CCPP Technical Documentation

Release 6.0.0

Bernardet, L., G. Firl, D. Heinzeller, L. Pan, M. Zhang, M. Kavulich, L Carson, and J. Schramm

Nov 16, 2022

For referencing this document please use:

Bernardet, L., G. Firl, D. Heinzeller, L. Pan, M. Zhang, M. Kavulich, J. Schramm, and L. Carson, 2022. CCPP Technical Documentation Release v6.0.0. Available at https://ccpp-techdoc.readthedocs.io/_/downloads/en/v6.0.0/pdf/.

CONTENTS

CHAPTER

ONE

CCPP OVERVIEW

Ideas for the Common Community Physics Package (*CCPP*) originated within the Earth System Prediction Capability physics interoperability group (now the Interagency Council for Advancing Meteorological Services; ICAMS), which has representatives from the US National Center for Atmospheric Research (*NCAR*), the Navy, National Oceanic and Atmospheric Administration (NOAA) Research Laboratories, NOAA National Weather Service, and other groups. Physics interoperability, or the ability to run a given physics *suite* in various *host models*, has been a goal of this multi-agency group for several years. An initial mechanism to run the physics of NOAA's Global Forecast System (GFS) model in other host models, the Interoperable Physics Driver (IPD), was developed by the NOAA Environmental Modeling Center (EMC) and later augmented by the NOAA Geophysical Fluid Dynamics Laboratory (GFDL).

The CCPP expanded on that work by meeting additional requirements put forth by NOAA, and brought new functionalities to the physics-dynamics interface. Those include the ability to choose the order of *parameterizations*, to *subcycle* individual parameterizations by running them more frequently than other parameterizations, and to group arbitrary *sets* of parameterizations allowing other computations in between them (e.g., dynamics and coupling computations). The IPD was phased out in 2021 in favor of the CCPP as a single way to interface with physics in the *UFS*

The architecture of the CCPP and its connection to a host model is shown in Figure 1.1. Two elements of the CCPP are highlighted: a library of physical parameterizations (*CCPP Physics*) that conforms to selected standards and an infrastructure (*CCPP Framework*) that enables connecting the physics to a host model. The third element (not shown) is the CCPP Single Column Model (*SCM*), a simple host model that can be used with the CCPP Physics and Framework.

The host model needs to have functional documentation (metadata) for any variable that will be passed to or received from the physics. The CCPP Framework is used to compare the variables requested by each physical *parameterization* against those provided by the host model[1], and to check whether they are available, otherwise an error will be issued. This process serves to expose the variables passed between physics and dynamics, and to clarify how information is exchanged among parameterizations. During runtime, the CCPP Framework is responsible for communicating the necessary variables between the host model and the parameterizations.

The CCPP Physics contains the parameterizations and suites that are used operationally in the UFS Atmosphere, as well as parameterizations that are under development for possible transition to operations in the future. The CCPP aims to support the broad community while benefiting from the community. In such a CCPP ecosystem (Figure 1.2), the CCPP can be used not only by the operational centers to produce operational forecasts, but also by the research community to conduct investigation and development. Innovations created and effectively tested by the research community can be funneled back to the operational centers for further improvement of the operational forecasts.

Both the CCPP Framework and the CCPP Physics are developed as open source code, follow industry-standard code management practices, and are freely distributed through GitHub (https://github.com/NCAR/ccpp-physics and https://github.com/NCAR/ccpp-framework). This documentation is housed in repository https://github.com/NCAR/ccpp-doc.

[1] As of this writing, the CCPP has been validated with two host models: the CCPP SCM and the atmospheric component of NOAA's Unified Forecast System (UFS) (hereafter the UFS Atmosphere) that utilizes the Finite-Volume Cubed Sphere (FV3) dynamical core. The CCPP can be utilized both with the global and limited-area configurations of the UFS Atmosphere. The CCPP has also been run experimentally with a Navy model. Work is under way to connect and validate the use of the CCPP Framework with NCAR models.

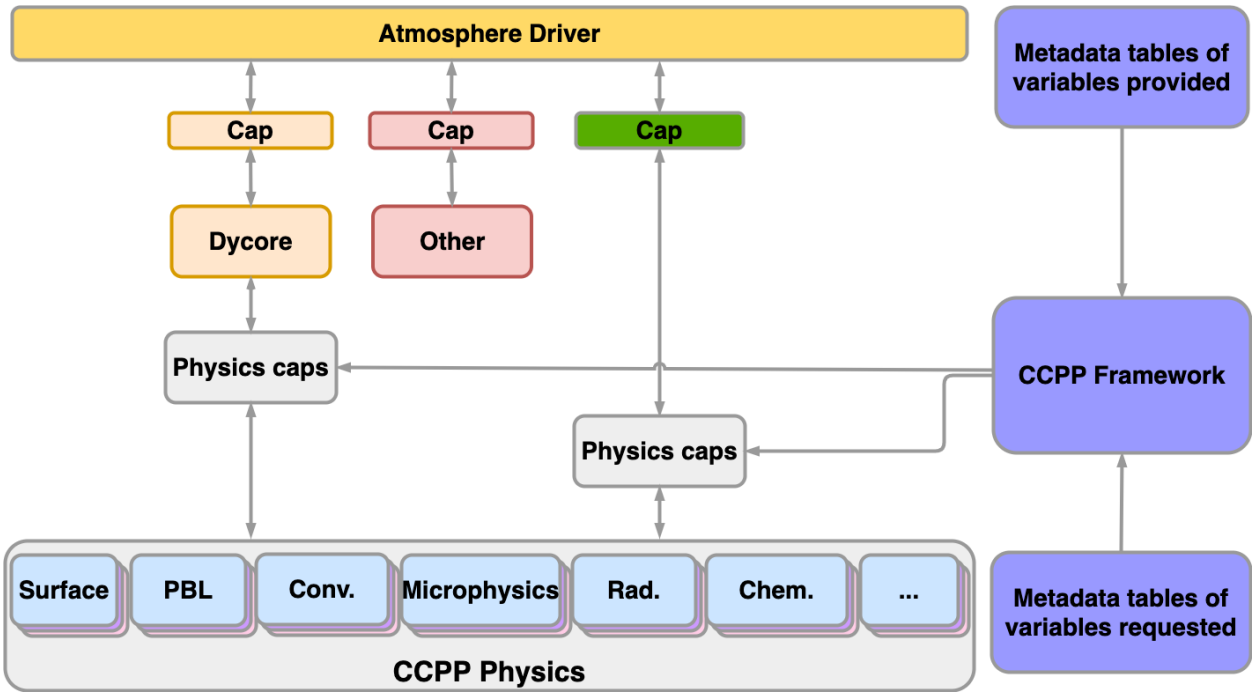


Fig. 1.1: *Architecture of the CCPP and its connection to a host model, represented here as the driver for an atmospheric model (yellow box). The dynamical core (dycore), physics, and other aspects of the model (such as coupling) are connected to the driving host through the pool of physics caps. The CCPP Physics is denoted by the gray box at the bottom of the physics, and encompasses the parameterizations, which are accompanied by physics caps.*

The CCPP is governed by the groups that contribute to its development. The CCPP Physics code management is collaboratively determined by NOAA, NCAR, and the Navy Research Laboratory (NRL), and the DTC works with EMC and its sponsors to determine *schemes* and suites to be included and supported. The governance of the CCPP Framework is jointly undertaken by NOAA and NCAR (see more information at https://github.com/NCAR/ccpp-framework/wiki and https://dtcenter.org/community-code/common-community-physics-package-ccpp).

The table below lists all parameterizations supported in CCPP public releases and the CCPP Scientific Documentation describes the parameterizations in detail. The parameterizations are grouped in suites, which can be classified primarily as *operational* or *developmental*. *Operational* suites are those used by operational, real-time weather prediction models. For this release, the only operational suite is GFS_v16, which is used for version 16 of the GFS model. *Developmental* suites are those that are officially supported for this CCPP release with one or more host models, but are not currently used in any operational models. These may include schemes needed exclusively for research, or "release candidate" schemes proposed for use with future operational models.

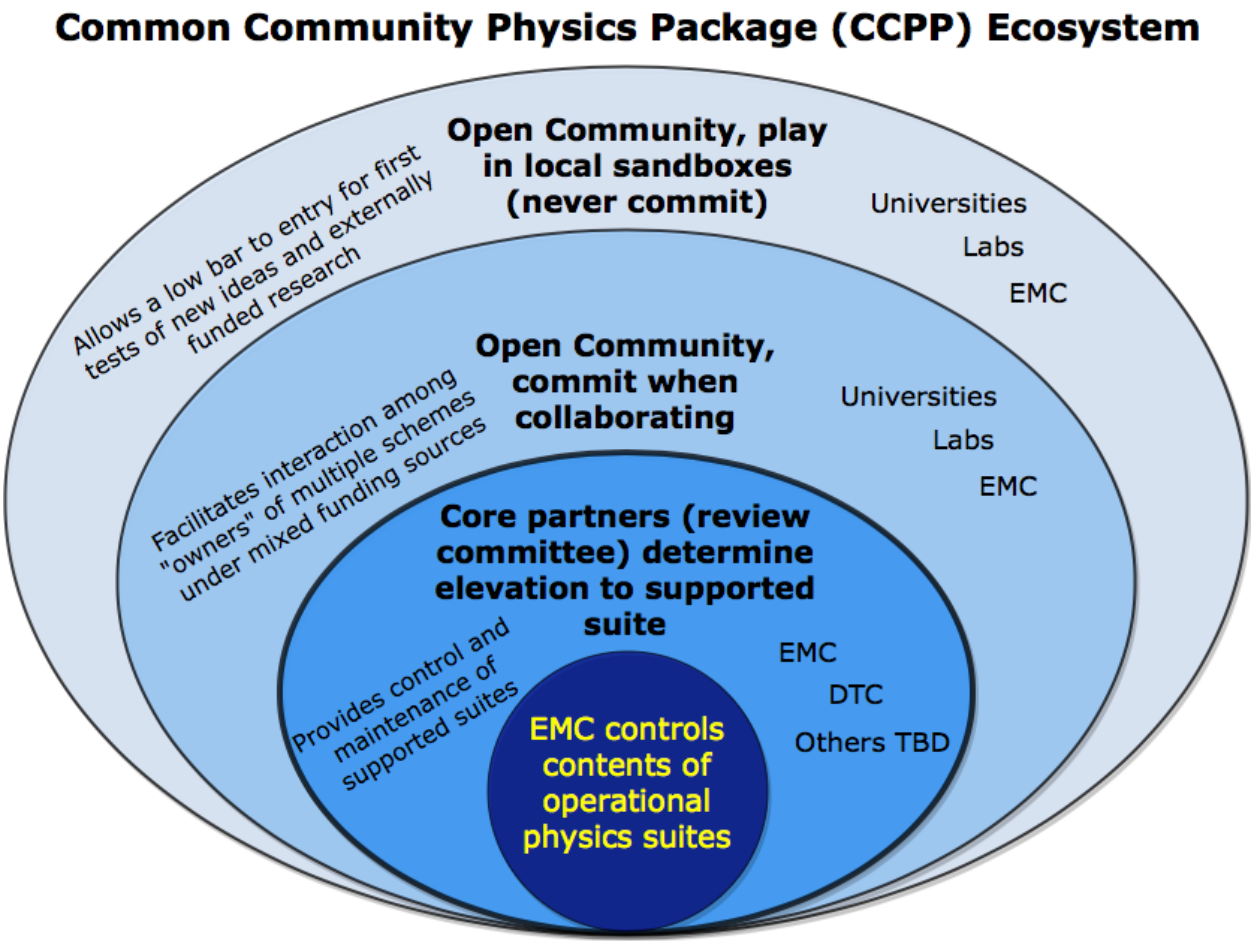


Fig. 1.2: *CCPP ecosystem.*

Table 1.1: *Suites supported in the CCPP for the UFS SRW v2.1.0 release*

	Operational	Developmental		
Physics Suite	GFS_v16	RRFS_v1beta	WoFS	HRRR
Microphysics	GFDL	Thompson	NSSL	Thompson
PBL	TKE EDMF	MYNN-EDMF	MYNN-EDMF	MYNN-EDMF
Deep convection	saSAS	N/A	N/A	N/A
Shallow convection	saMF	N/A	N/A	N/A
Radiation	RRTMG	RRTMG	RRTMG	RRTMG
Surface layer	GFS	MYNN-SFL	MYNN-SFL	MYNN-SFL
Gravity Wave Drag	CIRES-uGWP	CIRES-uGWP	CIRES-uGWP	GSL drag
Land surface	Noah	Noah-MP	Noah-MP	RUC
Ozone	NRL 2015	NRL 2015	NRL 2015	NRL 2015
Strat H_2O	NRL 2015	NRL 2015	NRL 2015	NRL 2015
Ocean	NSST	NSST	NSST	NSST

Only the suites supported with the UFS SRW App v2.1.0 release are listed in the table. Currently all supported suites use the 2015 Navy Research Laboratory (NRL) ozone and stratospheric water vapor schemes, and the NSST ocean scheme.

The operational GFS_v16 suite includes GFDL microphysics, the Turbulent Kinetic Energy (TKE)-based Eddy Diffusivity Mass-Flux (EDMF) planetary boundary layer (PBL) scheme, scale-aware (sa) Simplified Arakawa-Schubert (SAS) deep convection, scale-aware mass-flux (saMF) shallow convection, Rapid Radiation Transfer Model for General Circulation Models (RRTMG) radiation, GFS surface layer scheme, the Cooperative Institute for Research in the Environmental Sciences (CIRES) unified gravity wave drag (uGWD) scheme, and the Noah Land Surface Model (LSM).

The three developmental suites are either analogues for current operational physics schemes, or candidates for future operational implementations.

- The RRFS_v1beta suite is being used for development of the future Rapid Refresh Forecast System (RRFS), which is scheduled for implementation in late 2023. This scheme features Thompson microphysics, Mellor-Yamada-Nakanishi-Niino (MYNN) EDMF PBL, RRTMG radiation, MYNN surface layer (SFL) scheme, CIRES uGWD, and Noah Multiparameterization (Noah-MP) land surface parameterization. This suite, like the WoFS and HRRR suites that follow, does not feature convective parameterization due to its intended use at higher convective-permitting resolutions.
- The Warn-on-Forecast System (WoFS) suite is being used by the WoFS project at the National Severe Storms Laboratory (NSSL) for real-time and potential future operational high-resolution modeling products. The WoFS suite is identical to the RRFS_v1beta suite, except using NSSL 2-moment microphysics.
- Finally, the HRRR suite is similar to the operational High-Resolution Rapid Refresh (HRRR) model physics package, and features Thompson microphysics, Mellor-Yamada-Nakanishi-Niino (MYNN) EDMF PBL, RRTMG radiation, MYNN surface layer (SFL) scheme, Global Systems Laboratory (GSL) gravity wave drag scheme, and the Rapid Update Cycle (RUC) Land Surface Model.

