines hdminit and lndfm_init both of which are private to CNFireMod.F90 and called from the public initialization for them CNFireInit.