Common Community Physics Package (CCPP)

Developers' Guide
v1.0

April 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

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. For more information about the CCPP design and implementation, please 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 v1.0 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 expected in future releases.

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 but 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, 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}`. Note: at present, the `_init` and `_finalize` routines can not be used, and are simply placeholders (c. f. listing 2.1)
- 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`).
- All information (units, rank) must match the specifications on the host model side.

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 inside the module, i. e. no variables carrying the `save` attribute.
- 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:

- If OpenMP is used, the number of allowed threads must be provided by the host model as an `intent(in)` argument in the argument list.
- If MPI is used, it is restricted to global communications: barrier, broadcast, gather, scatter, reduce; the MPI communicator must be provided by the host model as an `intent(in)` argument in the argument list.
- If Fortran coarrays are used, consult with the GMTB helpdesk at (gmtb-help@ucar.edu).

Scientific Documentation rules:

- Scientific documentation is not technically 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 v1.0 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/.
- 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,
    ! create a meaningfull error message 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 (see, for example, GMTB Single Column Model Technical Guide v1.0, chapter 6.1.3, https://dtcenter.org/gmtb/users/ccpp/docs) and add the new scheme at the place it should be run.
9. Done!

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). In case a required variable 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 defined in a module or the main program.

With the CCPP, it is possible to refer to components of derived types or to slices of arrays in the metadata table (see Listing 3.1 in the following section for an example).

3.2 Adding metadata variable tables for the host model

In order 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, but 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:

- 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.
- For metadata tables describing module variables, the table must be placed inside the module.
- For metadata tables describing components of derived data types, the table must be placed immediately before the type definition.

Listing 3.1 provides examples for host model metadata tables.

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.

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

Listing 3.1: 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
```

`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.

Populating the `cdata` structure. Each variable required by the physics schemes must be added to the `cdata` structure 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 3.1), 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 current implementations of CCPP in SCM and FV3 require a few manual additions of variables to the `cdata` structure to complete the CCPP suite initialization step. These are special cases that will be addressed in the future.

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 (dycore) time integration loop and that internally call `ccpp_run` and handle any errors returned.

Listing 3.2 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`.

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, the CCPP framework 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.

Listing 3.2: Fortran template for a CCPP host model cap

```
module example_ccpp_host_cap

  use ccpp_types,          only: ccpp_t
  use ccpp,                only: ccpp_init, ccpp_finalize
  use ccpp_fcall,          only: ccpp_run
  use ccpp_fields,         only: ccpp_field_add
  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

    call ccpp_init(ccpp_suite_name, cdata, 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"

  end subroutine physics_init

  subroutine physics_run(step)
    ! the step (currently called IPD step) to run as
    ! defined in the runtime suite definition file
    integer, intent(in) :: step
    integer :: ierr
    ierr = 0

    call ccpp_run(cdata%suite%ipds(step), cdata, ierr)
    ! future versions: call ccpp_run(cdata, step=step, ierr=ierr)
    if (ierr/=0) then
      ! errmsg is known because of #include ccpp_modules.inc
      write(*,'(a,i0,a)') "An error occurred in physics step ", step, &
                          "; error message: '" // trim(errmsg) // "'"
      stop
    end if

  end subroutine physics_run

  subroutine physics_finalize()
    integer :: ierr
    ierr = 0

    call ccpp_finalize(cdata, ierr)
    if (ierr/=0) then
      write(*,'(a)') "An error occurred in ccpp_finalize"
      stop
    end if

  end subroutine physics_finalize

end module example_ccpp_host_cap
```

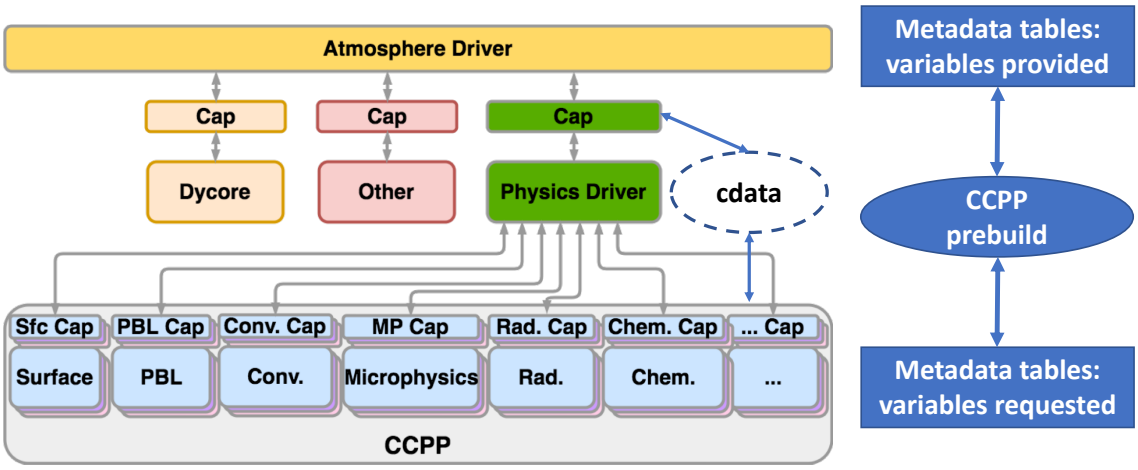

Figure 3.1: Role and position of the CCPP prebuild script and the `cdata` structure in the software architecture of an atmospheric modeling system.

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