Those interested in the history of previous CCPP releases should know that the first public release of the CCPP took place in April 2018 and included all the parameterizations of the operational GFS v14, along with the ability to connect to the SCM. The second public release of the CCPP took place in August 2018 and additionally included the physics suite tested for the implementation of GFS v15. The third public release of the CCPP, in June 2019, had four suites: GFS_v15, corresponding to the GFS v15 model implemented operationally in June 2019, and three developmental suites considered for use in GFS v16 (GFS_v15plus with an alternate PBL scheme, csawmg with alternate convection and microphysics schemes, and GFS_v0 with alternate convection, microphysics, PBL, and land surface schemes). The CCPP v4.0 release, issued in March 2020, contained suite GFS_v15p2, which is an updated version of the operational GFS v15 and replaced suite GFS_v15. It also contained three developmental suites: csawmg with minor updates, GSD_v1 (an update over the previously released GSD_v0), and GFS_v16beta, which was the target suite at the time for implementation in the upcoming operational GFSv16 (it replaced suite GFSv15plus). CCPP v4.0 was the first

release supported for use with the UFS Weather Model, more specifically as part of the UFS Medium-Range Weather (MRW) Application. The CCPP v4.1 release, issued in October 2020, was a minor upgrade with the capability to build the code using Python 3 (previously only Python 2 was supported). The CCPP v5.0 release, issued in February 2021, was a major upgrade to enable use with the UFS Short-Range Weather (SRW) Application and the RRFS_v1alpha suite. The CCPP v6.0.0 release, issued in June 2022, was a major upgrade in conjunction with the release of the UFS SRW v2.0 release.

1.1 Additional Resources

For the latest version of the released code and additional documentation, please visit the DTC Website.

Please post questions and comments to the GitHub discussions board for the relevant code repository:

- CCPP Physics https://github.com/NCAR/ccpp-physics/discussions
- CCPP Framework https://github.com/NCAR/ccpp-framework/discussions
- Single Column Model https://github.com/NCAR/ccpp-scm/discussions
- UFS Weather Model https://github.com/ufs-community/ufs-weather-model/discussions

1.2 How to Use this Document

This document contains documentation for the Common Community Physics Package (*CCPP*). It describes the

- *Primary physics schemes* and *interstitials*
- *Suite definition files*
- CCPP-compliant *parameterizations*
- Process to add a new scheme or suite
- Host-side coding
- CCPP code management and governance
- Parameterization-specific output
- Debugging strategies

The following table describes the type changes and symbols used in this guide.

Table 1.2: *Type changes and symbols used in this guide.*

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	• The names of commands, files, and directories • On-screen terminal output	• Edit your `.bashrc` file • Use `ls -a` to list all files. • `host$ You have mail!`
AaBbCc123	• The names of CCPP-specific terms, subroutines, etc. • Captions for figures, tables, etc.	• Each scheme must include at least one of the following subroutines: *_timestep_init*, *_init*, *_run*, *_finalize*, and *_timestep_finalize*. • *Listing 2.1: Fortran template for a CCPP-compliant scheme showing the* _run *subroutine.*
AaBbCc123	Words or phrases requiring particular emphasis	Fortran77 code should **not** be used

Following these typefaces and conventions, shell commands, code examples, namelist variables, etc. will be presented in this style:

```
mkdir ${TOP_DIR}
```

Some CCPP-specific terms will be highlighted using *italics*, and words requiring particular emphasis will be highlighted in **bold** text.

In some places there are helpful asides or warnings that the user should pay attention to; these will be presented in the following style:

Note: This is an important point that should **not** be ignored!

In several places in the technical documentation, we need to refer to locations of files or directories in the source code. Since the directory structure depends on the *host model*, in particular the directories where the `ccpp-framework` and `ccpp-physics` source code is checked out, and the directory from which the `ccpp_prebuild.py` code generator is called, we use the following convention:

1. When describing files relative to the `ccpp-framework` or `ccpp-physics` top-level, without referring to a specific model, we use `ccpp-framework/path/to/file/A` and `ccpp-physics/path/to/file/B`.
2. When describing specific tasks that depend on the directory structure within the host model, for example how to run `ccpp_prebuild.py`, we explicitly mention the host model and use its directory structure relative to the top-level directory. For the example of the *SCM*: `./ccpp/framework/path/to/file/A`.

CHAPTER

TWO

CCPP-COMPLIANT PHYSICS PARAMETERIZATIONS

The rules for a *scheme* to be considered *CCPP*-compliant are summarized in this section. It should be noted that making a scheme CCPP-compliant is a necessary but not guaranteed step for the acceptance of the scheme in the pool of supported *CCPP Physics*. Acceptance is dependent on scientific innovation, demonstrated value, and compliance with the rules described below. The criteria for acceptance of a scheme into the CCPP is under development.

It is recommended that *parameterizations* be comprised of the smallest units that will be used independently. For example, if a given pair of deep and shallow convection schemes will always be called together and in a pre-established order, it is acceptable to group them within a single scheme. However, if one envisions that the deep and shallow convection schemes may someday operate independently, it is recommended to code two separate schemes to allow more flexibility.

Some schemes in the CCPP have been implemented using a driver as an entry point. In this context, a driver is defined as a wrapper of code around the actual scheme, providing the CCPP entry points. In order to minimize the layers of code in the CCPP, the implementation of a driver is discouraged, that is, it is preferable that the CCPP be composed of atomic parameterizations. One example is the implementation of the MG microphysics, in which a simple entry point leads to two versions of the scheme, MG2 and MG3. A cleaner implementation would be to retire MG2 in favor of MG3, to turn MG2 and MG3 into separate schemes, or to create a single scheme that can behave as MG2 and MG3 depending on namelist options.

The implementation of a driver is reasonable under the following circumstances:

- To preserve schemes that are also distributed outside of the CCPP. For example, the Thompson microphysics scheme is distributed both with the Weather Research and Forecasting (WRF) model and with the CCPP. Having a driver with CCPP directives allows the Thompson scheme to remain intact so that it can be synchronized between the WRF model and the CCPP distributions. See more in `mp_thompson.F90` in the `ccpp-physics/physics` directory.
- To perform unit conversions or array transformations, such as flipping the vertical direction and rearranging the index order, for example, `cu_gf_driver.F90` or `gfdl_cloud_microphys.F90` in the `ccpp-physics/physics` directory.

Schemes in the CCPP are classified into two categories: *primary schemes* and *interstitial schemes*. A *primary* scheme is one that updates the state variables and tracers or that produces tendencies for updating state variables and tracers based on the representation of major physical processes, such as radiation, convection, microphysics, etc. This does **not** include:

- Schemes that compute tendencies exclusively for diagnostic purposes.
- Schemes that adjust tendencies for different timesteps (e.g., create radiation tendencies based on a radiation scheme called at coarser intervals).
- Schemes that update the model state based on tendencies generated in primary schemes.

Interstitial schemes are modularized pieces of code that perform data preparation, diagnostics, or other "glue" functions, and allow primary schemes to work together as a *suite*. They can be categorized as "scheme-specific" or "suite-level". Scheme-specific interstitial schemes augment a specific primary scheme (to provide additional functionality).

Suite-level interstitial schemes provide additional functionality on top of a class of primary schemes, connect two or more schemes together, or provide code for conversions, initializing sums, or applying tendencies, for example. The rules and guidelines provided in the following sections apply both to primary and interstitial schemes.

CCPP-compliant physics parameterizations are broken down into one or more of the following five *phases*:

- The *init* phase, which performs actions needed to set up the scheme before the model integration begins. Examples of actions needed in this phase include the reading/computation of lookup tables, setting of constants (as described in Section 2.6), etc.
- The *timestep_init* phase, which performs actions needed at the start of each physics timestep. Examples of actions needed in this phase include updating of time-based settings (e.g. solar angle), reading lookup table values, etc.
- The *run* phase, which is the main body of the scheme. Here is where the physics is integrated forward to the next timestep.
- The *timestep_finalize* phase, which performs post-integration calculations such as computing statistics or diagnostic tendencies. Not currently used by any scheme.
- The *finalize* phase, which performs cleanup and finalizing actions at the end of model integration. Examples of actions needed in this phase include deallocating variables, closing files, etc.

The various phases have different rules when it comes to parallelization, especially with regards to how data is blocked among parallel processes; see Section 2.7 for more information.

2.1 General Rules

A CCPP-compliant scheme is written in the form of Fortran modules. Each scheme must be in its own module, and must include at least one of the following subroutines (*entry points*): *_init*, *_timestep_init*, *_run*, *_timestep_finalize*, and *_finalize*. Each subroutine corresponds to one of the five *phases* of the *CCPP Framework* as described above. The module name and the subroutine names must be consistent with the scheme name; for example, the scheme "schemename" can have the entry points *schemename_init*, *schemename_run*, etc. The *_run* subroutine contains the code to execute the scheme. If subroutines *_timestep_init* or *_timestep_finalize* are present, they will be executed at the beginning and at the end of the *host model* physics timestep, respectively. Further, if present, the *_init* and *_finalize* subroutines associated with a scheme are run at the beginning and at the end of the model run. The *_init* and *_finalize* subroutines may be called more than once depending on the host model's parallelization strategy, and as such must be idempotent (the answer must be the same when the subroutine is called multiple times). This can be achieved by using a module variable `is_initialized` that keeps track whether a scheme has been initialized or not.

Listing 2.1 contains a template for a CCPP-compliant scheme, which includes the *_run* subroutine for an example *scheme_template* scheme. Each `.F` or `.F90` file that contains an entry point(s) for CCPP scheme(s) must be accompanied by a .meta file in the same directory as described in Section 2.2

```
    module scheme_template

      contains

!> \section arg_table_scheme_template_run Argument Table
!! \htmlinclude scheme_template_run.html
!!
      subroutine scheme_template_run (errmsg, errflg)

         implicit none

         !--- arguments
         ! add your arguments here
```

(continues on next page)

(continued from previous page)

```
        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 meaningful message to errmsg and return

        return

    end subroutine scheme_template_run

end module scheme_template
```

*Listing 2.1: Fortran template for a CCPP-compliant scheme showing the _run subroutine. The structure for the other phases (*_timestep_init, _init, _finalize, *and* _timestep_finalize*) is identical.*

The three lines in the example template beginning `!> \section` are required. They begin with *!* and so will be treated as comments by the Fortran compiler, but are interpreted by Doxygen as part of the process to create scientific documentation. Those lines specifically insert an external file containing metadata information (in this case, `scheme_template_run.html`) in the documentation. See more on this topic in Section 2.8.

All external information required by the scheme must be passed in via the argument list, including physical constants. Statements such as `use EXTERNAL_MODULE` should not be used for passing in any data. See Section 2.6 for more information on how to use physical constants.

Note that *standard names*, variable names, module names, scheme names and subroutine names are all case insensitive.

Interstitial modules (*schemename_pre* and *schemename_post*) can be included if any part of the physics scheme must be executed sequentially before (*_pre*) or after (*_post*) the scheme, but can not be included in the scheme itself (e.g., for including host-specific code).

2.2 Metadata Table Rules

Each CCPP-compliant physics scheme (`.F` or `.F90` file) must have a corresponding metadata file (`.meta`) that contains information about CCPP entry point schemes and their dependencies. These files contain two types of metadata tables: `ccpp-table-properties` and `ccpp-arg-table`, both of which are mandatory. The contents of these tables are described in the sections below.

Metadata files (`.meta`) are in a relaxed config file format and contain metadata for one or more CCPP entry points.

2.2.1 ccpp-table-properties

The `[ccpp-table-properties]` section is required in every metadata file and has four valid entries:

1. `type`: In the CCPP Physics, `type` can be `scheme`, `module`, or `ddt` (derived data type) and must match the `type` in the associated `[ccpp-arg-table]` section(s).
2. `name`: This depends on the `type`. For types `ddt` and `module` (for variable/type/kind definitions), `name` must match the name of the **single** associated `[ccpp-arg-table]` section. For type `scheme`, the name must match the root names of the `[ccpp-arg-table]` sections for that scheme, without the suffixes `_timestep_init`, `_init`, `_run`, `_finalize`, or `_timestep_finalize`.
3. `dependencies`: type/kind/variable definitions and physics schemes often depend on code in other files (e.g. "use machine" –> depends on `machine.F`). These dependencies must be provided as a comma-separated list. Relative path(s) to those file(s) must be specified here or using the `relative_path` entry described below. Dependency attributes are additive; multiple lines containing dependencies can be used. With the exception of specific files, such as `machine.F`, which provides the *kind_phys* Fortran kind definition, shared dependencies between schemes are discouraged.
4. `relative_path`: If specified, the relative path is added to every file listed in the `dependencies`.

The information in this section table allows the CCPP to compile only the schemes and dependencies needed by the selected CCPP suite(s).

An example for type and variable definitions from the file `ccpp-physics/physics/radlw_param.meta` is shown in *Listing 2.2*.

Note: A single metadata file may require multiple instances of the `[ccpp-table-properties]` section. For example, if a scheme requires multiple derived data types, each should have its own `[ccpp-table-properties]` entry. Subsequent `[ccpp-table-properties]` sections should be preceded by a separating line of `#` characters, as shown in the examples on this page.

```
[ccpp-table-properties]
  name = topflw_type
  type = ddt
  dependencies =

[ccpp-arg-table]
  name = topflw_type
  type = ddt

########################################################################
[ccpp-table-properties]
  name = sfcflw_type
  type = ddt
```

(continues on next page)

(continued from previous page)

```
  dependencies =

[ccpp-arg-table]
  name = sfcflw_type
  type = ddt

########################################################################
[ccpp-table-properties]
  name = proflw_type
  type = ddt
  dependencies =

[ccpp-arg-table]
  name = proflw_type
  type = ddt

########################################################################
[ccpp-table-properties]
  name = module_radlw_parameters
  type = module
  dependencies =

[ccpp-arg-table]
  name = module_radlw_parameters
  type = module
[topflw_type]
  standard_name = topflw_type
  long_name = definition of type topflw_type
  units = DDT
  dimensions = ()
  type = topflw_type
[sfcflw_type]
  standard_name = sfcflw_type
  long_name = definition of type sfcflw_type
  units = DDT
  dimensions = ()
  type = sfcflw_type
[proflw_type]
  standard_name = proflw_type
  long_name = definition of type proflw_type
  units = DDT
  dimensions = ()
  type = proflw_type
```

Listing 2.2: Example of a CCPP-compliant metadata file showing the use of the [ccpp-table-properties] section and how it relates to [ccpp-arg-table].

