CCPP Training

College Park, MD, March 12-13, 2019

What Makes a Parameterization CCPP-compliant?

Grant Firl

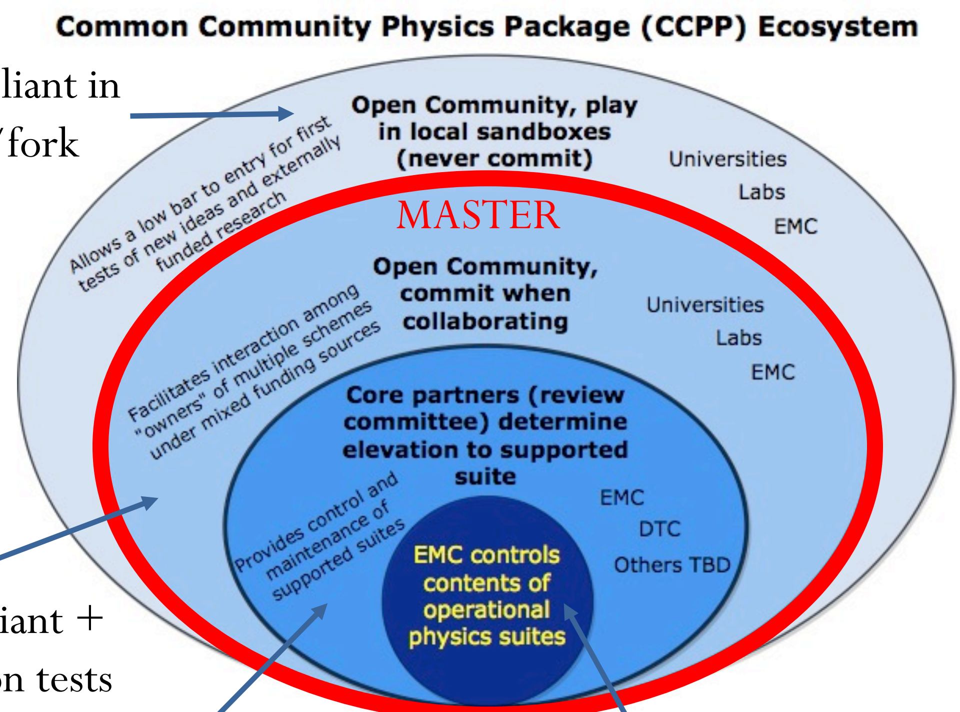
Global Model Test Bed

Outline of Talk

- CCPP-compliant vs officially supported
- Argument metadata
- Required subroutines
- Error handling
- Other coding rules

Two Tiers of Acceptance

- CCPP-compliancy
  - lowest bar (mechanistic)
- Supported CCPP
  - highest bar (governance-related)
    - performance and memory optimized
    - full scientific/technical documentation
    - merit
    - governance process



CCPP-compliant in user branch/fork

CCPP-compliant + pass regression tests

CCPP-compliant + fully supported

CCPP-compliant + fully supported + operational

Physics Scheme Argument Metadata

- Metadata for variables needed by physics is KEY to how the CCPP works
- **Variables provided by the host are matched to those needed by physics based on comparing metadata**
- Metadata are provided in commented tables that precede a subroutine's code (and do double-duty as documentation)
- Special formatting is required for:
  - Doxygen (documentation) parsing
  - CCPP framework script parsing

Physics Scheme Argument Metadata

- Current metadata attributes
  - local_name – what a variable is called in the local subroutine
  - **standard_name** – how a variable is referred to internally to the CCPP
    - must be unique within the CCPP
    - based on CF conventions where possible
  - long_name – more verbose description of variable
  - units – use standard unit abbreviations and exponents immediately follow (`m2 s-2`)
  - rank – variable dimensionality
  - type – Fortran intrinsic type or derived type name
  - kind – specifies precision or length
  - intent – in, out, inout
  - optional

Physics Scheme Argument Metadata

- Current formatting (align for readability)

must match subroutine name

mind the format!

```
!! \section arg_table_scheme_X_run Argument Table
!! | local_name | standard_name                               | long_name                                | units   | rank | type      |    kind   | intent | optional |
!! |------------|---------------------------------------------|------------------------------------------|---------|------|-----------|-----------|--------|----------|
!! | im         | horizontal_loop_extent                      | horizontal loop extent                   | count   |    0 | integer   |           | in     | F        |
!! | levs       | vertical_dimension                          | vertical layer dimension                 | count   |    0 | integer   |           | in     | F        |
!! | vdftra     | vertically_diffused_tracer_concentration    | tracer concentration diffused by PBL scheme | kg kg-1 |    3 | real      | kind_phys | inout  | F        |
!!
```

required for Doxygen formatting

- The order of arguments in the table does not have to match the order of actual arguments in the subroutine, but it is preferred.

