Common Community Physics Package (CCPP)

Developers' Guide
v2.0

August 2018

Dom Heinzeller, Ligia Bernardet
CIRES/CU at NOAA/ESRL Global Systems Division and Developmental Testbed Center

Laurie Carson, Grant Firl
National Center for Atmospheric Research and Developmental Testbed Center

Developmental Testbed Center

Acknowledgement

If significant help was provided via the GMTB helpdesk for work resulting in a publication, please acknowledge the Developmental Testbed Center GMTB Team.

For referencing this document please use:

Heinzeller, D., L. Bernardet, L. Carson, and G. Firl, 2018. Common Community Physics Package (CCPP) v2.0 Developers' Guide. 19pp. Available at https://dtcenter.org/gmtb/users/ccpp/docs/CCPP-DevGuide-v2.pdf

Contents

Preface

Meaning of typographic changes and symbols

Table 1 describes the type changes and symbols used in this book.

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.bashrc` Use `ls -a` to list all files. `host$ You have mail!.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`host$` **`su`**
`AaBbCc123`	Command line placeholder: replace with a real name or value	To delete a file, type `rm` *`filename`*

Table 1: Typographic Conventions

1 Introduction

The Common Community Physics Package (CCPP) is designed to facilitate the implementation of physics innovations in state-of-the-art atmospheric models, the use of various models to develop physics, and the acceleration of transition of physics innovations to operational NOAA models. The CCPP consists of two separate software packages, the pool of CCPP-compliant physics schemes (`ccpp-physics`) and the framework (driver) that connects the physics schemes with a host model (`ccpp-framework`).

The connection between the host model and the physics schemes through the CCPP framework is realized with caps on both sides as illustrated in Fig. 3.1 in Chapter 3. While the caps to the individual physics schemes are auto-generated, the cap that connects the framework (Physics Driver) to the host model must be created manually. The CCPP framework generates a large fraction of code that can be included in the host model cap to facilitate this process. For more information about the CCPP design and implementation, see the CCPP Design Overview at `https://dtcenter.org/gmtb/users/ccpp/docs/`.

This document serves two purposes, namely to describe the technical work of writing a CCPP-compliant physics scheme and adding it to the pool of CCPP physics schemes (Chapter 2), and to explain in detail the process of connecting an atmospheric model (host model) with the CCPP (Chapter 3). For further information and an example for integrating CCPP with a host model, the reader is referred to the GMTB Single Column Model (SCM) User and Technical Guide v2.1 available at `https://dtcenter.org/gmtb/users/ccpp/docs`.

At the time of writing, the CCPP is supported for use with the GMTB Single Column Model (SCM). Support for use of CCPP with the experimental version of NCEP's Global Forecast System (GFS) that employs the Finite-Volume Cubed-Sphere dynamical core (FV3GFS) is available as an internal release for the developers. A public release of FV3GFS with CCPP is planned for early 2019.

The GMTB welcomes contributions to CCPP, whether those are bug fixes, improvements to existing parameterizations, or new parameterizations. There are two aspects of adding innovations to the CCPP: technical and programmatic. This Developer's Guide explains how to make parameterizations technically compliant with the CCPP. Acceptance in the master branch of the CCPP repositories, and elevation of a parameterization to supported status, depends on a set of scientific and technical criteria that are under development as part of the incipient CCPP Governance. Contributions can be made in form of git pull requests to the development repositories. Before initiating a major development for the CCPP please contact GMTB at `gmtb-help@ucar.edu` to create an integration and transition plan. For further information, see the Developer's Corner for CCPP at `https://dtcenter.org/gmtb/users/ccpp/developers/index.php`. Note that while the pool of CCPP physics and the CCPP framework are managed by the Global Model Test Bed (GMTB) and governed jointly with partners (e.g., NCAR), the code governance for the host models lies with their respective organizations. Therefore, inclusion of CCPP within those models should be brought up to their governing bodies.

2 CCPP-compliant physics schemes

2.1 Writing a CCPP-compliant physics scheme

The rules for writing a CCPP-compliant scheme are summarized in the following. Listing 2.1 contains a Fortran template for a CCPP-compliant scheme, which can also be found in `ccpp-framework/doc/DevelopersGuide/scheme_template.F90`.

General rules:

- Scheme must be in its own module (module name = scheme name) and must have three entry points (subroutines) starting with the name of the module: module `scheme_template` → subroutines `scheme_template_{init,finalize,run}`. The `_init` and `_finalize` routines are run automatically when the CCPP physics are initialized. These routines may be called more than once, depending on the host model's parallelization strategy, and as such must be idempotent (that is, multiple calls must not change the answer).
- Empty schemes (e. g. `scheme_template_init` in listing 2.1) need no argument table.
- Schemes in use require an argument table as below, the order of arguments in the table must be the same as in the argument list of the subroutine.
- An argument table must precede the subroutine, and must start with

  ```
  !> \section arg_table_subroutine_name Argument Table
  ```

  and end with a line containing only

  ```
  !!
  ```

- All external information required by the scheme must be passed in via the argument list, i.e. no external modules (except if defined in the Fortran standards 95–2003).
- If the width of an argument table exceeds 250 characters, wrap the argument table in CPP preprocessor directives:

  ```
  #if 0
  !> \section arg_table_scheme_template_run Argument Table
  ...
  !!
  #endif
  ```

- Module names, scheme names and subroutine names are case sensitive.
- For better readability, it is suggested to align the columns in the metadata table.