An example metadata file for the CCPP scheme `mp_thompson.meta` (with many sections omitted as indicated by `...`) is shown in *Listing 2.3*.

```
[ccpp-table-properties]
  name = mp_thompson
  type = scheme
```

(continues on next page)

(continued from previous page)

```
  dependencies = machine.F,module_mp_radar.F90,module_mp_thompson.F90,module_mp_thompson_
↪make_number_concentrations.F90

###############################################################################
[ccpp-arg-table]
  name = mp_thompson_init
  type = scheme
[ncol]
  standard_name = horizontal_dimension
  long_name = horizontal dimension
  units = count
  dimensions = ()
  type = integer
  intent = in
...

###############################################################################
[ccpp-arg-table]
  name = mp_thompson_run
  type = scheme
[ncol]
  standard_name = horizontal_loop_extent
  long_name = horizontal loop extent
  units = count
  dimensions = ()
  type = integer
  intent = in
...

###############################################################################
[ccpp-arg-table]
  name = mp_thompson_finalize
  type = scheme
[errmsg]
  standard_name = ccpp_error_message
  long_name = error message for error handling in CCPP
  units = none
  dimensions = ()
  type = character
  kind = len=*
  intent = out
...
```

Listing 2.3: Example metadata file for a CCPP-compliant physics scheme using a single `[ccpp-table-properties]` *entry and how it defines dependencies for multiple* `[ccpp-arg-table]` *entries. In this example the* timestep_init *and* timestep_finalize *phases are not used.*

2.2.2 ccpp-arg-table

For each CCPP compliant scheme, the `ccpp-arg-table` for a scheme, module or derived data type starts with this set of lines

```
[ccpp-arg-table]
 name = <name>
 type = <type>
```

- `ccpp-arg-table` indicates the start of a new metadata section for a given scheme.
- <name> is name of the corresponding subroutine/module.
- <type> can be `scheme`, `module`, or `DDT`.
- The metadata must describe all input and output arguments to the scheme using the following format:

```
[varname]
 standard_name = <standard_name>
 long_name = <long_name>
 units = <units>
 rank = <rank>
 dimensions = <dimensions>
 type = <type>
 kind = <kind>
 intent = <intent>
```

- The `intent` argument is only valid in `scheme` metadata tables, as it is not applicable to the other `types`.
- The following attributes are optional: `long_name`, `kind`.
- Lines can be combined using | as a separator, e.g.,

```
type = real | kind = kind_phys
```

- `[varname]` is the local name of the variable in the subroutine.
- The dimensions attribute should be empty parentheses for scalars or contain the `standard_name` for the start and end for each dimension of an array. `ccpp_constant_one` is the assumed start for any dimension which only has a single value. For example:

```
dimensions = ()
dimensions = (ccpp_constant_one:horizontal_loop_extent, vertical_level_dimension)
dimensions = (horizontal_dimension,vertical_dimension)
dimensions = (horizontal_dimension,vertical_dimension_of_ozone_forcing_data,number_of_
↪coefficients_in_ozone_forcing_data)
```

- The order of arguments in the entry point subroutines must match the order of entries in the metadata file.

Listing 2.4 contains the template .meta file for an example CCPP-compliant scheme (`scheme_template.meta`)

```
[ccpp-table-properties]
  name = ozphys
  type = scheme
  dependencies = machine.F

[ccpp-arg-table]
  name = ozphys_run
```

(continues on next page)

(continued from previous page)

```
  type = scheme
[errmsg]
  standard_name = ccpp_error_message
  long_name = error message for error handling in CCPP
  units = none
  dimensions = ()
  type = character
  kind = len=*
  intent = out
  optional = F
[errflg]
  standard_name = ccpp_error_code
  long_name = error code for error handling in CCPP
  units = 1
  dimensions = ()
  type = integer
  intent = out
  optional = F
```

Listing 2.4: Fortran template for a metadata file accompanying a CCPP-compliant scheme.

2.2.3 `horizontal_dimension` vs. `horizontal_loop_extent`

It is important to understand the difference between these metadata dimension names.

- `horizontal_dimension` refers to all (horizontal) grid columns that an MPI process owns/is responsible for, and that are passed to the physics in the *init*, *timestep_init*, *timestep_finalize*, and *finalize* phases.
- `horizontal_loop_extent` or, equivalent, `ccpp_constant_one:horizontal_loop_extent` stands for a subset of grid columns that are passed to the physics during the time integration, i.e. in the *run* phase.
- Note that `horizontal_loop_extent` is identical to `horizontal_dimension` for host models that pass all columns to the physics during the time integration.

Since physics developers cannot know whether a host model is passing all columns to the physics during the time integration or just a subset of it, the following rules apply to all schemes:

- Variables that depend on the horizontal decomposition must use
  - `horizontal_dimension` in the metadata tables for the following phases: *init*, *timestep_init*, *timestep_finalize*, *finalize*.
  - `horizontal_loop_extent` or `ccpp_constant_one:horizontal_loop_extent` in the *run* phase.

2.3 Standard names

Variables available for CCPP physics schemes are identified by their unique *standard name*. This policy is in place to ensure that variables will always be unique and unambiguous when communicating between different schemes and different host models. Schemes are free to use their own variable names within their individual codes, but these variables must be assigned to a *standard name* within the scheme's metadata table as described in Section 2.4.

Standard names are listed and defined in a GitHub repository (https://github.com/ESCOMP/CCPPStandardNames), along with rules for adding new standard names as needed. While an effort is made to comply with existing *standard name* definitions of the Climate and Forecast (CF) conventions (http://cfconventions.org), additional names are used in

the CCPP to cover the wide range of use cases the CCPP intends to include. Each hash of the CCPP Physics repository contains information in the top-level `README.md` file indicating which version of the CCPPStandardNames repository corresponds to that version of CCPP code.

An up-to-date list of available standard names for a given host model can be found by running the CCPP *prebuild* script (described in Chapter 8), which 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.

2.4 Input/Output Variable (argument) Rules

- 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 variable information (standard_name, units, dimensions) must match the specifications on the host model side, but sub-slices can be used/added in the host model. For example, when using the *UFS Atmosphere* as the host model, tendencies are split in `GFS_typedefs.meta` so they can be used in the necessary physics scheme:

```
[dt3dt(:,:,1)]
  standard_name = cumulative_change_in_temperature_due_to_longwave_radiation
  long_name = cumulative change in temperature due to longwave radiation
  units = K
  dimensions = (horizontal_dimension,vertical_dimension)
  type = real
  kind = kind_phys
[dt3dt(:,:,2)]
  standard_name = cumulative_change_in_temperature_due_to_shortwave_radiation
  long_name = cumulative change in temperature due to shortwave radiation
  units = K
  dimensions = (horizontal_dimension,vertical_dimension)
  type = real
  kind = kind_phys
[dt3dt(:,:,3)]
  standard_name = cumulative_change_in_temperature_due_to_PBL
  long_name = cumulative change in temperature due to PBL
  units = K
  dimensions = (horizontal_dimension,vertical_dimension)
  type = real
  kind = kind_phys
```

  For performance reasons, slices of arrays should be contiguous in memory, which, in Fortran, implies that the dimension that is split is the rightmost (outermost) dimension as in the example above.

- The two mandatory variables that any scheme-related subroutine must accept as `intent(out)` arguments are `errmsg` and `errflg` (see also coding rules in Section 2.5).
- At present, only two types of variable definitions are supported by the CCPP Framework:
  - Standard intrinsic Fortran variables are preferred (`character`, `integer`, `logical`, `real`, `complex`). For character variables, the length should be specified as * in order to allow the host model to specify the corresponding variable with a length of its own choice. 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, some use cases may justify their application (e.g. DDTs for chemistry that contain tracer arrays or information on whether tracers are advected). These DDTs must be defined by the scheme itself, not by the host model. It should be understood that

use of DDTs within schemes forces their use in host models and potentially limits a scheme's portability. Where possible, DDTs should be broken into components that could be usable for another scheme of the same type.

- It is preferable to have separate variables for physically-distinct quantities. For example, an array containing various cloud properties should be split into its individual physically-distinct components to facilitate generality. An exception to this rule is if there is a need to perform the same operation on an array of otherwise physically-distinct variables. For example, tracers that undergo vertical diffusion can be combined into one array where necessary. This tactic should be avoided wherever possible, and is not acceptable merely as a convenience.
- If a scheme is to make use of CCPP's *subcycling* capability, the current loop counter and the loop extent can be obtained from CCPP as `intent(in)` variables (see a *mandatory list of variables* that are provided by the CCPP Framework and/or the host model for this and other purposes).
- It is preferable to use assumed-size array declarations for input/output variables for CCPP schemes, i.e. instead of

```
real(kind=kind_phys), dimension(is:ie,ks:ke), intent(inout) :: foo
```

one should use

```
real(kind=kind_phys), dimension(:,:), intent(inout) :: foo
```

This allows the compiler to perform bounds checking and detect errors that otherwise may go unnoticed.

> **Warning:** Fortran assumes that the lower bound of assumed-size arrays is 1. If `foo` has lower bounds `is` and `ks` that are different from 1, then these must be specified explicitly:

```
real(kind=kind_phys), dimension(is:,ks:), intent(inout) :: foo
```

2.5 Coding Rules

- Code must comply to modern Fortran standards (Fortran 90 or newer), where possible.
- Uppercase file endings (*.F*, *.F90*) are preferred to enable preprocessing by default.
- Labeled `end` statements should be used for modules, subroutines, functions, and type definitions; for example, `module scheme_template` → `end module scheme_template`.
- Implicit variable declarations are not allowed. The `implicit none` statement is mandatory and is preferable at the module-level so that it applies to all the subroutines in the module.
- All `intent(out)` variables must be set inside the subroutine, including the mandatory variables `errflg` and `errmsg`.
- Decomposition-dependent host model data inside the module cannot be permanent, i.e. variables that contain domain-dependent data cannot be kept using the `save` attribute.
- The use of `goto` statements is discouraged.
- `common` blocks are not allowed.
- Schemes are not allowed to abort/stop execution.
- Errors are handled by the host model using the two mandatory arguments `errmsg` and `errflg`. In the event of an error, a meaningful error message should be assigned to `errmsg` and `errflg` set to a value other than 0. For example:

```
errmsg = 'Logic error in scheme xyz: ...'
errflg = 1
return
```

- Schemes are not allowed to perform I/O operations except for reading lookup tables or other information needed to initialize the scheme, including stdout and stderr. Diagnostic messages are tolerated, but should be minimal.
- Line lengths of no more than 120 characters are suggested for better readability.

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 NOAA Environmental Modeling System (*NEMS*) Modeling Applications and Suites (available at https://docs.google.com/document/u/1/d/1bjnyJpJ7T3XeW3zCnhRLTL5a3m4_3XIAUeThUPWD9Tg/edit).

2.6 Using Constants

There are two principles that must be followed when using physical constants within CCPP-compliant physics schemes:

1. All schemes should use a single, consistent set of constants.
2. The host model must control (define and use) that single set, to provide consistency between a host model and the physics.

As long as a host application provides metadata describing its physical constants so that the CCPP framework can pass them to the physics schemes, these two principles are realized, and the CCPP physics schemes are model-agnostic. Since CCPP-compliant hosts provide metadata about the available physical constants, they can be passed into schemes like any other data.

For simple schemes that consist of one or two files and only a few "helper" subroutines, passing in physical constants via the argument list and propagating those constants down to any subroutines that need them is the most direct approach. The following example shows how the constant `karman` can be passed into a physics scheme:

```
subroutine my_physics_run(im,km,ux,vx,tx,karman)
  ...
real(kind=kind_phys),intent(in)   ::  karman
```

Where the following has been added to the `my_physics.meta` file:

```
[karman]
  standard_name = von_karman_constant
  long_name = von karman constant
  units = none
  dimensions = ()
  type = real
  intent = in
```

This allows the von Karman constant to be defined by the host model and be passed in through the CCPP scheme subroutine interface.

For pre-existing complex schemes that contain many software layers and/or many "helper" subroutines that require physical constants, another method is accepted to ensure that the two principles are met while eliminating the need to modify many subroutine interfaces. This method passes the physical constants once through the argument list for the top-level `_init` subroutine for the scheme. This top-level `_init` subroutine also imports scheme-specific constants from a user-defined module. For example, constants can be set in a module as:

```
module my_scheme_common
  use machine,     only : kind_phys
  implicit none
  real(kind=kind_phys)   ::  pi, omega1, omega2
end module my_scheme_common
```

Within the `_init` subroutine body, the constants in the `my_scheme_common` module can be set to the ones that are passed in via the argument list, including any derived ones. For example:

```
module my_scheme
  use machine, only: kind_phys
  implicit none
  private
  public my_scheme_init, my_scheme_run, my_scheme_finalize
  logical :: is_initialized = .false.
contains
  subroutine my_scheme_init (a, b, con_pi, con_omega)
    use my_scheme_common, only: pi, omega1, omega2
      ...
    pi = con_pi
    omega1 = con_omega
    omega2 = 2.*omega1
      ...
    is_initialized = .true.
  end subroutine my_scheme_init

  subroutine my_scheme_run (a, b)
    use my_scheme_common, only: pi, omega1, omega2
      ...
  end subroutine my_scheme_run

  subroutine my_scheme_finalize
      ...
    is_initialized = .false.
    pi = -999.
    omega1 = -999.
    omega2 = -999.
      ...
  end subroutine my_scheme_finalize
end module my_scheme
```

After this point, physical constants can be imported from `my_scheme_common` wherever they are needed. Although there may be some duplication in memory, constants within the scheme will be guaranteed to be consistent with the rest of physics and will only be set/derived once during the initialization phase. Of course, this will require that any constants in `my_scheme_common` that are coming from the host model cannot use the Fortran `parameter` keyword. To guard against inadvertently using constants in `my_scheme_common` without setting them from the host, they should be initially set to some invalid value. The above example also demonstrates the use of `is_initialized` to guarantee idempotence of the `_init` routine. To clean up during the finalize phase of the scheme, the `is_initialized` flag can be set back to false and the constants can be set back to an invalid value.

In summary, there are two ways to pass constants to a physics scheme. The first is to directly pass constants via the subroutine interface and continue passing them down to all subroutines as needed. The second is to have a user-specified scheme constants module within the scheme and to sync it once with the physical constants from the host model at initialization time. The approach to use is somewhat up to the developer.

Note: Use of the *physcons* module (`ccpp-physics/physics/physcons.F90`) is **not recommended**, since it is specific to FV3 and will be removed in the future.

2.7 Parallel Programming Rules

Most often, shared memory (OpenMP: Open Multi-Processing) and distributed memory (MPI: Message Passing Interface) communication is done outside the physics, in which case the loops and arrays already take into account the sizes of the threaded tasks through their input indices and array dimensions.

The following rules should be observed when including OpenMP or MPI communication in a physics scheme:

- CCPP standards require that in every phase but the *run* phase, blocked data structures must be combined so that their entire contents are available to a given MPI task (i.e. the data structures can not be further subdivided, or "chunked", within those phases). The *run* phase may be called by multiple threads in parallel, so data structures may be divided into blocks for that phase.
- 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 (*Listing 6.2*).
- MPI communication is allowed in the *init*, *timestep_init*, *timestep_finalize*, and *finalize*, phases for the purpose of computing, reading or writing scheme-specific data that is independent of the host model's data decomposition.
- If MPI is used, it is restricted to global communications: barrier, broadcast, gather, scatter, reduction. Point-to-point communication is not allowed. The MPI communicator must be passed to the physics scheme by the host model, the use of `MPI_COMM_WORLD` is not allowed (*see list of mandatory variables*).
- An example of a valid use of MPI is the initial read of a lookup table of aerosol properties by one or more MPI processes, and its subsequent broadcast to all processes.
- The implementation of reading and writing of data must be scalable to perform efficiently from a few to thousands of tasks.
- Calls to MPI and OpenMP functions, and the import of the MPI and OpenMP libraries, must be guarded by C preprocessor directives as illustrated in the following listing. OpenMP pragmas can be inserted without C preprocessor 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 CCPP Forum (https://dtcenter.org/forum/ccpp-user-support).