Physics Scheme Argument Metadata

- Changes in the pipeline:
  - metadata will reside in external file with the same root name with a different file extension (.meta)
  - rank will be replaced by actual dimensions:
    - (standard_name_of_dim1, standard_name_of_dim2)
  - python config-file-like format (similar to INI for MS Win)
  - a converter for the new format will be provided

Physics Scheme Argument Metadata

- List of standard names currently being used in the host
  - A list of available standard names and an example of naming conventions can be found in **ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_${HOST}.pdf,** where **${HOST}** is the name of the host model. Running the CCPP prebuild script (see Chapter 3: Running ccpp_prebuild.py) will generate a LaTeX source file that can be compiled to produce a PDF file with all variables defined by the host model and requested by the physics schemes. The script will also indicate if additional variables need to be added.
- All variable information (units, rank, index ordering) must match the specifications on the host model side, but sub-slices can be used/added in the host model. For example, in GFS_typedefs.F90, tendencies can be split so they can be used in the necessary physics scheme:
  - dt3dt(:,:,1) = cumulative_change_in_temperature_due_to_longwave_radiation
  - dt3dt(:,:,2) = cumulative_change_in_temperature_due_to_shortwave_radiation

Required Subroutines

- _init, _run, _finalize subroutines
- consistency between module name and subroutine names
- _init and _finalize subroutines run during ccpp_physics_initialize / ccpp_physics_finalize calls
- idempotent
- empty schemes don't need metadata tables

```
module scheme
  implicit none
  private
  public :: scheme_init,scheme_run,scheme_finalize
  contains
  subroutine scheme_init()
  end subroutine scheme_init
  subroutine scheme_finalize()
  end subroutine scheme_finalize
  subroutine scheme_run()
  end subroutine scheme_run
end module scheme
```

Scheme—specific Interstitial

- pre- and post- scheme-specific interstitial code may be placed in the same source file as different modules (also need _init, _run, and _finalize)

```
module scheme_pre
  implicit none
  private
  public :: scheme_pre_init, scheme_pre_run, &
            scheme_pre_finalize
  contains
  subroutine scheme_pre_init()
  end subroutine scheme_pre_init
  subroutine scheme_pre_finalize()
  end subroutine scheme_pre_finalize
  subroutine scheme_pre_run()
  end subroutine scheme_pre_run
 end module scheme_pre
```

Parameterization Drivers

- Although discouraged, it may be necessary to add a driver layer on top of some schemes. In this case the driver is the CCPP-compliant "scheme".
  - to preserve schemes distributed outside of CCPP (e.g., Thompson MP from WRF)
  - (temporary) unit conversions and array transformations (vertically flip)

Error Handling

- Schemes should make use of CCPP error-handling variables and not stop/abort/print errors within

```
!! | errflg | ccpp_error_flag     | error flag for error handling      | flag | 0 | integer   |         | none |F |
!! | errmsg | ccpp_error_message  | error message for error handling   | none | 0 | character | len=512 | none |F |
```

- `ccpp_error_flag` and `ccpp_error_message` must be arguments (intent OUT)
- In the event of an error, assign a meaningful error message to **errmsg** and set **errflg** to a value other than 0:

```
write (errmsg, '(*(a))') 'Logic error in scheme xyz: …'
errflg = 1
return
```

Developmental Testbed Center

Other Coding Rules

- All external information required by the scheme must be passed in via the argument list.
  - No ‘use EXTERNAL_MODULE’ for passing in data
  - Physical constants should go through the argument list
- Code must comply to modern Fortran standards (Fortran 90/95/2003).
- Use labeled **end** statements for modules, subroutines and functions, example:
  - **module scheme_template** → **end module scheme_template**.
- Use **implicit none**.
- All **intent(out)** variables must be set inside the subroutine, including the mandatory variables **errflg** and **errmsg**. [Watch out for partially set **intent(out)** variables.]
- No permanent state of decomposition-dependent host model data inside the module, i.e. no variables that contain domain-dependent data using the **save** attribute.
- No **goto** statements.
- No **common** blocks.

Additional coding rules are listed under the *Coding Standards* section of the NOAA NGGPS Overarching System team document on Code, Data, and Documentation Management for NEMS Modeling Applications and Suites (available at https://docs.google.com/document/u/1/d/1bjnyJpJ7T3XeW3zCnhRLTL5a3m4_3XIAUeThUPWD9Tg/edit#heading=h.97v79689onyd).

Parallel Programming

- Shared-memory (OpenMP) parallelization inside a scheme is allowed with the restriction that the number of OpenMP threads to use is obtained from the host model as an **intent(in)** argument in the argument list
- MPI communication is allowed in the **_init** and **_finalize** phase for the purpose of computing, reading or writing scheme-specific data that is independent of the host model's data decomposition.
- MPI calls are restricted to global communication at this time, no point-to-point; the MPI communicator is also an input argument to the scheme
- Calls to MPI and OpenMP functions, and the import of the MPI and OpenMP libraries, must be guarded by C preprocessor directives.

Wrap Up

- CCPP-compliancy vs supported schemes
- Scheme argument variable metadata
  - what is included and how to write it
- Required subroutines and scheme-specific interstitial
- Error handling
- Other coding rules, parameterization drivers, and parallel programming