Input/output variable (argument) rules:

- Variables available for CCPP physics schemes are identified by their unique `standard_name`. While an effort is made to comply with existing `standard_name`

definitions of the CF conventions (http://cfconventions.org), additional names are introduced by CCPP (see below for further information).
- A `standard_name` cannot be assigned to more than one local variable (`local_name`). The `local_name` of a variable can be chosen freely and does not have to match the `local_name` in the host model.
- All information (units, rank) must match the specifications on the host model side.
- The two mandatory variables that every scheme must accept as `intent(out)` arguments are `errmsg` and `errflg` (see also coding rules).
- At present, only two types of variable definitions are supported by the CCPP framework:
  - Standard Fortran variables (`character`, `integer`, `logical`, `real`). For `character` variables, the length should be specified as $*$. All others can have a `kind` attribute of a kind type defined by the host model.
  - Derived data types (DDTs). While the use of DDTs is discouraged in general, some use cases may justify their application (e.g. DDTs for chemistry that contain tracer arrays, information on whether tracers are advected, ...).
- If a scheme is to make use of CCPP's subcycling capability in the runtime suite definition file (SDF; see also GMTB Single Column Model Technical Guide v2.1, chapter 6.1.3, https://dtcenter.org/gmtb/users/ccpp/docs), the loop counter can be obtained from CCPP as an `intent(in)` variable (see Listings 3.1 and 3.2 for a mandatory list of variables that are provided by the CCPP framework and/or the host model for this and other purposes).

Coding rules:

- 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 initialized properly inside the subroutine.
- No permanent state of decomposition-dependent host model data inside the module.
- No `goto` statements.
- Errors are handled by the host model using the two mandatory arguments `errmsg` and `errflg`. In the event of an error, assign a meaningful error message to `errmsg` and set `errflg` to a value other than 0.
- Schemes are not allowed to abort/stop the program.
- Schemes are not allowed to perform I/O operations (except for reading lookup tables or other information needed to initialize the scheme)
- Line lengths of 120 characters are suggested for better readibility (exception: CCPP metadata argument tables).

Parallel programming rules:

- 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 through the subroutine's argument table (Listings 3.1 and 3.2).
- 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. An example is the initial read of a lookup table

of aerosol properties by one or more MPI processes which is then broadcasted to all processes. Several restrictions apply:
  – Reading and writing of data must be implemented in a scalable way to perform efficiently from a few to millions of tasks.
  – The MPI communicator to use must be received from the host model through the subroutine's argument table (Listings 3.1 and 3.2).
  – The use of MPI is restricted to global communications, for example `barrier`, `broadcast`, `gather`, `scatter`, `reduce`.
- Calls to MPI and OpenMP functions, and the import of the MPI and OpenMP libraries, must be guarded by CPP preprocessor directives as illustrated in the following listing. OpenMP pragmas can be inserted without CPP guards, since they are ignored by the compiler if the OpenMP compiler flag is omitted.

```
...

#ifdef MPI
   use mpi
#endif
#ifdef OPENMP
   use omp_lib
#endif

...

#ifdef MPI
   call MPI_BARRIER(mpicomm, ierr)
#endif

#ifdef OPENMP
   me = OMP_GET_THREAD_NUM()
#else
   me = 0
#endif
```

- For Fortran coarrays, consult with the GMTB helpdesk (gmtb-help@ucar.edu).

Scientific Documentation rules:

- Technically, scientific documentation is not needed for a parameterization to work with the CCPP. However, inclusion of inline scientific documentation is highly recommended and necessary before a parameterization is submitted for inclusion in the CCPP.
- Scientific documentation for CCPP parameterizations should be inline within the Fortran code using markups according to the Doxygen software. Reviewing the documentation for CCPP v2.0 parameterizations is a good way of getting started in writing documentation for a new scheme.
- The CCPP Scientific Documentation can be converted to html format (see https://dtcenter.org/gmtb/users/ccpp/docs/sci_doc_v2/.
- For precise instructions on creating the scientific documentation, contact the GMTB helpdesk at gmtb-help@ucar.edu.