2.8 Scientific Documentation Rules

Scientific and technical documents are important for code maintenance and for fostering understanding among stakeholders. As such, physics *schemes* are required to include scientific documentation in order to be included in the *CCPP*. This section describes the process used for documenting *parameterizations* in the CCPP.

Doxygen was chosen as a tool for generating human-readable output due to its built-in functionality with Fortran, its high level of configurability, and its ability to parse inline comments within the source code. Keeping documentation with the source itself increases the likelihood that the documentation will be updated along with the underlying code. Additionally, inline documentation is amenable to version control.

The purpose of this section is to provide an understanding of how to properly document a physics scheme using doxygen inline comments in the Fortran code and metadata information contained in the `.meta` files. It covers what kind of information should be in the documentation, how to mark up the inline comments so that doxygen will parse them correctly, where to put various comments within the code, how to include information from the `.meta` files, and how to configure and run doxygen to generate HTML output. For an example of the HTML rendering of the CCPP Scientific Documentation, see https://dtcenter.ucar.edu/GMTB/UFS_SRW_App_v2.1.0/sci_doc/index.html Part of this documentation, namely metadata about subroutine arguments, has functional significance as part of the CCPP infrastructure. The metadata must be in a particular format to be parsed by Python scripts that "automatically" generate a *software cap* for a given physics scheme. Although the procedure outlined herein is not unique, following it will provide a level of continuity with previous documented schemes.

Reviewing the documentation for CCPP parameterizations is a good way of getting started in writing documentation for a new scheme.

2.8.1 Doxygen Comments and Commands

All doxygen commands start with a backslash ("\") or an at-sign ("@"). The doxygen inline comment blocks begin with "!>", and subsequent lines begin with "!!", which means that regular Fortran comments using "!" are not parsed by doxygen.

In the first line of each Fortran file, a brief one-sentence overview of the file's purpose should be included, using the doxygen command `\file`:

```
!> \file cires_ugwp.F90
!! This file contains the Unified Gravity Wave Physics (UGWP) scheme by Valery Yudin␣
↪(University of Colorado, CIRES)
```

A parameter definition begins with "!<", where the "<" sign tells Doxygen that documentation follows. Example:

```
integer, parameter, public :: NF_VGAS = 10   !< number of gas species
integer, parameter         :: IMXCO2  = 24   !< input CO2 data longitude points
integer, parameter         :: JMXCO2  = 12   !< input CO2 data latitude points
integer, parameter         :: MINYEAR = 1957 !< earlist year 2D CO2 data available
```

2.8.2 Doxygen Documentation Style

To document a physics *suite*, a broad array of information should be included in order to serve both software engineering and scientific purposes. The documentation style could be divided into four categories:

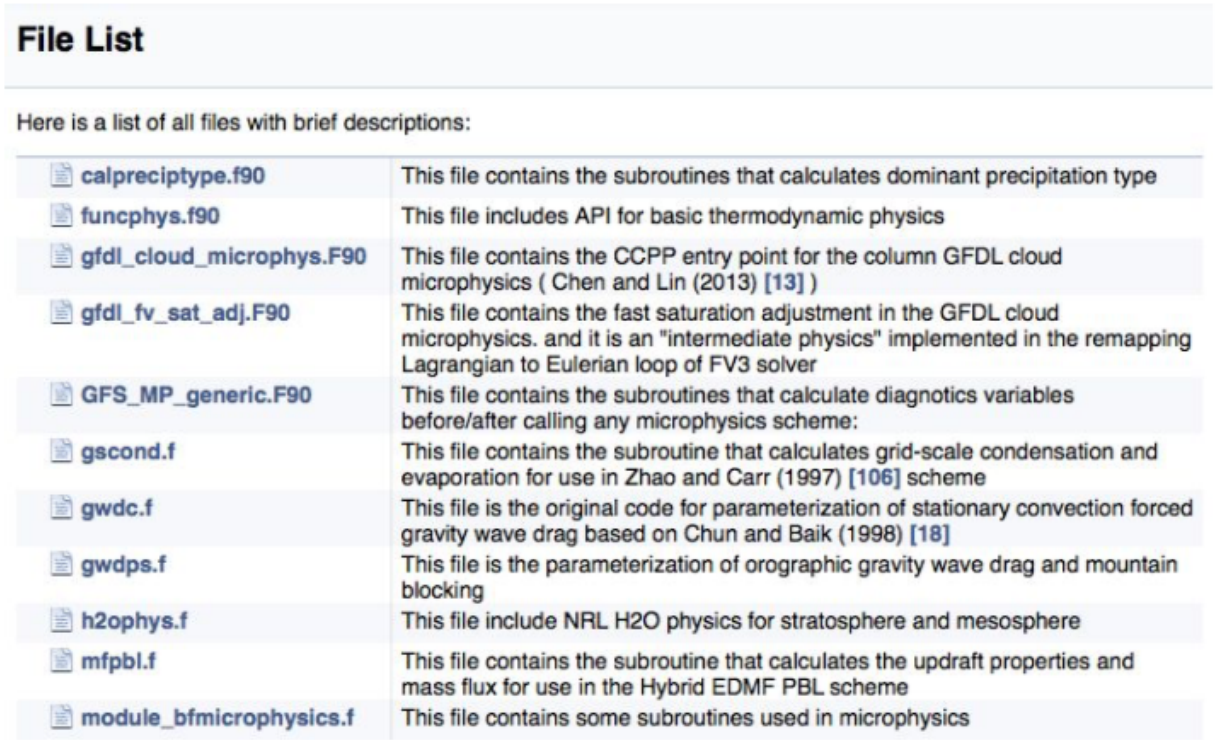
- Doxygen Files
- Doxygen Pages (overview page and scheme pages)
- Doxygen Modules
- Bibliography

Doxygen files

Doxygen provides the `\file` tag as a way to provide documentation on the level of Fortran source code files. That is, in the generated documentation, one may navigate by source code filenames (if desired) rather than through a "functional" navigation. The most important documentation organization is through the "module" concept mentioned below, because the division of a scheme into multiple source files is often functionally irrelevant. Nevertheless, using a `\file` tag provides an alternate path to navigate the documentation and it should be included in every source file. Therefore, it is prudent to include a small documentation block to describe what code is in each file using the `\file` tag, e.g.:

```
!>\file cu_gf_deep.F90
!! This file is the Grell-Freitas deep convection scheme.
```

The brief description for each file is displayed next to the source filename on the doxygen-generated "File List" page:



File List

Here is a list of all files with brief descriptions:

calpreciptype.f90	This file contains the subroutines that calculates dominant precipitation type
funcphys.f90	This file includes API for basic thermodynamic physics
gfdl_cloud_microphys.F90	This file contains the CCPP entry point for the column GFDL cloud microphysics (Chen and Lin (2013) [13])
gfdl_fv_sat_adj.F90	This file contains the fast saturation adjustment in the GFDL cloud microphysics. and it is an "intermediate physics" implemented in the remapping Lagrangian to Eulerian loop of FV3 solver
GFS_MP_generic.F90	This file contains the subroutines that calculate diagnotics variables before/after calling any microphysics scheme:
gscond.f	This file contains the subroutine that calculates grid-scale condensation and evaporation for use in Zhao and Carr (1997) [106] scheme
gwdc.f	This file is the original code for parameterization of stationary convection forced gravity wave drag based on Chun and Baik (1998) [18]
gwdps.f	This file is the parameterization of orographic gravity wave drag and mountain blocking
h2ophys.f	This file include NRL H2O physics for stratosphere and mesosphere
mfpbl.f	This file contains the subroutine that calculates the updraft properties and mass flux for use in the Hybrid EDMF PBL scheme
module_bfmicrophysics.f	This file contains some subroutines used in microphysics

Doxygen Overview Page

Pages in Doxygen can be used for documentation that is not directly attached to a source code entity such as a file or module. In the context of CCPP they are used for external text files that generate pages with a high-level scientific overview, typically containing a longer description of a project or suite. You can refer to any source code entity from within a page.

The DTC maintains a main page, created by the Doxygen command `\mainpage`, which contains an overall description and background of the CCPP. Physics developers do not have to edit the file with the mainpage (`mainpage.txt`), which is formatted like this:

```
/**
\mainpage Introduction
...
*/
```

All other pages listed under the main page are created using the Doxygen tag `\page` described in the next section. In any Doxygen page, you can refer to any entity of source code by using Doxygen tag `\ref` or `@ref`. Example from `suite_FV3_GFS_v16.txt`:

```
/**
\page GFS_v16_page GFS_v16 Suite

\section gfsv16_suite_overview Overview

Version 16 of the Global Forecast System (GFS) was implemented operationally by the NOAA
National Centers for Environmental Prediction (NCEP) in 2021. This suite is available for
use with the UFS SRW App and with the CCPP SCM.

The GFS_v16 suite uses the parameterizations in the following order:
 - \ref GFS_RRTMG
 - \ref GFS_SFCLYR
 - \ref GFS_NSST
 - \ref GFS_OCEAN
 - \ref GFS_NOAH
 - \ref GFS_SFCSICE
 - \ref GFS_SATMEDMFVDIFQ
 - \ref GFS_UGWP_v0
 - \ref GFS_OZPHYS
 - \ref GFS_H2OPHYS
 - \ref GFS_SAMFdeep
 - \ref GFS_SAMFshal
 - \ref GFDL_cloud

\section sdf_gfsv16b Suite Definition File
\include  suite_FV3_GFS_v16.xml
...
*/
```

The HTML result of this Doxygen code can be viewed here. You can see that the - symbols at the start of a line generate a list with bullets, and the `\ref` commands generate links to the appropriately labeled pages. The `\section` comands indicate section breaks, and the `\include` commands will include the contents of another file.

Other valid Doxygen commands for style, markup, and other functionality can be found in the Doxygen documentation.

Physics Scheme Pages

Each major scheme in CCPP should have its own scheme page containing an overview of the parameterization. These pages are not tied to the Fortran code directly; instead, they are created with a separate text file that starts with the command `\page`. For CCPP, the stand-alone Doxygen pages, including the main page and the scheme pages, are contained in the *ccpp-physics* repository, under the `ccpp-physics/physics/docs/pdftxt/` directory. Each page (aside from the main page) has a *label* (e.g., "GFS_SAMFdeep" in the following example) and a user-visible title ("GFS Scale-Aware Simplified Arakawa-Schubert (sa-SAS) Deep Convection Scheme" in the following example). It is noted that labels must be unique across the entire doxygen project so that the `\ref` command can be used to create an unambiguous link to the structuring element. It therefore makes sense to choose label names that refer to their context.

```
/**
\page GFS_SAMFdeep GFS Scale-Aware Simplified Arakawa-Schubert (sa-SAS) Deep Convection␣
↪Scheme
\section des_deep Description
 The scale-aware mass-flux (SAMF) deep convection scheme is an
 updated version of the previous Simplified Arakawa-Schubert (SAS) scheme
 with scale and aerosol awareness and parameterizes the effect of deep
 convection on the environment (represented by the model state variables)
 in the following way...
...
\section intra_deep  Intraphysics Communication
\ref arg_table_samfdeepcnv_run

\section gen_al_deep General Algorithm
\ref general_samfdeep

*/
```

The physics scheme page will often describe the following:

1. A "Description" section, which usually includes:
   - **Scientific origin and scheme history**
     - External sources and citations can be referenced with `\cite` tags
   - Key features and differentiating points compared to other schemes
   - **Tables, schematics, other images inserted using the `\image` tag**
     - To insert images into doxygen documentation, you'll need to prepare your images in a graphical format, such as Portable Network Graphic (png), depending on which type of doxygen output you are planning to generate. For example, for LaTeX output, the images must be provided in Encapsulated PostScript (.eps), while for HTML output the images can be provided in the png format. Images are stored in `ccpp-physics/physics/docs/img` directory. Example of including an image for HTML output:

       ```
       \image  html  gfdl_cloud_mp_diagram.png "Figure 1: GFDL MP at a glance␣
       ↪(Courtesy of S.J. Lin at GFDL)" width=10cm
       ```

2. An "Intraphysics Communication" section

   The argument table for CCPP entry point subroutine `{scheme}_run` will be in this section. It is created by inserting a reference link (`\ref`) to the corresponding Doxygen label in the Fortran code for the scheme. In the *above example*, the `\ref arg_table_samfdeepcnv_run` tag references the section of Doxygen-annotated source code in `ccpp-physics/physics/samfdeepcnv.f` that contains the scheme's argument table as an included html document, as described in the *following section*.

3. A "General Algorithm" section

   The general description of the algorithm will be in this section. It is created by inserting a reference link (\ref) pointing to the corresponding Doxygen-annotated source code for the scheme, as described in the *following section*.

As can be seen in the above examples, symbols /** and */ need to be the first and last entries of the page.

Note that separate pages can also be created to document something that is not a scheme. For example, a page could be created to describe a suite, or how a set of schemes work together. Doxygen automatically generates an index of all pages that is visible at the top-level of the documentation, thus allowing the user to quickly find, and navigate between, the available pages.

Doxygen Modules

The CCPP documentation is based on doxygen modules (note this is not the same as Fortran modules). Each doxygen module pertains to a particular parameterization and is used to aggregate all code related to that scheme, even when it is in separate files. Since doxygen cannot know which files or subroutines belong to each physics scheme, each relevant subroutine must be tagged with the module name. This allows doxygen to understand your modularized design and generate the documentation accordingly. Here is a list of modules defined in CCPP.

A module is defined using:

```
!>\defgroup group_name group_title
```

Where `group_name` is the identifier and the `group_title` is what the *group* is referred to in the output. In the example below, we're defining a parent module "GFS radsw Main":

```
!> \defgroup module_radsw_main GFS radsw Main
!! This module includes NCEP's modifications of the RRTMG-SW radiation
!! code from AER.
!! ...
!!\author   Eli J. Mlawer, emlawer@aer.com
!!\author   Jennifer S. Delamere, jdelamer@aer.com
!!\author   Michael J. Iacono, miacono@aer.com
!!\author   Shepard A. Clough
!!\version NCEP SW v5.1  Nov 2012 -RRTMG-SW v3.8
!!
```

One or more contact persons should be listed with author. If you make significant modifications or additions to a file, consider adding an author and a version line for yourself. The above example generates the Author, Version sections on the page. All email addresses are converted to mailto hypertext links automatically:

Author
Eli J. Mlawer, emlawer@aer.com

Jennifer S. Delamere, jdelamer@aer.com

Michael J. Iacono, miacono@aer.com

Shepard A. Clough

Version
NCEP SW v5.1 Nov 2012 -RRTMG-SW v3.8

In order to include other pieces of code in the same module, the following tag must be used at the beginning of a comment block:

```
\ingroup group_name
```

For example:

```
!>\ingroup module_radsw_main
!> The subroutine computes the optical depth in band 16:  2600-3250
!! cm-1 (low - h2o,ch4; high - ch4)
!----------------------------------
      subroutine taumol16
!..................................
```

In the same comment block where a group is defined for a physics scheme, there should be some additional documentation. First, using the `\brief` command, a brief one or two sentence description of the scheme should be included. After a blank doxygen comment line, begin the scheme origin and history using `\version`, `\author` and `\date`.