```
cp ccpp_prebuild_config_FV3.py ccpp_prebuild_config_XYZ.py
```

and add `HOST_MODEL=XYZ` to section `User definitions` in `ccpp_prebuild.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`).

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 (recommended setup). The following variables need to be configured in `ccpp_prebuild_config_XYZ.py`, here shown for the example of SCM:

```
# 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 files relative to basedir
SCHEME_FILES = [
    'ccpp-physics/GFS_layer/GFS_initialize_scm.F90',
    'ccpp-physics/physics/GFS_DCNV_generic.f90',
    ...
    'ccpp-physics/physics/sfc_sice.f',
    ]

# Auto-generated makefile fragment that contains all schemes
SCHEMES_MAKEFILE = 'ccpp-physics/CCPP_SCHEMES.mk'
```

```
# 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 fragment that contains all caps
CAPS_MAKEFILE = 'ccpp-physics/CCPP_CAPS.mk'

# 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: 'all', 'none',
# or a list of standard_names: [ 'var1', 'var3' ].
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, ...
'''
```

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

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

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

```
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: Metadata table for model SCM written to ccpp-physics/CCPP_VARIABLES.html
INFO: Parsing metadata tables in physics scheme files ...
INFO: Parsed tables in scheme GFS_initialize_scm
INFO: Parsed tables in scheme GFS_DCNV_generic_pre
...
INFO: Parsed tables in scheme sfc_sice
INFO: Checking optional arguments in physics schemes ...
INFO: Metadata table for model SCM written to ccpp-framework/doc/DevelopersGuide/
    CCPP_VARIABLES.tex
INFO: Comparing metadata for requested and provided variables ...
INFO: Generating module use statements ...
INFO: Generated module use statements for 3 module(s)
INFO: Generating ccpp_field_add statements ...
INFO: Generated ccpp_field_add statements for 394 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 fragment ...
INFO: Added 38 schemes to makefile ccpp-physics/CCPP_SCHEMES.mk
INFO: Generating caps makefile fragment ...
INFO: Added 66 auto-generated caps to makefile ccpp-physics/CCPP_CAPS.mk
INFO: CCPP prebuild step completed successfully.
```

3.5 Building the CCPP physics library and software framework

3.5.1 Preface – word of caution

As of now, the CCPP physics library and software framework are built as part of the host model (SCM, FV3GFS). The SCM uses a `cmake` build system for both the CCPP physics library and the CCPP framework, while FV3GFS employs a traditional `make` build system for the the CCPP physics library and a `cmake` build system for the CCPP software framework. Accordingly, `CMakeLists.txt` files in the `ccpp-physics` directory tree refer to an SCM build, while `makefile` files refer to an FV3GFS build. Work is underway to provide a universal build system based on `cmake` that can be used with all host models.

It should be noted that the current build systems do not make full use of the makefile fragments auto-generated by `ccpp_prebuild.py` (c. f. previous section). The SCM uses hardcoded lists of physics schemes and auto-generated physics scheme caps, while FV3GFS makes use of the auto-generated list of physics scheme caps but uses a hardcoded list of physics scheme files. This is due to the fact that script `ccpp_prebuild.py` at the moment only produces traditional `makefile` fragments (e. g. `CCPP_SCHEMES.mk` and `CCPP_CAPS.mk`). Work is underway to create include files suitable for `cmake` for both schemes and caps, and to integrate these into the build system.

3.5.2 Build steps

The instructions laid out below to build the CCPP physics library and CCPP software framework independently of the host model make use of the `cmake` build system (which is also used with the GMTB SCM). Several steps are required in the following order:

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 configure the CCPP prebuild script such that it can be run from the top-level directory.

Set environment variables. In general, the 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 (see below), OpenMP-capable compilers must be used here. The setup scripts for SCM in `scm/etc` provide useful examples for the correct environment settings (note that setting `NETCDF` is not required for CCPP, but may be required for the host model).

Configure and run `ccpp_prebuild.py`. This step is described in detail in Sect. 3.4.

Build CCPP framework. The following steps outline a suggested way to build the CCPP framework:

```
cd ccpp-framework
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=$PWD ..
# add -DOPENMP=1 before .. for OpenMP build
# add -DCMAKE_BUILD_TYPE=Debug before .. for debug build
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$PWD/include -I$PWD/src $FFLAGS"
export LDFLAGS="-L$PWD/lib -lccpp"
```

Build CCPP physics library. Starting from the build directory `ccpp-framework/build`:

```
cd ../.. # back to top-level directory
cd ccpp-physics
mkdir build && cd build
cmake ..
# add -DOPENMP=1 before .. for OpenMP build
make
# add VERBOSE=1 after install for verbose output
```

3.5.3 Optional: Integration with host model build system

Following the steps outlined Section 3.5.2, the include files and the library `libccpp.so` that the host model needs to be compiled/linked against to call the CCPP physics through the CCPP framework 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 v1.0, Chapter 6.1.3, (https://dtcenter.org/gmtb/users/ccpp/docs/) for further information)

Thus, setting the environment variables `FFLAGS` and `LDFLAGS` as 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 implementation in the GMTB SCM.