Listing 2.1: Fortran template for a CCPP-compliant scheme

```
module scheme_template

  contains

  subroutine scheme_template_init ()
  end subroutine scheme_template_init

  subroutine scheme_template_finalize()
  end subroutine scheme_template_finalize

!> \section arg_table_scheme_template_run Argument Table
!! | local_name | standard_name | long_name          | units | rank | type      | kind  | intent | optional |
!! |------------|---------------|--------------------|-------|------|-----------|-------|--------|----------|
!! | errmsg     | error_message | CCPP error message | none  |    0 | character | len=* | out    | F        |
!! | errflg     | error_flag    | CCPP error flag    | flag  |    0 | integer   |       | out    | F        |
!!
  subroutine scheme_template_run (errmsg, errflg)

     implicit none

     !--- arguments
     ! add your arguments here
     character(len=*), intent(out)   :: errmsg
     integer,          intent(out)   :: errflg

     !--- local variables
     ! add your local variables here

     continue

     !--- initialize CCPP error handling variables
     errmsg = ''
     errflg = 0

     !--- initialize intent(out) variables
     ! initialize all intent(out) variables here

     !--- actual code
     ! add your code here

     ! in case of errors, set errflg to a value != 0,
     ! assign a meaningfull message to errmsg and return

     return

  end subroutine scheme_template_run

end module scheme_template
```

2.2 Adding a new scheme to the CCPP pool

This section describes briefly how to add a new scheme to the CCPP pool and use it with a host model that already supports the CCPP.

1. Identify the required variables for your target host model: for a list of variables available for host model *XYZ* (currently `SCM` and `FV3`), see `ccpp-framework/doc/DevelopersGuide/CCPP_VARIABLES_XYZ.pdf`. Contact the GMTB helpdesk at gmtb-help@ucar.edu if you need additional variables that you believe should be provided by the host model or as part of a pre-/post-scheme (interstitial scheme) instead of being calculated from existing variables inside your scheme.
2. Identify if your new scheme requires additional interstitial code that must be run before/after the scheme and that cannot be part of the scheme itself, for example because of dependencies on other schemes and/or the order the scheme is run in the suite definition file. As of now, interstitial schemes should be created in cooperation with the GMTB helpdesk.
3. Follow the guidelines outlined in the previous section to make your scheme CCPP-compliant. Make sure to use an uppercase suffix `.F90` to enable CPP preprocessing.
4. Locate the CCPP prebuild configuration files for the target host model, for example:

```
ccpp/framework/scripts/ccpp_prebuild_config_FV3.py # for GFDL FV3
ccpp-framework/scripts/ccpp_prebuild_config_SCM.py # FOR GMTB SCM
```

5. Add the new scheme to the list of schemes using the same path as the existing schemes:

```
SCHEME_FILES = [
    ...
    '../some_relative_path/existing_scheme.F90',
    '../some_relative_path/new_scheme.F90',
    ...
    ]
```

6. If the new scheme uses optional arguments, add information on which ones to use further down in the configuration file. See existing entries and documentation in the configuration file for the possible options:

```
OPTIONAL_ARGUMENTS = {
    'SCHEME_NAME' : {
        'SCHEME_NAME_run' : [
            # list of all optional arguments in use for this model,
                by standard_name
            ],
            # instead of list [...], can also say 'all' or 'none'
        },
    }
```

7. Place new scheme in the same location as existing schemes in the CCPP directory structure, e. g. `../some_relative_path/new_scheme.F90`.
8. Edit the runtime suite definition file and add the new scheme at the place it should be run. SDFs are located in

```
ccpp/framework/suites/suite_FV3_GFS_2017_updated*.xml # FV3
ccpp-framework/suites/suite_SCM_GFS_2017_updated*.xml # SCM
```

9. Done. Note that no further modifications of the build system are required, since the CCPP framework will auto-generate the necessary makefiles that allow the host model to compile the scheme.

Note: Making a scheme CCPP-compliant is a necessary step for acceptance of the scheme in the pool of supported CCPP physics schemes, but does not guarantee it. Acceptance is subject to approval by a Governance committee and depends on scientific innovation, demonstrated added value, and compliance with the above rules. The criteria for acceptance of innovations into the CCPP is under development. For further information, please contact the GMTB helpdesk at `gmtb-help@ucar.edu`.

3 Integrating CCPP with a host model

This chapter describes the process of connecting a host model with the pool of CCPP physics schemes through the CCPP framework. This work can be split into several distinct steps outlined in the following sections.

3.1 Checking variable requirements on host model side

The first step consists of making sure that the necessary variables for running the CCPP physics schemes are provided by the host model. A list of all variables required for the current pool of physics can be found in `ccpp{-,/}framework/doc/DevelopersGuide/CCPP_VARIABLES_XYZ.pdf (XYZ: SCM, FV3)`. While most of the variable requirements come from the CCPP physics schemes, a small number of variables are required for correct operation of the CCPP and for compliance with its standards. These variables are described in Listings 3.1 and 3.2. In case a required variable (that is not mandatory for CCPP) is not provided by the host model, there are several options:

- If a particular variable is only required by schemes in the pool that will not get used, these schemes can be commented out in the ccpp prebuild config (see Sect. 2.2).
- If a variable can be calculated from existing variables in the model, an interstitial scheme (usually called *`scheme_name_pre`*) can be created that calculates the missing variable. However, the memory for this variable must be allocated on the host model side (i. e. the variable must be defined but not initialized in the host model). Another interstitial scheme (usually called *`scheme_name_post`*) might be required to update variables used by the host model with the results from the new scheme. At present, adding interstitial schemes should be done in cooperation with the GMTB Help Desk (`gmtb-help@ucar.edu`).
- In some cases, the declaration and calculation of the missing variable can be placed entirely inside the host model. Please consult with the GMTB Help Desk.