Each subroutine that is a CCPP entry point to a parameterization should be further documented with a documentation block immediately preceding its definition in the source. The documentation block should include at least the following components:

- A brief one- or two-sentence description with the `\brief` tag
- A more detailed one or two paragraph description of the function of the subroutine
- A comment indicating that metadata information about the subroutine arguments follows (in this example, the subroutine is called `SUBROUTINE_NAME`. Note that this line is also functional documentation used during the CCPP *prebuild* step.

```
!! \section arg_table_SUBROUTINE_NAME Argument Table
```

- A second comment indicating that a table of metadata to describe the subroutine arguments will be included from a separate file in HTML format (in this case, file `SUBROUTINE_NAME.html`). Please refer to the section below for information on how to generate the HTML files with metadata information from the `.meta` files.

  The argument table should be immediately followed by a blank doxygen line "!!".

```
!! \htmlinclude SUBROUTINE_NAME.html
!!
```

- A section called "General Algorithm" with a bullet or numbered list of the tasks completed in the subroutine algorithm
- At the end of initial subroutine documentation block, a "Detailed algorithm" section is started and the entirety of the code is encompassed with the `!> @{` and `!> @}` delimiters. This way, any comments explaining detailed aspects of the code are automatically included in the "Detailed Algorithm" section.

For subroutines that are not a CCPP entry point to a scheme, no inclusion of metadata information is required. But it is suggested that following `\ingroup` and `\brief`, use `\param` to define each argument with local name, a short description and unit, i.e.,

```
!>  \ingroup HEDMF
!!  \brief This subroutine is used for calculating the mass flux and updraft properties.
!!  ...
!!
!!  \param[in] im      integer, number of used points
!!  \param[in] ix      integer, horizontal dimension
!!  \param[in] km      integer, vertical layer dimension
!!  \param[in] ntrac   integer, number of tracers
```

(continues on next page)

(continued from previous page)

```
!!  \param[in] delt    real, physics time step
!!  ...
!!  \section general_mfpbl mfpbl General Algorithm
!!  -# Determine an updraft parcel's entrainment rate, buoyancy, and vertical velocity.
!!  -# Recalculate the PBL height ...
!!  -# Calculate the mass flux profile and updraft properties.
!!  \section detailed_mfpbl mfpbl Detailed Algorithm
!>  @{
       subroutine mfpbl(im,ix,km,ntrac,delt,cnvflg,                      &
       &   zl,zm,thvx,q1,t1,u1,v1,hpbl,kpbl,                             &
       &   sflx,ustar,wstar,xmf,tcko,qcko,ucko,vcko)
         ...
       end subroutine mfpbl
!>  @}
```

Bibliography

Doxygen can handle in-line paper citations and link to an automatically created bibliography page. The bibliographic data for any papers that are cited need to be put in BibTeX format and saved in a .bib file. The .bib file for CCPP is included in the *CCPP Physics* repository (`ccpp-physics/physics/docs/library.bib`), and the doxygen configuration option `cite_bib_files` points to the included file.

Citations are invoked with the following tag:

```
\cite bibtex_key_to_paper
```

Equations

See the Doxygen documentation for information about including equations. For the best rendering, the following option should be set in the *Doxygen configuration file*:

```
USE_MATHJAX            = YES
MATHJAX_RELPATH        = https://cdn.jsdelivr.net/npm/mathjax@2
```

There are many great online resources to use the LaTeX math typesetting used in doxygen.

2.8.3 Doxygen Configuration

Configuration File

The CCPP is distributed with a doxygen configuration file `ccpp-physics/physics/physics/docs/ccpp_doxyfile`, such that you don't need to create an additional one.

Doxygen files for layout (`ccpp_dox_layout.xml`), HTML style (`doxygen-awesome-ccpp.css`), and the bibliography (`library.bib`) are provided with the CCPP. Additionally, a configuration file is supplied, with the following variables modified from the default:

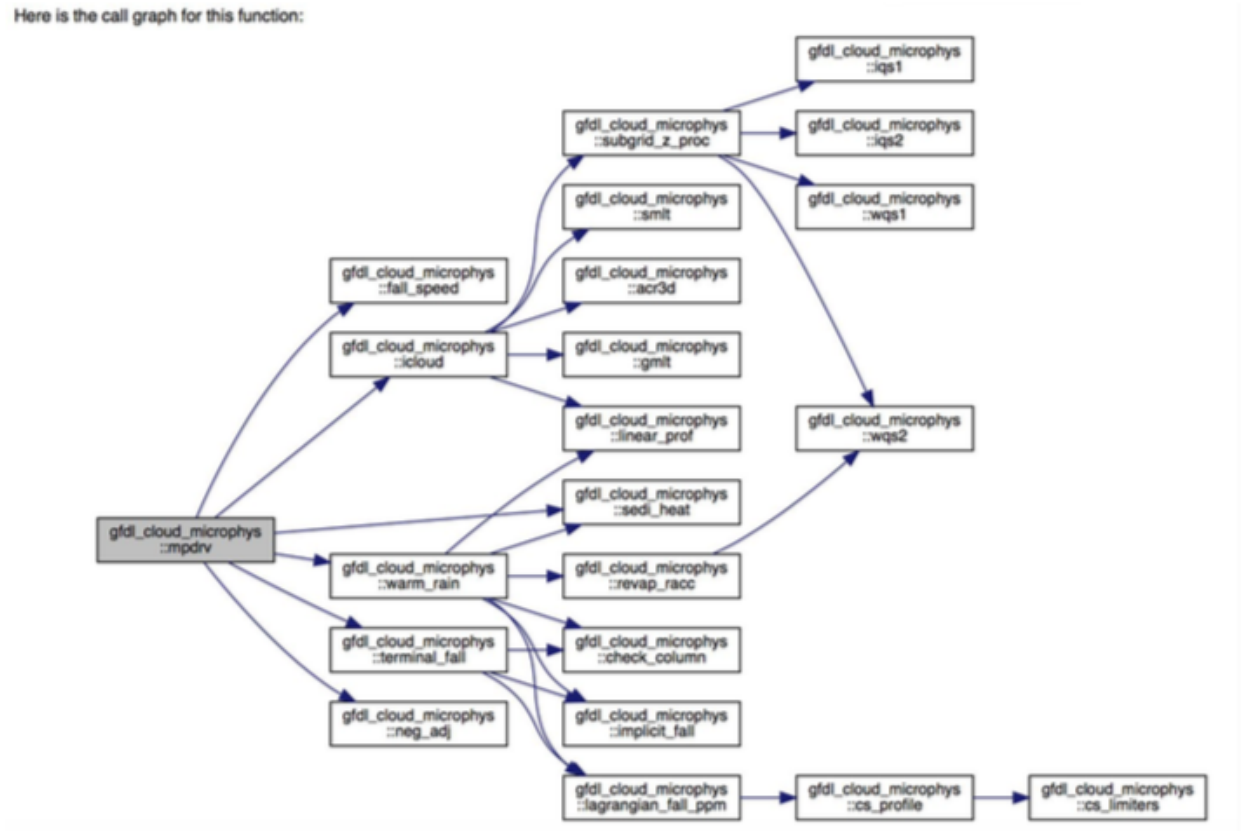
Diagrams

On its own, Doxygen is capable of creating simple text-based class diagrams. With the help of the additional software GraphViz, Doxygen can generate additional graphics-based diagrams, optionally in Unified Modeling Language (UML) style. To enable GraphViz support, the configure file parameter "HAVE_DOT" must be set to "YES".

You can use doxygen to create call graphs of all the physics schemes in CCPP. In order to create the call graphs you will need to set the following options in your doxygen config file:

```
HAVE_DOT               = YES
EXTRACT_ALL            = YES
EXTRACT_PRIVATE        = YES
EXTRACT_STATIC         = YES
CALL_GRAPH             = YES
```

Note that you will need the DOT (graph description language) utility to be installed when starting doxygen. Doxygen will call it to generate the graphs. On most distributions the DOT utility can be found in the GraphViz package. Here is the call graph for subroutine *mpdrv* in GFDL cloud microphysics generated by doxygen:



2.8.4 Including metadata information

As described above, a table of metadata information should be included in the documentation for every CCPP entrypoint scheme. Before doxygen is run, the table for each scheme must be manually created in separate files in HTML format, with one file per scheme. The HTML files are included in the Fortran files using the doygen markup below.

```
!! \htmlinclude SUBROUTINE_NAME.html
!!
```

The tables should be created using a Python script distributed with the *CCPP Framework*, `ccpp-framework/scripts/metadata2html.py`.

Note: You will need to set the environment variable `PYTHONPATH` to include the directories `ccpp/framework/scripts` and `ccpp/framework/scripts/parse_tools`. As an example for bash-like shells:

```
export PYTHONPATH=`pwd`/ccpp/framework/scripts:`pwd`/ccpp/framework/scripts/parse_tools
```

For the example of the *SCM*, where both scripts need to be called from the *host model* top-level directory:

```
./ccpp/framework/scripts/metadata2html.py -m ccpp/physics/physics/file.meta -o ccpp/
↪physics/physics/docs
```

where `-m` is used to specify a file with metadata information and `-o` is used to specify the directory for output. Note that a single input file (`.meta`) may have more than one CCPP entrypoint scheme, and therefore can be used to generate more than one HTML file.

Note that the `.meta` files are supplied in the CCPP Physics repository, and that there is a `.meta` file for each Fortran file that contains one or more CCPP entrypoint scheme. The `.meta` files are located in the same directory as the scheme Fortran files (`ccpp-physics/physics`).

To generate the complete Scientific Documentation, the script `./ccpp/framework/scripts/metadata2html.py` must be run separately for each `.meta` file available in `ccpp-physics/physics`. Alternatively, a batch mode exists that converts all metadata files associated with schemes and variable definitions in the CCPP prebuild config; again using the SCM as an example:

```
./ccpp/framework/scripts/metadata2html.py -c ccpp/config/ccpp_prebuild_config.py
```

Note that the options `-c` and `-m` are mutually exclusive, but that one of them is required. The option `-m` also requires the user to specify `-o`, while the option `-c` will ignore `-o`. For more information, use

```
./ccpp/framework/scripts/metadata2html.py --help
```

2.8.5 Using Doxygen

In order to generate the doxygen-based documentation, you will need to follow five steps:

1. Have the executables `doxygen` (https://doxygen.nl/), `graphviz` (https://graphviz.org/), and `bibtex` (http://www.bibtex.org/) installed on your machine and in your `PATH`. These utilities can be installed on MacOS via Homebrew, or installed manually via the instructions on each utility's page linked above.
2. Document your code, including the doxygen main page, scheme pages, and inline comments within the source code as described above.
3. Run `metadata2html.py` to create files in HTML format containing metadata information for each CCPP entrypoint scheme.

4. Prepare a Bibliography file in BibTex format for papers or other references cited in the physics suites.
5. Create or edit a doxygen configuration file to control which doxygen pages, source files, and bibliography get parsed, in addition to how the source files get parsed, and to customize the output.
6. Run doxygen from the directory `ccpp/physics/physics/docs` using the command line to specify the doxygen configuration file as an argument. For the CCPP Scientific documentation, this file is called `ccpp_doxyfile`:

   ```
   doxygen ccpp_doxyfile
   ```

   Running this command may generate warnings or errors that need to be fixed in order to produce proper output. The location and type of output (HTML, LaTeX, etc.) are specified in the configuration file. The generated HTML documentation can be viewed by pointing an HTML browser to the `index.html` file in the `./docs/doc/html/` directory.

For precise instructions or other help creating the scientific documentation, visit the CCPP GitHub discussions page at https://github.com/NCAR/ccpp-physics/discussions

CHAPTER

THREE

CCPP CONFIGURATION AND BUILD OPTIONS

While the *CCPP Framework* code, consisting of a single Fortran source file and associated metadata file, can be compiled and tested independently, the *CCPP Physics* code can only be used within a *host modeling* system that provides the variables required to execute the physics. As such, it is advisable to integrate the CCPP configuration and build process with the host model build system. Part of the build process, known as the *prebuild* step since it precedes compilation, involves running a Python script that performs multiple functions. These functions include configuring the *CCPP Physics* for use with the host model and autogenerating FORTRAN code to communicate variables between the physics and the dynamical core. The *prebuild* step will be discussed in detail in Chapter 8.

The *SCM* and the *UFS Atmosphere* are supported for use with the CCPP. In the case of the UFS Atmosphere as the host model, build configuration options can be specified as cmake options to the `build.sh` script for manual compilation or through a regression test (RT) configuration file. Detailed instructions for building the UFS Atmosphere and the SCM are discussed in the UFS Weather Model User Guide and the SCM User Guide. For both SCM and UFS the `ccpp_prebuild.py` script is run automatically as a step in the build system, although it can be run manually for debugging purposes.

The path to a host-model specific configuration file is the only required argument to `ccpp_prebuild.py`. Such files are included with the ccpp-scm and ufs-weather-model repositories, and must be included with the code of any host model to use the CCPP. Figure 3.1 depicts the main functions of the `ccpp_prebuild.py` script for the build. Using information included in the configuration file and the *SDF*(s), the script parses the SDF(s) and only matches provided/requested variables that are used within the particular physics suite(s). The script autogenerates software *caps* for the physics suite(s) as a whole and for each physics *group* as defined in the SDF(s), as well as for an API that the host model calls into from the (manually written) host model cap. At runtime, a single SDF is used to select the suite that will be executed in the run. This arrangement allows for efficient variable recall (which is done once for all physics *schemes* within each group of a suite), leads to a reduced memory footprint of the CCPP, and speeds up execution.

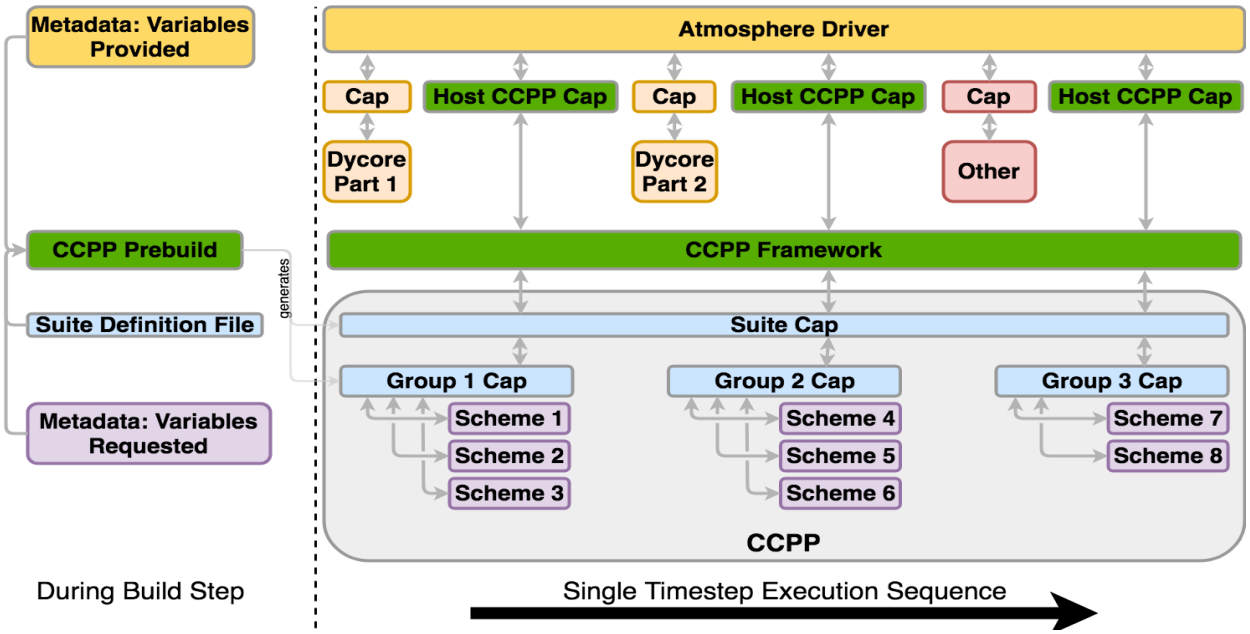


Fig. 3.1: *This figure depicts an example of the interaction between an atmospheric model and CCPP Physics for one timestep, and a single SDF, with execution progressing toward the right. The "Atmosphere Driver" box represents model superstructure code, perhaps responsible for I/O, time-stepping, and other model component interactions. Software caps are autogenerated for the suite and physics groups, defined in the SDF provided to the* `ccpp_prebuild.py` *script. The suite must be defined via the SDF at prebuild time. When multiple SDFs are provided during the build step, multiple suite caps and associated group caps are produced, but only one is used at runtime.*

CHAPTER

FOUR

CONSTRUCTING SUITES

4.1 Suite Definition File

The *SDF* is a file in XML format used to specify the name of the suite, the physics *schemes* to run, *groups* of physics that run together, the order in which to run the physics, and whether *subcycling* will be used to run any of the *parameterizations* with shorter timesteps. The SDF files are part of the *host model* code.

In addition to the *primary parameterization* categories (such as radiation, boundary layer, deep convection, resolved moist physics, etc.), the SDF can have an arbitrary number of *interstitial scheme*s in between the parameterizations to preprocess or postprocess data. In many models, this interstitial code is not obvious to the model user but, with the SDF, both the primary parameterizations and the interstitial schemes are listed explicitly.

The format of the SDF is specified by a schema and all host models that use *CCPP* include file `suite.xsd` to describe the schema.

The name of the suite is listed at the top of the SDF, right after the XML declaration, and must be consistent with the name of the SDF: file `suite_ABC.xml` contains `suite name='ABC'`, as in the example below. The suite name is followed by the version of the XML schema used.

4.1.1 Groups

The concept of grouping physics in the SDF (reflected in the `<group name="XYZ">` elements) enables *groups* of parameterizations to be called with other computation (such as related to the dycore, I/O, etc.) in between. One can edit the groups to suit the needs of the host application. For example, if a subset of physics schemes needs to be more tightly connected with the dynamics and called more frequently, one could create a group consisting of that subset and place a `ccpp_physics_run` call in the appropriate place in the host application. The remainder of the parameterization groups could be called using `ccpp_physics_run` calls in a different part of the host application code.

4.1.2 Subcycling

The SDF allows subcycling of schemes, or calling a subset of schemes at a smaller time step than others. The `<subcycle loop = n>` element in the SDF controls this function. All schemes within such an element are called `n` times during one `ccpp_physics_run` call. An example of this is found in the `suite_FV3_GFS_v16.xml` SDF, where the surface schemes are executed twice for each timestep (implementing a predictor/corrector paradigm):

```
<!-- Surface iteration loop -->
<subcycle loop="2">
  <scheme>sfc_diff</scheme>
  <scheme>GFS_surface_loop_control_part1</scheme>
  <scheme>sfc_nst_pre</scheme>
```

(continues on next page)

(continued from previous page)

```
  <scheme>sfc_nst</scheme>
  <scheme>sfc_nst_post</scheme>
  <scheme>lsm_noah</scheme>
  <scheme>sfc_sice</scheme>
  <scheme>GFS_surface_loop_control_part2</scheme>
</subcycle>
```

Note that currently no time step information is included in the SDF and that the subcycling of schemes resembles more an iteration over schemes with the loop counter being available as integer variable with *standard name* `ccpp_loop_counter`. If subcycling is used for a set of parameterizations, the smaller time step must be an input argument for those schemes, or computed in the scheme from the default physics time step (`timestep_for_physics`) and the number of subcycles (`ccpp_loop_extent`).

4.1.3 Order of Schemes

Schemes may be interdependent and the order in which the schemes are run may make a difference in the model output. Reading the SDF(s) and defining the order of schemes for each suite happens at compile time. Some schemes require additional interstitial code that must be run before or after the scheme and cannot be part of the scheme itself. This can be due to dependencies on other schemes and/or the order of the schemes as determined in the SDF. Note that more than one SDF can be supplied at compile time, but only one can be used at runtime.

4.2 Interstitial Schemes

The SDF can have an arbitrary number of additional interstitial schemes in between the primary parameterizations to preprocess or postprocess data. There are two main types of interstitial schemes, scheme-specific and suite-level. The scheme-specific interstitial scheme is needed for one specific scheme and the suite-level interstitial scheme processes data that are relevant for various schemes within a suite.

4.3 SDF Examples

4.3.1 Simplest Case: Single Group and no Subcycling

Consider the simplest case, in which all physics schemes are to be called together in a single group with no subcycling (i.e. `subcycle loop="1"`). The subcycle loop must be set in each group. The SDF `suite_Suite_A.xml` could contain the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<suite name="Suite_A" ver="1">
  ...
  <group name="physics">
    <subcycle loop="1">
      <scheme>Suite_A_interstitial_1</scheme>
      <scheme>scheme_1_pre</scheme>
      <scheme>scheme_1</scheme>
      <scheme>scheme_1_post</scheme>
      <scheme>scheme_2_generic_pre</scheme>
      <scheme>scheme_2</scheme>
```

(continues on next page)

(continued from previous page)

```
      <scheme>scheme_2_generic_post</scheme>
      <scheme>Suite_A_interstitial_2</scheme>
      <scheme>scheme_3</scheme>
      ...
      <scheme_n</scheme>
    </subcycle>
  </group>
</suite>
```

Note the syntax of the SDF. The root (the first element to appear in the xml file) is the `suite` with the `name` of the suite given as an attribute. In this example, the suite name is `Suite_A`. Within each suite are groups, which specify a physics group to call (i.e. `physics`, `fast_physics`, `time_vary`, `radiation`, `stochastics`). Each group has an option to subcycle. The value given for loop determines the number of times all of the schemes within the `subcycle` element are called. Finally, the `scheme` elements are children of the `subcycle` elements and are listed in the order they will be executed. In this example, `scheme_1_pre` and `scheme_1_post` are scheme-specific preprocessing and postprocessing interstitial schemes, respectively. The suite-level preprocessing and postprocessing interstitial `schemes scheme_2_generic_pre` and `scheme_2_generic_post` are also called in this example. `Suite_A_interstitial_2` is a scheme for `suite_A` and connects various schemes within this suite.

4.3.2 Case with Multiple Groups

Some models require that the physics be called in groups, with non-physics computations in-between the groups.

```
<?xml version="1.0" encoding="UTF-8"?>

<suite name="Suite_B" ver="1">
  <group name="g1">
    <subcycle loop="1">
      <scheme>SchemeX</scheme>
      <scheme>SchemeY</scheme>
      <scheme>SchemeZ</scheme>
    </subcycle>
  </group>
  <group name="g2">
    <subcycle loop="1">
      <scheme>SchemeA</scheme>
      <scheme>SchemeB</scheme>
      <scheme>SchemeC</scheme>
    </subcycle>
  </group>
</suite>
```

4.3.3 Case with Subcycling

Consider the case where a model requires that some subset of physics be called on a smaller time step than the rest of the physics, e.g. for computational stability. In this case, one would make use of the subcycle element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<suite name="Suite_C" ver="1">
  <group name="g1">
    <subcycle loop="1">
      <scheme>scheme_1</scheme>
      <scheme>scheme_2</scheme>
    </subcycle>
    <subcycle loop="2">
      <!-- block of schemes 3 and 4 is called twice -->
      <scheme>scheme_3</scheme>
      <scheme>scheme_4</scheme>
    </subcycle>
  </group>
</suite>
```

4.3.4 GFS v16beta Suite

Here is the SDF for the physics suite equivalent to the GFS v16beta in the Single Column Model (*SCM*), which employs various groups and subcycling:

```
<?xml version="1.0" encoding="UTF-8"?>

<suite name="SCM_GFS_v16beta" version="1">
  <group name="time_vary">
    <subcycle loop="1">
      <scheme>GFS_time_vary_pre</scheme>
      <scheme>GFS_rrtmg_setup</scheme>
      <scheme>GFS_rad_time_vary</scheme>
      <scheme>GFS_phys_time_vary</scheme>
    </subcycle>
  </group>
  <group name="radiation">
    <subcycle loop="1">
      <scheme>GFS_suite_interstitial_rad_reset</scheme>
      <scheme>GFS_rrtmg_pre</scheme>
      <scheme>rrtmg_sw_pre</scheme>
      <scheme>rrtmg_sw</scheme>
      <scheme>rrtmg_sw_post</scheme>
      <scheme>rrtmg_lw_pre</scheme>
      <scheme>rrtmg_lw</scheme>
      <scheme>rrtmg_lw_post</scheme>
      <scheme>GFS_rrtmg_post</scheme>
    </subcycle>
  </group>
  <group name="physics">
    <subcycle loop="1">
```

(continues on next page)

(continued from previous page)

```
    <scheme>GFS_suite_interstitial_phys_reset</scheme>
    <scheme>GFS_suite_stateout_reset</scheme>
    <scheme>get_prs_fv3</scheme>
    <scheme>GFS_suite_interstitial_1</scheme>
    <scheme>GFS_surface_generic_pre</scheme>
    <scheme>GFS_surface_composites_pre</scheme>
    <scheme>dcyc2t3</scheme>
    <scheme>GFS_surface_composites_inter</scheme>
    <scheme>GFS_suite_interstitial_2</scheme>
  </subcycle>
  <!-- Surface iteration loop -->
  <subcycle loop="2">
    <scheme>sfc_diff</scheme>
    <scheme>GFS_surface_loop_control_part1</scheme>
    <scheme>sfc_nst_pre</scheme>
    <scheme>sfc_nst</scheme>
    <scheme>sfc_nst_post</scheme>
    <scheme>lsm_noah</scheme>
    <scheme>sfc_sice</scheme>
    <scheme>GFS_surface_loop_control_part2</scheme>
  </subcycle>
  <!-- End of surface iteration loop -->
  <subcycle loop="1">
    <scheme>GFS_surface_composites_post</scheme>
    <scheme>sfc_diag</scheme>
    <scheme>sfc_diag_post</scheme>
    <scheme>GFS_surface_generic_post</scheme>
    <scheme>GFS_PBL_generic_pre</scheme>
    <scheme>satmedmfvdifq</scheme>
    <scheme>GFS_PBL_generic_post</scheme>
    <scheme>GFS_GWD_generic_pre</scheme>
    <scheme>cires_ugwp</scheme>
    <scheme>cires_ugwp_post</scheme>
    <scheme>GFS_GWD_generic_post</scheme>
    <scheme>rayleigh_damp</scheme>
    <scheme>GFS_suite_stateout_update</scheme>
    <scheme>ozphys_2015</scheme>
    <scheme>h2ophys</scheme>
    <scheme>get_phi_fv3</scheme>
    <scheme>GFS_suite_interstitial_3</scheme>
    <scheme>GFS_DCNV_generic_pre</scheme>
    <scheme>samfdeepcnv</scheme>
    <scheme>GFS_DCNV_generic_post</scheme>
    <scheme>GFS_SCNV_generic_pre</scheme>
    <scheme>samfshalcnv</scheme>
    <scheme>GFS_SCNV_generic_post</scheme>
    <scheme>GFS_suite_interstitial_4</scheme>
    <scheme>cnvc90</scheme>
    <scheme>GFS_MP_generic_pre</scheme>
    <scheme>gfdl_cloud_microphys</scheme>
    <scheme>GFS_MP_generic_post</scheme>
    <scheme>maximum_hourly_diagnostics</scheme>
```

(continues on next page)

(continued from previous page)

```
      <scheme>phys_tend</scheme>
    </subcycle>
  </group>
</suite>
```

The suite name is `SCM_GFS_v16beta`. Three groups (`time_vary`, `radiation`, and `physics`) are used, because the physics needs to be called in different parts of the host model. The detailed explanation of each primary physics scheme can be found in scientific documentation. A short explanation of each scheme is below.

- `GFS_time_vary_pre`: GFS physics suite time setup
- `GFS_rrtmg_setup`: Rapid Radiative Transfer Model for Global Circulation Models (RRTMG) setup
- `GFS_rad_time_vary`: GFS radiation time setup
- `GFS_phys_time_vary`: GFS physics suite time setup
- `GFS_suite_interstitial_rad_reset`: GFS suite interstitial radiation reset
- `GFS_rrtmg_pre`: Preprocessor for the GFS radiation schemes
- `rrtmg_sw_pre`: Preprocessor for the RRTMG shortwave radiation
- `rrtmg_sw`: RRTMG for shortwave radiation
- `rrtmg_sw_post`: Postprocessor for the RRTMG shortwave radiation
- `rrtmg_lw_pre`: Preprocessor for the RRTMG longwave radiation
- `rrtmg_lw`: RRTMG for longwave radiation
- `rrtmg_lw_post`: Postprocessor for the RRTMG longwave radiation
- `GFS_rrtmg_post`: Postprocessor for the GFS radiation schemes
- `GFS_suite_interstitial_phys_reset`: GFS suite interstitial physics reset
- `GFS_suite_stateout_reset`: GFS suite stateout reset
- `get_prs_fv3`: Adjustment of the geopotential height hydrostatically in a way consistent with FV3 discretization
- `GFS_suite_interstitial_1`: GFS suite interstitial 1
- `GFS_surface_generic_pre`: Preprocessor for the surface schemes (land, sea ice)
- `GFS_surface_composites_pre`: Preprocessor for surafce composites
- `dcyc2t3`: Mapping of the radiative fluxes and heating rates from the coarser radiation timestep onto the model's more frequent time steps
- `GFS_surface_composites_inter`: Interstitial for the surface composites
- `GFS_suite_interstitial_2`: GFS suite interstitial 2
- `sfc_diff`: Calculation of the exchange coefficients in the GFS surface layer
- `GFS_surface_loop_control_part1`: GFS surface loop control part 1
- `sfc_nst_pre`: Preprocessor for the near-surface sea temperature
- `sfc_nst`: GFS Near-surface sea temperature
- `sfc_nst_post`: Postprocessor for the near-surface temperature
- `lsm_noah`: Noah land surface scheme driver
- `sfc_sice`: Simple sea ice scheme