At present, only two types of variable definitions are supported by the CCPP framework:

- Standard Fortran variables (`character`, `integer`, `logical`, `real`) defined in a module or in the main program. For `character` variables, a fixed length is required. All others can have a `kind` attribute of a kind type defined by the host model.
- Derived data types (DDTs) defined in a module or the main program. While the use of derived data types as arguments to physics schemes in general is discouraged (see Sect. 2.1), it is perfectly acceptable for the host model to define the variables

Listing 3.1: Mandatory variables that are provided by the CCPP framework (and must not be defined by the host model)

```
!! | local_name | standard_name      | long_name                            | units | rank | type      | kind    | intent | optional |
!! |------------|--------------------|--------------------------------------|-------|------|-----------|---------|--------|----------|
!! | 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        |
!! | loop_cnt   | ccpp_loop_counter  | loop counter for subcycling loops    | index |    0 | integer   |         | none   | F        |
!!
```

Listing 3.2: Mandatory variables that must be provided by the host model (local name is not fixed)

```
!! | local_name | standard_name      | long_name                            | units | rank | type      | kind    | intent | optional |
!! |------------|--------------------|--------------------------------------|-------|------|-----------|---------|--------|----------|
!! | mpirank    | mpi_rank           | current MPI rank                     | index |    0 | integer   |         | none   | F        |
!! | mpiroot    | mpi_root           | master MPI rank                      | index |    0 | integer   |         | none   | F        |
!! | mpicomm    | mpi_comm           | MPI communicator                     | index |    0 | integer   |         | none   | F        |
!! | mpisize    | mpi_size           | number of MPI tasks in communicator  | count |    0 | integer   |         | none   | F        |
!! | nthreads   | omp_threads        | number of threads for use by physics | count |    0 | integer   |         | none   | F        |
```

requested by physics schemes as components of DDTs and pass these components to CCPP by using the correct `local_name` (see Listing 3.3 for an example).

With the CCPP, it is possible to not only refer to components of derived types, but also to slices of arrays in the metadata table as long as these are contiguous in memory (see Listing 3.3 in the following section for an example).

3.2 Adding metadata variable tables for the host model

To establish the link between host model variables and physics scheme variables, the host model must provide metadata tables similar to those presented in Sect. 2.1. The host model can have multiple metadata tables or just one. For each variable required by the pool of CCPP physics schemes, one and only one entry must exist on the host model side. The connection between a variable in the host model and in the physics scheme is made through its `standard_name`.

The following requirements must be met when defining variables in the host model metadata tables (see also listing 3.3 for examples of host model metadata tables).

- The `standard_name` must match that of the target variable in the physics scheme.
- The type, kind, shape and size of the variable (as defined in the host model Fortran code) must match that of the target variable.
- The attributes `units`, `rank`, `type` and `kind` in the host model metadata table must match those in the physics scheme table.
- The attributes `optional` and `intent` must be set to `F` and `none`, respectively.
- The `local_name` of the variable must be set to the name the host model cap (see Sect. 3.3) uses to refer to the variable.
- The name of the metadata table must match the name of the module or program in which the variable is defined, or the name of the derived data type if the variable is a component of this type.
- Metadata tables describing module variables must be placed inside the module.
- Metadata tables describing components of derived data types must be placed immediately before the type definition.

3.3 Writing a host model cap for the CCPP

The purpose of the host model cap is to abstract away the communication between the host model and the CCPP physics schemes. While CCPP calls can be placed directly inside the host model code, it is recommended to separate the cap in its own module for clarity and simplicity. The host model cap is responsible for:

Allocating memory for variables needed by physics. This is only required if the variables are not allocated by the host model, for example for interstitial variables used exclusively for communication between the physics schemes.

Listing 3.3: Example metadata table for a host model

```
  module example_vardefs

    implicit none

!> \section arg_table_example_vardefs
!! | local_name | standard_name | long_name | units | rank | type      | kind   | intent | optional |
!! |------------|---------------|-----------|-------|------|-----------|--------|--------|----------|
!! | ex_int     | example_int   | ex. int   | none  |    0 | integer   |        | none   | F        |
!! | ex_real1   | example_real1 | ex. real  | m     |    2 | real      | kind=8 | none   | F        |
!! | errmsg     | error_message | err. msg. | none  |    0 | character | len=64 | none   | F        |
!! | errflg     | error_flag    | err. flg. | flag  |    0 | logical   |        | none   | F        |
!!

    integer, parameter            :: r15 = selected_real_kind(15)
    integer                       :: ex_int
    real(kind=8), dimension(:,:) :: ex_real1
    character(len=64)             :: errmsg
    logical                       :: errflg

! Derived data types

!> \section arg_table_example_ddt
!! | local_name | standard_name | long_name | units | rank | type      | kind   | intent | optional |
!! |------------|---------------|-----------|-------|------|-----------|--------|--------|----------|
!! | ext%l      | example_flag  | ex. flag  | flag  |    0 | logical   |        | none   | F        |
!! | ext%r      | example_real3 | ex. real  | kg    |    2 | real      | r15    | none   | F        |
!! | ext%r(:,1) | example_slice | ex. slice | kg    |    1 | real      | r15    | none   | F        |
!!
    type example_ddt
      logical               :: l
      real, dimension(:,:) :: r
    end type example_ddt

    type(example_ddt) :: ext

  end module example_vardefs
```

Allocating the `cdata` structure. The `cdata` structure handles the data exchange between the host model and the physics schemes and must be defined in the host model cap or another suitable location in the host model. The `cdata` variable must be persistent in memory. Note that `cdata` is not restricted to being a scalar but can be a multi-dimensional array, depending on the needs of the host model. For example, a model that uses a 1-dimensional array of blocks for better cache-reuse may require `cdata` to be a 1-dimensional array of the same size. Another example of a multi-dimensional array of `cdata` is in the GMTB SCM, which uses a 1-dimensional `cdata` array for N independent columns.

Calling the suite initialization subroutine. The suite initialization subroutine takes two arguments, the name of the runtime suite definition file (of type `character`) and the name of the `cdata` variable that must be allocated at this point. *Note.* The suite initialization routine `ccpp_init` parses the suite definition file and initializes the state of the suite and its schemes. This process must be repeated for every element of a multi-dimensional `cdata`. For performance reasons, it is possible to avoid repeated reads of the suite definition file and to have a single state of the suite shared between the elements of `cdata`. This is a developmental feature and has implications on the physics initialization. Host model developers interested in this feature should contact the GMTB Help Desk (gmtb-help@ucar.edu).

Populating the `cdata` structure. Each variable required by the physics schemes must be added to the `cdata` structure – or to each element of a multi-dimensional `cdata` – on the host model side. This is an automated task and accomplished by inserting a preprocessor directive

```
#include ccpp_modules.inc
```

at the top of the cap (before `implicit none`) to load the required modules (e. g. module `example_vardefs` in listing 3.3), and a second preprocessor directive

```
#include ccpp_fields.inc
```

after the `cdata` variable and the variables required by the physics schemes are allocated.
Note. The CCPP framework supports splitting physics schemes into different sets that are used in different parts of the host model. An example therefore is the separation between slow and fast physics processes for the GFDL microphysics implemented in FV3GFS: while the slow physics are called as part of the usual model physics, the fast physics are integrated in the dynamical core. The separation of physics into different sets is part of the CCPP prebuild configuration (see Sect. 3.4), which allows to create multiple include files (e.g. `ccpp_fields_slow_physics.inc` and `ccpp_fields_fast_physics.inc` that can be used by different `cdata` structures in different parts of the model). Please contact the GMTB Help Desk (gmtb-help@ucar.edu) if you would like to use this feature.

Providing interfaces to call CCPP for the host model. The cap must provide functions or subroutines that can be called at the appropriate places in the host model time integration loop and that internally call `ccpp_init`, `ccpp_physics_init`, `ccpp_physics_run`, `ccpp_physics_finalize` and `ccpp_finalize`, and handle any errors returned.

Listing 3.4 contains a simple template of a host model cap for CCPP, which can also be found in `ccpp-framework/doc/DevelopersGuide/host_cap_template.F90`.

Listing 3.4: Fortran template for a CCPP host model cap

```fortran
module example_ccpp_host_cap

  use ccpp_api, only: ccpp_t, ccpp_field_add, ccpp_init, ccpp_finalize, &
                      ccpp_physics_init, ccpp_physics_run, ccpp_physics_finalize
  use iso_c_binding, only: c_loc
! Include auto-generated list of modules for ccpp
#include "ccpp_modules.inc"

  implicit none

! CCPP data structure
  type(ccpp_t), save, target :: cdata

  public :: physics_init, physics_run, physics_finalize

contains

  subroutine physics_init(ccpp_suite_name)
    character(len=*), intent(in) :: ccpp_suite_name
    integer :: ierr
    ierr = 0

    ! Initialize the CCPP framework, parse SDF
    call ccpp_init(ccpp_suite_name, cdata, ierr=ierr)
    if (ierr/=0) then
      write(*,'(a)') "An error occurred in ccpp_init"
      stop
    end if
! Include auto-generated list of calls to ccpp_field_add
#include "ccpp_fields.inc"
    ! Initialize CCPP physics (run all _init routines)
    call ccpp_physics_init(cdata, ierr=ierr)
    ! error handling as above

  end subroutine physics_init

  subroutine physics_run(group, scheme)
    ! Optional arguments group and scheme can be used
    ! to run a group of schemes or an individual scheme
    ! defined in the SDF. Otherwise, run entire suite.
    character(len=*), optional, intent(in) :: group
    character(len=*), optional, intent(in) :: scheme

    integer :: ierr
    ierr = 0

    if (present(scheme)) then
       call ccpp_physics_run(cdata, scheme_name=scheme, ierr=ierr)
    else if (present(group)) then
       call ccpp_physics_run(cdata, group_name=group, ierr=ierr)
    else
       call ccpp_physics_run(cdata, ierr=ierr)
    end if
    ! error handling as above

  end subroutine physics_run

  subroutine physics_finalize()
    integer :: ierr
    ierr = 0

    ! Finalize CCPP physics (run all _finalize routines)
    call ccpp_physics_finalize(cdata, ierr=ierr)
    ! error handling as above
    call ccpp_finalize(cdata, ierr=ierr)
    ! error handling as above

  end subroutine physics_finalize

end module example_ccpp_host_cap
```