- GFS_surface_loop_control_part2: GFS surface loop control part 2
- GFS_surface_composites_post: Postprocess for surface composites
- sfc_diag: Land surface diagnostic calculation
- sfc_diag_post: Postprocessor for the land surface diagnostic calculation
- GFS_surface_generic_post: Postprocessor for the GFS surface process
- GFS_PBL_generic_pre: Preprocessor for all Planetary Boundary Layer (PBL) schemes (except MYNN)
- satmedmfvdifq: Scale-aware TKE-based moist eddy-diffusion mass-flux
- GFS_PBL_generic_post: Postprocessor for all Planetary Boundary Layer (PBL) schemes (except MYNN)
- GFS_GWD_generic_pre: Preprocessor for the orographic gravity wave drag
- cires_ugwp: Unified gravity wave drag
- cires_ugwp_post: Postprocessor for the unified gravity wave drag
- GFS_GWD_generic_post: Postprocessor for the GFS gravity wave drag
- rayleigh_damp: Rayleigh damping
- GFS_suite_stateout_update: GFS suite stateout update
- ozphys_2015: Ozone photochemistry
- h2ophys: H2O physics for stratosphere and mesosphere
- get_phi_fv3: Hydrostatic adjustment to the height in a way consistent with FV3 discretization
- GFS_suite_interstitial_3: GFS suite interstitial 3
- samfdeepcnv: Simplified Arakawa Schubert (SAS) Mass Flux deep convection
- GFS_DCNV_generic_post: Postprocessor for all deep convective schemes
- GFS_SCNV_generic_pre: Preprocessor for the GFS shallow convective schemes
- samfshalcnv: SAS mass flux shallow convection
- GFS_SCNV_generic_post: Postprocessor for the GFS shallow convective scheme
- GFS_suite_interstitial_4: GFS suite interstitial 4
- cnvc90: Convective cloud cover
- GFS_MP_generic_pre: Preprocessor for all GFS microphysics
- gfdl_cloud_microphys: GFDL cloud microphysics
- GFS_MP_generic_post: Postprocessor for GFS microphysics
- maximum_hourly_diagnostics: Computation of the maximum of the selected diagnostics
- phys_tend: Physics tendencies

CHAPTER

FIVE

SUITE AND GROUP *CAPS*

The connection between the *host model* and the physics *schemes* through the *CCPP Framework* is realized with *caps* on both sides as illustrated in Figure 1.1. The CCPP *prebuild* script discussed in Chapter 3 generates the *caps* that connect the physics schemes to the CCPP Framework. This chapter describes the *suite* and *group caps*, while the host model *caps* are described in Chapter 6. These *caps* autogenerated by `ccpp_prebuild.py` reside in the directory defined by the `CAPS_DIR` variable (see example in *Listing 8.1*).

5.1 Overview

When CCPP is built, the CCPP Framework and physics are statically linked to the executable. This allows the best performance and efficient memory use. This build requires metadata provided by the host model and variables requested from the physics scheme. Only the variables required for the specified suites are kept, requiring one or more *SDF*s (see left side of Figure 3.1) as arguments to the `ccpp_prebuild.py` script. The CCPP *prebuild* step performs the tasks below.

- Check requested vs provided variables by `standard_name`.
- Check units, rank, type. Perform unit conversions if a mismatch of units is detected and the required conversion has been implemented (see Section 5.2 for details).
- Filter unused schemes and variables.
- Create Fortran code for the static Application Programming Interface (API).
- Create *caps* for groups and suite(s).
- Populate makefiles with schemes and *caps*.

The *prebuild* step will produce the following files for any host model. Note that the location of these files varies between the host models and whether an in-source or out-of-source build is used.

- List of variables provided by host model and required by physics:

```
CCPP_VARIABLES_FV3.tex
```

- cmake/gnumake snippets and shell script that contain all *caps* to be compiled:

```
CCPP_CAPS.{cmake,mk,sh}
```

- cmake/gnumake snippets and shell script that contain all schemes to be compiled:

```
CCPP_SCHEMES.{cmake,mk,sh}
```

- List of CCPP types:

```
CCPP_TYPEDEFS.{cmake,mk,sh}
```

- List of variables provided by host model:

```
CCPP_VARIABLES_FV3.html
```

- One *cap* per physics group (fast_physics, physics, radiation, time_vary, stochastic, ...) for each suite:

```
ccpp_{suite_name}_{group_name}_cap.F90
```

- *Cap* for each suite:

```
ccpp_{suite_name}_cap.F90
```

- Autogenerated API (aka CCPP Framework).

```
ccpp_static_api.F90
```

`ccpp_static_api.F90` is an interface, which contains subroutines `ccpp_physics_init`, `ccpp_physics_timestep_init`, `ccpp_physics_run`, `ccpp_physics_timestep_finalize`, and `ccpp_physics_finalize`. Each subroutine uses a `suite_name` and an optional argument, `group_name`, to call the groups of a specified suite (e.g. `fast_physics`, `physics`, `time_vary`, `radiation`, `stochastic`, etc.), or to call the entire suite. For example, `ccpp_static_api.F90` would contain module `ccpp_static_api` with subroutines `ccpp_physics_{init, timestep_init, run, timestep_finalize, finalize}`. Interested users should run `ccpp_prebuild.py` as appropriate for their model and inspect these auto-generated files.

5.2 Automatic unit conversions

The CCPP framework is capable of performing automatic unit conversions if a mismatch of units between the host model and a physics scheme is detected, provided that the required unit conversion has been implemented.

If a mismatch of units is detected and an automatic unit conversion can be performed, the CCPP prebuild script will document this with a log message as in the following example:

```
INFO: Comparing metadata for requested and provided variables ...
INFO: Automatic unit conversion from m to um for effective_radius_of_stratiform_cloud_
↪ice_particle_in_um after returning from MODULE_mp_thompson SCHEME_mp_thompson␣
↪SUBROUTINE_mp_thompson_run
INFO: Automatic unit conversion from m to um for effective_radius_of_stratiform_cloud_
↪liquid_water_particle_in_um after returning from MODULE_mp_thompson SCHEME_mp_thompson␣
↪SUBROUTINE_mp_thompson_run
INFO: Automatic unit conversion from m to um for effective_radius_of_stratiform_cloud_
↪snow_particle_in_um after returning from MODULE_mp_thompson SCHEME_mp_thompson␣
↪SUBROUTINE_mp_thompson_run
INFO: Generating schemes makefile/cmakefile snippet ...
```

The CCPP framework is performing only the minimum unit conversions necessary, depending on the intent information of the variable in the *parameterization*'s metadata table. In the above example, the cloud effective radii are `intent(out)` variables, which means that no unit conversion is required before entering the subroutine `mp_thompson_run`. Therefore, it is imperative to use the correct value for the `intent` attribute in the metadata. A common pitfall is to declare a variable as `intent(out)`, and then fail to guarantee to completely overwrite the contents of the variable in the file. Below are examples for auto-generated code performing automatic unit conversions

from `m` to `um` or back, depending on the intent of the variable. The conversions are performed in the individual physics scheme caps for the dynamic build, or the group caps for the build.

```
! var1 is intent(in)
        call mp_thompson_run(...,recloud=1.0E-6_kind_phys*re_cloud,...,errmsg=cdata
↪%errmsg,errflg=cdata%errflg)
        ierr=cdata%errflg

! var1 is intent(inout)
        allocate(tmpvar1, source=re_cloud)
        tmpvar1 = 1.0E-6_kind_phys*re_cloud
        call mp_thompson_run(...,re_cloud=tmpvar1,...,errmsg=cdata%errmsg,errflg=cdata
↪%errflg)
        ierr=cdata%errflg
        re_cloud = 1.0E+6_kind_phys*tmpvar1
        deallocate(tmpvar1)

! var1 is intent(out)
        allocate(tmpvar1, source=re_cloud)
        call mp_thompson_run(...,re_cloud=tmpvar1,...,errmsg=cdata%errmsg,errflg=cdata
↪%errflg)
        ierr=cdata%errflg
        re_cloud = 1.0E+6_kind_phys*tmpvar1
        deallocate(tmpvar1)
```

If a required unit conversion has not been implemented the CCPP prebuild script will generate an error message as follows:

```
INFO: Comparing metadata for requested and provided variables ...
ERROR: Error, automatic unit conversion from m to pc for effective_radius_of_stratiform_
↪cloud_ice_particle_in_um in MODULE_mp_thompson SCHEME_mp_thompson SUBROUTINE_mp_
↪thompson_run not implemented
```

All automatic unit conversions are implemented in `ccpp-framework/scripts/conversion_tools/unit_conversion.py`, new unit conversions can be added to this file by following the existing examples.

CHAPTER

SIX

HOST SIDE CODING

This chapter describes the connection of a host model with the pool of *CCPP Physics schemes* through the *CCPP Framework*.

6.1 Variable Requirements on the Host Model Side

All variables required to communicate between the host model and the physics, as well as to communicate between physics schemes, need to be allocated by the host model. An exception is variables `errflg`, `errmsg`, `loop_cnt`, `loop_max`, `blk_no`, and `thrd_no`, which are allocated by the CCPP Framework, as explained in Section 6.4.1. See Section 2.3 for information about the variables required for the current pool of CCPP physics.

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 DDTs as arguments to physics schemes in general is discouraged (see Section 2.4), it is perfectly acceptable for the host model to define the variables requested by physics schemes as components of DDTs and pass these components to CCPP by using the correct local_name (e.g., `myddt%thecomponentIwant`; see Section 6.2.)

6.2 Metadata for Variables in the Host Model

To establish the link between host model variables and physics scheme variables, the host model must provide metadata information similar to those presented in Section 2.2. The host model can have multiple metadata files (`.meta`), each with the required `[ccpp-table-properties]` section and the related `[ccpp-arg-table]` sections. The host model Fortran files contain three-line snippets to indicate the location for insertion of the metadata information contained in the corresponding section in the `.meta` file.

```
!!> \section arg_table_example_vardefs
!! \htmlinclude example_vardefs.html
!!
```

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 metadata for variables in the host model (see also *Listing 6.1* and *Listing 6.2* for examples of host model metadata).

- 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 must match those in the physics scheme metadata.
- The attribute `active` is used to allocate variables under certain conditions. It must be written as a Fortran expression that equates to `.true.` or `.false.`, using the CCPP standard names of variables. `active` attributes for all variables are `.true.` by default. See Section 6.2.2 for details.
- The `intent` attribute is not a valid attribute for host model metadata and will be ignored, if present.
- The `local_name` of the variable must be set to the name the host model cap uses to refer to the variable.
- The metadata section that exposes a DDT to the CCPP (as opposed to the section that describes the components of a DDT) must be in the same module where the memory for the DDT is allocated. If the DDT is a module variable, then it must be exposed via the module's metadata section, which must have the same name as the module.
- Metadata sections describing module variables must be placed inside the module.
- Metadata sections describing components of DDTs must be placed immediately before the type definition and have the same name as the DDT.

```
    module example_vardefs

      implicit none

!!> \section arg_table_example_vardefs
!! \htmlinclude example_vardefs.html
!!

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

!!> \section arg_table_example_ddt
!! \htmlinclude example_ddt.html
!!

      type ex_ddt
        logical              :: l
        real, dimension(:,:) :: r
      end type ex_ddt

      type(ex_ddt) :: ext

    end module example_vardefs
```

Listing 6.1: Example host model file with reference to metadata. In this example, both the definition and the declaration (memory allocation) of a DDT `ext` *(of type* `ex_ddt` *) are in the same module.*

```
########################################################################
[ccpp-table-properties]
```

(continues on next page)

(continued from previous page)

```
  name = arg_table_example_vardefs
  type = module

[ccpp-arg-table]
  name = arg_table_example_vardefs
  type = module
[ex_int]
  standard_name = example_int
  long_name = ex. int
  units = none
  dimensions = ()
  type = integer
[ex_real]
  standard_name = example_real
  long_name = ex. real
  units = m
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = kind=8
[ex_ddt]
  standard_name = example_ddt
  long_name = ex. ddt
  units = DDT
  dimensions = ()
  type = ex_ddt
[ext]
  standard_name = example_ddt_instance
  long_name = ex. ddt inst
  units = DDT
  dimensions = ()
  type = ex_ddt
[errmsg]
  standard_name = ccpp_error_message
  long_name = error message for error handling in CCPP
  units = none
  dimensions = ()
  type = character
  kind = len=64
[errflg]
  standard_name = ccpp_error_code
  long_name = error code for error handling in CCPP
  units = 1
  dimensions = ()
  type = integer

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
[ccpp-table-properties]
  name = arg_table_example_ddt
  type = ddt

[ccpp-arg-table]
  name = arg_table_example_ddt
```

(continues on next page)

(continued from previous page)

```
  type = ddt
[ext%1]
  standard_name = example_flag
  long_name = ex. flag
  units = flag
  dimensions =
  type = logical
[ext%r]
  standard_name = example_real3
  long_name = ex. real
  units = kg
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = r15
[ext%r(:,1)]
  standard_name = example_slice
  long_name = ex. slice
  units = kg
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = r15
[nwfa2d]
  standard_name = tendency_of_water_friendly_aerosols_at_surface
  long_name = instantaneous water-friendly sfc aerosol source
  units = kg-1 s-1
  dimensions = (horizontal_loop_extent)
  type = real
  kind = kind_phys
  active = (flag_for_microphysics_scheme == flag_for_thompson_microphysics_scheme .and.␣
↪flag_for_aerosol_physics)
[qgrs(:,:,index_for_water_friendly_aerosols)]
  standard_name = water_friendly_aerosol_number_concentration
  long_name = number concentration of water-friendly aerosols
  units = kg-1
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  active = (index_for_water_friendly_aerosols > 0)
  type = real
  kind = kind_phys
```

Listing 6.2: Example host model metadata file (.meta).

6.2.1 horizontal_dimension vs. horizontal_loop_extent

Please refer to section Section 2.2.3 for a description of the differences between `horizontal_dimension` and `horizontal_loop_extent`. The host model must define both variables to represent the horizontal dimensions in use by the physics in the metadata.

For the examples in listing *Listing 6.2*, the host model stores all horizontal grid columns of each variable in one contiguous block, and the variables `horizontal_dimension` and `horizontal_loop_extent` are identical. Alternatively, a host model could store (non-contiguous) blocks of data in an array of DDTs with a length of the total number of blocks, as shown in listing *Listing 6.3*. Figure 3.1 depicts the differences in variable allocation for these two cases.

```
########################################################################
[ccpp-table-properties]
  name = arg_table_example_vardefs
  type = module

[ccpp-arg-table]
  name = arg_table_example_vardefs
  type = module
...
[ex_ddt]
  standard_name = example_ddt
  long_name = ex. ddt
  units = DDT
  dimensions = ()
  type = ex_ddt
[ext(ccpp_block_number)]
  standard_name = example_ddt_instance
  long_name = ex. ddt inst
  units = DDT
  dimensions = ()
  type = ex_ddt
[ext]
  standard_name = example_ddt_instance_all_blocks
  long_name = ex. ddt inst
  units = DDT
  dimensions = (ccpp_block_count)
  type = ex_ddt
...

########################################################################
[ccpp-table-properties]
  name = arg_table_example_ddt
  type = ddt

[ccpp-arg-table]
  name = arg_table_example_ddt
  type = ddt
[ext%1]
  standard_name = example_flag
  long_name = ex. flag
  units = flag
  dimensions =
  type = logical
[ext%r]
  standard_name = example_real3
  long_name = ex. real
  units = kg
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = r15
...
```

Listing 6.3: Example host model metadata file (`.meta`) for a host model using blocked data structures.