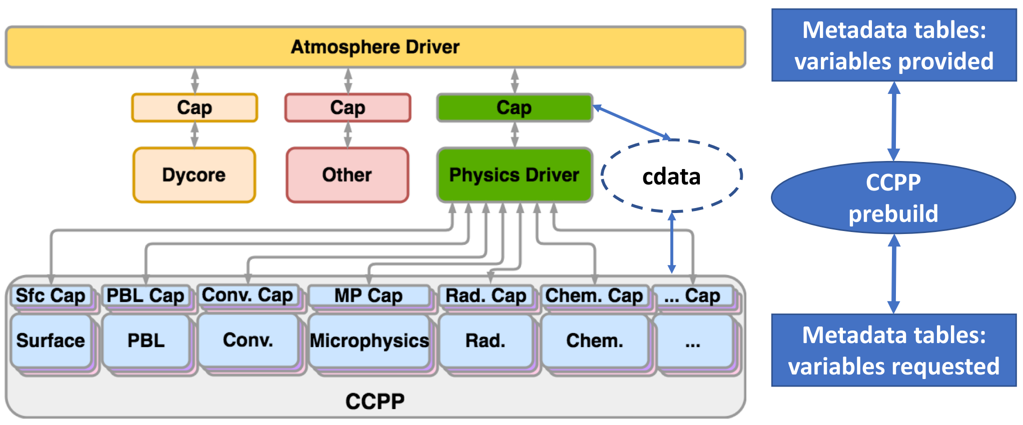

Figure 3.1: Role of the CCPP prebuild script and the `cdata` structure in the software architecture of an atmospheric modeling system.

3.4 Configuring and running the CCPP prebuild script

The CCPP prebuild script `ccpp-framework/scripts/ccpp_prebuild.py` is the central piece of code that connects the host model with the CCPP physics schemes (see Figure 3.1). This script must be run before compiling the CCPP physics library and the host model cap. The CCPP prebuild script automates several tasks based on the information collected from the metadata tables on the host model side and from the individual physics schemes:

- Compiles a list of variables required to run all schemes in the CCPP physics pool.
- Compiles a list of variables provided by the host model.
- Matches these variables by their `standard_name`, checks for missing variables and mismatches of their attributes (e. g., units, rank, type, kind) and processes information on optional variables (see also Sect. 2.1).
- Creates Fortran code (`ccpp_modules.inc`, `ccpp_fields.inc`) that stores pointers to the host model variables in the `cdata` structure.
- Auto-generates the caps for the physics schemes.
- Populates makefiles with schemes and caps.

In order to connect the CCPP with a host model *XYZ*, a Python-based configuration file for this model must be created in the directory `ccpp-framework/scripts`. The easiest way is to copy an existing configuration file in this directory, for example

```
cp ccpp_prebuild_config_SCM.py ccpp_prebuild_config_XYZ.py
```

The configuration in `ccpp_prebuild_config_XYZ.py` depends largely on (a) the directory structure of the host model itself, (b) where the `ccpp-framework` and the `ccpp-physics` directories are located relative to the directory structure of the host model, and (c) from which directory the `ccpp_prebuild.py` script is executed before/during the build process (this is referred to as `basedir` in `ccpp_prebuild_config_XYZ.py`).

Listing 3.5 contains an example for the SCM CCPP prebuild config. Here, it is assumed that both `ccpp-framework` and `ccpp-physics` are located in the top-level directory of the host model, and that `ccpp_prebuild.py` is executed from the same top-level directory.

Listing 3.5: CCPP prebuild config for SCM (shortened)

```
# Add all files with metadata tables on the host model side,
# relative to basedir = top-level directory of host model
VARIABLE_DEFINITION_FILES = [
    'scm/src/gmtb_scm_type_defs.f90',
    'scm/src/gmtb_scm_physical_constants.f90'
    ]

# Add all physics scheme dependencies relative to basedir - note that the CCPP
# rules stipulate that dependencies are not shared between the schemes!
SCHEME_FILES_DEPENDENCIES = [] # can be empty

# Add all physics scheme files relative to basedir
SCHEME_FILES = {
    # Relative path : [ list of sets in which scheme may be called ]
    'ccpp-physics/physics/GFS_DCNV_generic.f90' : ['physics'],
    'ccpp-physics/physics/sfc_sice.f'           : ['physics'],
    }

# Auto-generated makefile/cmakefile snippets that contains all schemes
SCHEMES_MAKEFILE = 'ccpp-physics/CCPP_SCHEMES.mk'
SCHEMES_CMAKEFILE = 'ccpp-physics/CCPP_SCHEMES.cmake'

# CCPP host cap in which to insert the ccpp_field_add statements;
# determines the directory to place ccpp_{modules,fields}.inc
TARGET_FILES = [
    'scm/src/gmtb_scm.f90',
    ]

# Auto-generated makefile/cmakefile snippets that contains all caps
CAPS_MAKEFILE = 'ccpp-physics/CCPP_CAPS.mk'
CAPS_CMAKEFILE = 'ccpp-physics/CCPP_CAPS.cmake'

# Directory where to put all auto-generated physics caps
CAPS_DIR = 'ccpp-physics/physics'

# Optional arguments - only required for schemes that use optional arguments.
# ccpp_prebuild.py will throw an exception if it encounters a scheme subroutine
# with optional arguments if no entry is made here. Possible values are:
OPTIONAL_ARGUMENTS = {
    #'subroutine_name_1' : 'all',
    #'subroutine_name_2' : 'none',
    #'subroutine_name_3' : [ 'var1', 'var2'],
    }

# HTML document containing the model-defined CCPP variables
HTML_VARTABLE_FILE = 'ccpp-physics/CCPP_VARIABLES.html'

# LaTeX document containing the provided vs requested CCPP variables
LATEX_VARTABLE_FILE = 'ccpp-framework/doc/DevelopersGuide/CCPP_VARIABLES.tex'

######## Template code to generate include files ########

# Name of the CCPP data structure in the host model cap;
# in the case of SCM, this is a vector with loop index i
CCPP_DATA_STRUCTURE = 'cdata(i)'

# Modules to load for auto-generated ccpp_field_add code
# in the host model cap (e.g. error handling)
MODULE_USE_TEMPLATE_HOST_CAP = \
'''
use ccpp_errors, only: ccpp_error
'''

# Modules to load for auto-generated ccpp_field_get code
# in the physics scheme cap (e.g. derived data types)
MODULE_USE_TEMPLATE_SCHEME_CAP = \
'''
       use machine, only: kind_phys
       use GFS_typedefs, only: GFS_statein_type, ...
'''

# EOF
```

Once the configuration in `ccpp_prebuild_config_XYZ.py` is complete, run

```
./ccpp-framework/scripts/ccpp_prebuild.py --model=XYZ [--debug]
```

from the top-level directory. Without the debugging flag, the output should look like

```
INFO: Logging level set to INFO
INFO: Parsing metadata tables for variables provided by host model ...
INFO: Parsed variable definition tables in module gmtb_scm_type_defs
INFO: Parsed variable definition tables in module gmtb_scm_physical_constants
INFO: Parsed variable definition tables in module ccpp_types
INFO: Metadata table for model SCM written to ccpp-physics/CCPP_VARIABLES_SCM.html
INFO: Parsing metadata tables in physics scheme files ...
INFO: Parsed tables in scheme rrtmg_lw
...
INFO: Checking optional arguments in physics schemes ...
INFO: Metadata table for model SCM written to ccpp-framework/doc/DevelopersGuide/
    CCPP_VARIABLES_SCM.tex
INFO: Comparing metadata for requested and provided variables ...
INFO: Generating module use statements for set physics ...
INFO: Generated module use statements for 4 module(s)
INFO: Generating ccpp_field_add statements for set physics ...
INFO: Generated ccpp_field_add statements for 606 variable(s)
INFO: Generating include files for host model cap scm/src/gmtb_scm.f90 ...
INFO: Generated module-use include file scm/src/ccpp_modules.inc
INFO: Generated fields-add include file scm/src/ccpp_fields.inc
INFO: Generating schemes makefile/cmakefile snippet ...
INFO: Added 81 schemes to ccpp-physics/CCPP_SCHEMES.mk and ccpp-physics/CCPP_SCHEMES.
    cmake
INFO: Generating caps makefile/cmakefile snippet ...
INFO: Added 64 auto-generated caps to ccpp-physics/CCPP_CAPS.mk and ccpp-physics/
    CCPP_CAPS.cmake
INFO: CCPP prebuild step completed successfully.
```

3.5 Building the CCPP framework and physics library

3.5.1 Preface

It is highly recommended to build the CCPP physics library and software framework as part of the host model. Both `ccpp-framework` and `ccpp-physics` use a cmake build system, which can be integrated in the host model's cmake build system, as it is the case for the SCM. For the example of FV3GFS, which employs a traditional make build system, the cmake build for the CCPP framework and physics are triggered by the host model's `compile.sh` script.

Note. It is possible to build the CCPP framework standalone, for example for testing purposes. It is generally not possible to build the CCPP physics library without running the CCPP prebuild script, since the build system relies on the auto-generated cmake code snippets that define the physics schemes and caps to compile. Further, any thirdparty libraries required by the physics schemes must be compiled and installed separately and the appropriate compiler and linker flags must be set manually. For example, the CCPP physics used by GMTB's SCM require several of NCEP's libraries (bacio, sp, w3nco); FV3GFS in addition requires the ESMF libraries and, depending on the operating system, also the Intel Math Kernel Library MKL (currently MacOSX only).

3.5.2 Standalone ccpp-framework build

The instructions laid out below demonstrate how to build the CCPP framework independently of the host model. It is assumed that the Github repository is checked out into a local directory `ccpp-framework`.

Set environment variables. In general, CCPP requires the `CC` and `FC` variables to point to the correct compilers. If threading (OpenMP) will be used inside the CCPP physics or the host model calling the CCPP physics, OpenMP-capable compilers must be used.

Build the CCPP framework. Use the following steps to build the CCPP framework.

```
cd ccpp-framework
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=$PWD ..
# add -DOPENMP=ON before .. for OpenMP build
# add -DCMAKE_BUILD_TYPE=Debug before .. for 'Debug' build
make
# add VERBOSE=1 for verbose output
make install
# add VERBOSE=1 after install for verbose output
```

Update environment variables. The previous install step creates directories `include` and `lib` inside the build directory. These directories and the newly built library `libccpp.so` need to be added to the environment variables `FFLAGS` and `LDFLAGS`, respectively (example for bash, assuming the current directory is still the above build directory):

```
export FFLAGS="-I$PWD/include/ccpp $FFLAGS"
export LDFLAGS="-L$PWD/lib -lccpp $LDFLAGS"
```

Testing the CCPP framework. Several unit tests are provided by the CCPP framework. These cover basic functionality and will be expanded to increase the test coverage in future releases. The unit tests are run from the build directory using

```
export LD_LIBRARY_PATH=$PWD/schemes/check/src/check-build:
   $LD_LIBRARY_PATH
make test
```

3.5.3 Integration with host model build system

To allow for a flexible configuration of the CCPP framework and physics with multiple models, the `CMakeLists.txt` configuration files for both packages use a cmake variable `PROJECT`. This variable can be set as part of the cmake call (`cmake -DPROJECT=XYZ`) or by a `CMakeLists.txt` that integrates `ccpp-framework` and `ccpp-physics`. If not specified, `PROJECT` is set to 'Unknown'.

The basic steps to build the CCPP framework and physics for a specific host model are outlined in the following.

Recommended directory structure. As mentioned in Section 3.4, we recommend placing the two directories (repositories) `ccpp-framework` and `ccpp-physics` in the top-level directory of the host model, and to adapt the CCPP prebuild config such that it can be run from the top-level directory. For FV3GFS, a slightly different directory structure is used that places `ccpp-framework` in `ccpp/framework` and `ccpp-physics` in `ccpp/physics`, and uses a shell script `ccpp/build_ccpp.sh` and a top-level cmake configuration `ccpp/CMakeLists.txt` for the build process.

Set environment variables. In addition to the compiler variables `CC` and `FC`, the CCPP physics require further enviroment variables for thirdparty libraries used by the physics schemes. The setup scripts for SCM (in `scm/etc`) or FV3GFS (in `conf` or `modulefiles`) provide useful examples for the correct environment settings.

Build the CCPP framework. See previous section on how to build the CCPP framework. The cmake variable `PROJECT` can be set to customize the build using `ccpp-framework/CMakeLists.txt`. This includes preprocessor flags such as `-DMPI`.

Update environment variables. See previous section on how to update the compiler and linker flags.

Build CCPP physics library. Before building `ccpp-physics`, its top-level cmake configuration `ccpp-physics/CMakeLists.txt` must be customized for the host model. This includes compiler flags, preprocessor flags etc. The user is referred to the existing configurations. The CCPP physics library is built starting from the build directory `ccpp-framework/build`:

```
cd ../.. # back to top-level directory
cd ccpp-physics
mkdir build && cd build
cmake ..
# add -DOPENMP=ON before .. for OpenMP build
# note that OpenMP build requires finding
# detect_openmp.cmake from ccpp-framework/cmake
make
# add VERBOSE=1 after install for verbose output
```

Following these steps, the include files and the library `libccpp.so` that the host model needs to be compiled and linked against are located in `ccpp-framework/build/include` and `ccpp-framework/build/lib`. Note that there is no need to link the host model to the CCPP physics library in `ccpp-physics/build`, as long as it is in the search path of the dynamic loader of the OS (for example by adding the directory `ccpp-physics/build` to the `LD_LIBRARY_PATH` environment variable). This is because the CCPP physics library is loaded dynamically by the CCPP framework using the library name specified in the runtime suite definition file (see the GMTB Single Column Model Technical Guide v2.1, Chapter 6.1.3, (https://dtcenter.org/gmtb/users/ccpp/docs/) for further information). Setting the environment variables `FFLAGS` and `LDFLAGS` as described for the CCPP framework standalone build in Sect. 3.5.2 should be sufficient to compile the host model with its newly created host model cap (Sect. 3.3) and connect to the CCPP library and framework.

For a complete integration of the CCPP infrastructure and physics library build systems in the host model build system, users are referred to the existing implementations in GMTB SCM and FV3GFS.