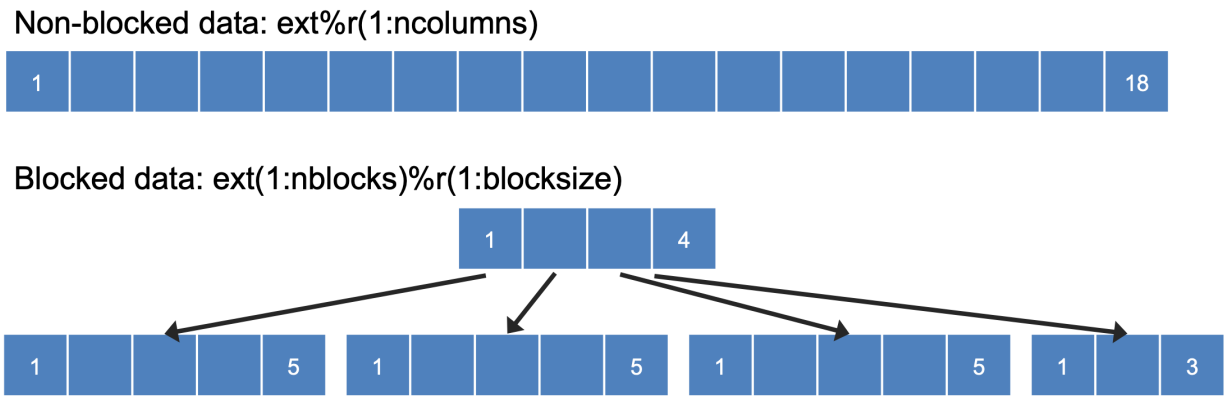


Fig. 6.1: *This figure depicts the difference between non-blocked (contiguous) and blocked data structures.*

When blocked data structures are used by the host model, `horizontal_loop_extent` corresponds to the block size, and the sum of all block sizes equals `horizontal_dimension`. In either case, the correct horizontal dimension for host model variables is `horizontal_loop_extent`. In the time integration (run) *phase*, the physics are called for one block at a time (although possibly in parallel using OpenMP threading). In all other phases, the CCPP Framework automatically combines the discontiguous blocked data into contiguous arrays before calling into a physics scheme, as shown in *Listing 6.4*.

```
allocate(bar_local(1:ncolumns))
ib = 1
do nb=1,nblocks
  bar_local(ib:ib+blocksize(nb)-1) = foo(nb)%bar
  ib = ib+blocksize(nb)
end do

call myscheme_init(bar=bar_local)

ib = 1
do nb=1,nblocks
  foo(nb)%bar = bar_local(ib:ib+blocksize(nb)-1)
  ib = ib+blocksize(nb)
end do
deallocate(bar_local)
```

Listing 6.4: Automatic combination of blocked data structures in the auto-generated caps

6.2.2 Active Attribute

The CCPP must be able to detect when arrays need to be allocated, and when certain tracers must be present in order to perform operations or tests in the auto-generated caps (e.g. unit conversions, blocked data structure copies, etc.). This is accomplished with the attribute `active` in the metadata for the host model variables (e.g., `GFS_typedefs.meta` for the *UFS Atmosphere* or the *SCM*).

Several arrays in the host model (e.g., `GFS_typedefs.F90` in the UFS Atmosphere or the SCM) are allocated based on certain conditions, for example:

```
!--- needed for Thompson's aerosol option
if(Model%imp_physics == Model%imp_physics_thompson .and. Model%ltaerosol) then
```

(continues on next page)

(continued from previous page)

```
  allocate (Coupling%nwfa2d (IM))
  allocate (Coupling%nifa2d (IM))
  Coupling%nwfa2d   = clear_val
  Coupling%nifa2d   = clear_val
endif
```

Other examples are the elements in the tracer array, where their presence depends on the corresponding index being larger than zero. For example:

```
integer              :: ntwa            !< tracer index for water friendly aerosol
...
Model%ntwa             = get_tracer_index(Model%tracer_names, 'liq_aero', ...)
...
if (Model%ntwa>0) then
  ! do something with qgrs(:,:,Model%ntwa)
end if
```

The `active` attribute is a conditional statement that, if true, will allow the corresponding variable to be allocated. It must be written as a Fortran expression that equates to `.true.` or `.false.`, using the CCPP standard names of variables. Active attributes for all variables are `.true.` by default.

If a developer adds a new variable that is only allocated under certain conditions, or changes the conditions under which an existing variable is allocated, a corresponding change must be made in the metadata for the host model variables (`GFS_typedefs.meta` for the UFS Atmosphere or the SCM). See variables `nwfa2d` and `qgrs` in *Listing 6.2* for an example.

6.3 CCPP Variables in the SCM and UFS Atmosphere Host Models

While the use of standard Fortran variables is preferred, in the current implementation of the CCPP in the UFS Atmosphere and in the SCM almost all data is contained in DDTs for organizational purposes. In the case of the SCM and UFS Atmosphere, DDTs are defined in both `GFS_typedefs.F90` and `CCPP_typedefs.F90`. The current implementation of the CCPP in both *host models* uses the following set of DDTs:

- `GFS_init_type` variables to allow proper initialization of GFS physics
- `GFS_statein_type` prognostic state data provided by dycore to physics
- `GFS_stateout_type` prognostic state after physical parameterizations
- `GFS_sfcprop_type` surface properties read in and/or updated by climatology, obs, physics
- `GFS_coupling_type` fields from/to coupling with other components, e.g., land/ice/ocean
- `GFS_control_type` control parameters input from a namelist and/or derived from others
- `GFS_grid_type` grid data needed for interpolations and length-scale calculations
- `GFS_tbd_type` data not yet assigned to a defined container
- `GFS_cldprop_type` cloud properties and tendencies needed by radiation from physics
- `GFS_radtend_type` radiation tendencies needed by physics
- `GFS_diag_type` fields targeted for diagnostic output to disk
- `GFS_data_type` combined type of all of the above except `GFS_control_type`

- `GFS_interstitial_type` fields used to communicate variables among schemes in the *slow physics group* required to replace interstitial code that resided in `GFS_{physics, radiation}_driver.F90` in IPD
- `GFDL_interstitial_type` fields used to communicate variables among schemes in the *fast physics* group

The DDT descriptions provide an idea of what physics variables go into which data type. `GFS_diag_type` can contain variables that accumulate over a certain amount of time and are then zeroed out. Variables that require persistence from one timestep to another should not be included in the `GFS_diag_type` nor the `GFS_interstitial_type` DDTs. Similarly, variables that need to be shared between groups cannot be included in the `GFS_interstitial_type` DDT. Although this memory management is somewhat arbitrary, new variables provided by the host model or derived in an interstitial scheme should be put in a DDT with other similar variables.

Each DDT contains a create method that allocates the data defined using the metadata. For example, the `GFS_stateout_type` contains:

```
type GFS_stateout_type

   !-- Out (physics only)
   real (kind=kind_phys), pointer :: gu0 (:,:)   => null()  !< updated zonal wind
   real (kind=kind_phys), pointer :: gv0 (:,:)   => null()  !< updated meridional wind
   real (kind=kind_phys), pointer :: gt0 (:,:)   => null()  !< updated temperature
   real (kind=kind_phys), pointer :: gq0 (:,:,:) => null()  !< updated tracers

   contains
     procedure :: create  => stateout_create  !<   allocate array data
 end type GFS_stateout_type
```

In this example, `gu0`, `gv0`, `gt0`, and `gq0` are defined in the host-side metadata section, and when the subroutine `stateout_create` is called, these arrays are allocated and initialized to zero. With the CCPP, it is possible to not only refer to components of DDTs, but also to slices of arrays with provided metadata as long as these are contiguous in memory. An example of an array slice from the `GFS_stateout_type` looks like:

```
########################################################################
[ccpp-table-properties]
   name = GFS_stateout_type
   type = ddt
   dependencies =

 [ccpp-arg-table]
   name = GFS_stateout_type
   type = ddt
 ...
 ...
 [gq0(:,:,index_of_snow_mixing_ratio_in_tracer_concentration_array)]
   standard_name = snow_mixing_ratio_of_new_state
   long_name = ratio of mass of snow water to mass of dry air plus vapor (without␣
↪condensates) updated by physics
   units = kg kg-1
   dimensions = (horizontal_loop_extent,vertical_layer_dimension)
   type = real
   kind = kind_phys
```

Array slices can be used by physics schemes that only require certain values from an array.

6.4 CCPP API

The CCPP Application Programming Interface (API) is comprised of a set of clearly defined methods used to communicate variables between the host model and the physics and to run the physics. The API is automatically generated by the CCPP prebuild script (see Chapter 8) and contains the subroutines `ccpp_physics_init`, `ccpp_physics_timestep_init`, `ccpp_physics_run`, `ccpp_physics_timestep_finalize`, and `ccpp_physics_finalize` (described below).

6.4.1 Data Structure to Transfer Variables between Dynamics and Physics

The `cdata` structure is used for holding six variables that must always be available to the physics schemes. These variables are listed in a metadata table in `ccpp-framework/src/ccpp_types.meta` (*Listing 6.5*).

- Error code for handling in CCPP (`errmsg`).
- Error message associated with the error code (`errflg`).
- Loop counter for *subcycling* loops (`loop_cnt`).
- Loop extent for subcycling loops (`loop_max`).
- Number of block for explicit data blocking in CCPP (`blk_no`).
- Number of thread for threading in CCPP (`thrd_no`).

```
[ccpp-table-properties]
  name = ccpp_types
  type = module
  dependencies =

[ccpp-arg-table]
  name = ccpp_types
  type = module
[ccpp_t]
  standard_name = ccpp_t
  long_name = definition of type ccpp_t
  units = DDT
  dimensions = ()
  type = ccpp_t

########################################################################
[ccpp-table-properties]
  name = ccpp_t
  type = ddt
  dependencies =

[ccpp-arg-table]
  name = ccpp_t
  type = ddt
[errflg]
  standard_name = ccpp_error_code
  long_name = error code for error handling in CCPP
  units = 1
  dimensions = ()
  type = integer
```

(continues on next page)

(continued from previous page)

```
[errmsg]
  standard_name = ccpp_error_message
  long_name = error message for error handling in CCPP
  units = none
  dimensions = ()
  type = character
  kind = len=512
[loop_cnt]
  standard_name = ccpp_loop_counter
  long_name = loop counter for subcycling loops in CCPP
  units = index
  dimensions = ()
  type = integer
[loop_max]
  standard_name = ccpp_loop_extent
  long_name = loop extent for subcycling loops in CCPP
  units = count
  dimensions = ()
  type = integer
[blk_no]
  standard_name = ccpp_block_number
  long_name = number of block for explicit data blocking in CCPP
  units = index
  dimensions = ()
  type = integer
[thrd_no]
  standard_name = ccpp_thread_number
  long_name = number of thread for threading in CCPP
  units = index
  dimensions = ()
  type = integer
```

Listing 6.5: Mandatory variables provided by the CCPP Framework from `ccpp-framework/src/ccpp_types.meta` *. These variables must not be defined by the host model.*

Two of the variables are mandatory and must be passed to every physics scheme: `errmsg` and `errflg`. The variables `loop_cnt`, `loop_max`, `blk_no`, and `thrd_no` can be passed to the schemes if required, but are not mandatory. They are, however, required for the auto-generated caps to pass the correct data to the physics and to realize the subcycling of schemes. The `cdata` structure is only used to hold these six variables, since the host model variables are directly passed to the physics without the need for an intermediate data structure.

Note that `cdata` is not restricted to being a scalar but can be a multidimensional array, depending on the needs of the host model. For example, a model that uses a one-dimensional array of blocks for better cache-reuse and OpenMP threading to process these blocks in parallel may require `cdata` to be a two-dimensional array of size "number of blocks" x "number of OpenMP threads".

6.4.2 Initializing and Finalizing the CCPP

At the beginning of each run, the `cdata` structure needs to be set up. Similarly, at the end of each run, it needs to be terminated. This is done with subroutines `ccpp_init` and `ccpp_finalize`. These subroutines should not be confused with `ccpp_physics_init` and `ccpp_physics_finalize`, which were described in Chapter 5.

Note that optional arguments are denoted with square brackets.

Suite Initialization

The *suite* initialization step consists of allocating (if required) and initializing the `cdata` structure(s), it does not call the CCPP Physics or any auto-generated code. The simplest example is a suite initialization step that consists of initializing a scalar `cdata` instance with `cdata%blk_no = 1` and `cdata%thrd_no = 1`.

A more complicated example is when multiple `cdata` structures are in use, namely one for the the CCPP phases that require access to all data of an MPI task (a scalar that is initialized in the same way as above), and one for the `run` phase, where chunks of blocked data are processed in parallel by multiple OpenMP threads, as shown in Listing *Listing 6.6*.

```
...

type(ccpp_t),                               target :: cdata_domain
type(ccpp_t), dimension(:,:), allocatable, target :: cdata_block

! ccpp_suite is set during the namelist read by the host model
character(len=256) :: ccpp_suite
integer            :: nthreads

...

! Get and set number of OpenMP threads (module
! variable) that are available to run physics
nthreads = omp_get_max_threads()

! For physics running over the entire domain,
! block and thread number are not used
cdata_domain%blk_no  = 1
cdata_domain%thrd_no = 1

! Allocate cdata structure for blocks and threads
allocate(cdata_block(1:nblks,1:nthreads))

! Assign the correct block and thread numbers
do nt=1,nthreads
  do nb=1,nblks
    cdata_block(nb,nt)%blk_no = nb
    cdata_block(nb,nt)%thrd_no = nt
  end do
end do
```

Listing 6.6: A morre complex suite initialization step that consists of allocating and initializing multiple ``cdata`` structures.

Depending on the implementation of CCPP in the host model, the suite name for the suite to be executed must be set in this step as well (omitted in Listing *Listing 6.6*).

Suite Finalization

The suite finalization consists of deallocating any `cdata` structures, if applicable, and optionally resetting scalar `cdata` instances as in the following example for the UFS:

```
deallocate(cdata_block)
! Optional
cdata_domain%blk_no = -999
cdata_domain%thrd_no = -999
...
```

6.4.3 Running the Physics

The physics is invoked by calling subroutine `ccpp_physics_run`. This subroutine is part of the CCPP API and is auto-generated. This subroutine is capable of executing the physics with varying granularity, that is, a single group, or an entire suite can be run with a single subroutine call. Typical calls to `ccpp_physics_run` are below,where `suite_name` is mandatory and `group_name` is optional:

```
call ccpp_physics_run(cdata, suite_name, [group_name], ierr=ierr)
```

6.4.4 Initializing and Finalizing the Physics

Many (but not all) physical *parameterizations* need to be initialized, which includes functions such as reading lookup tables, reading input datasets, computing derived quantities, broadcasting information to all MPI ranks, etc. Initialization procedures are done for the entire domain, that is, they are not subdivided by blocks and need access to all data that an MPI task owns. Similarly, many (but not all) parameterizations need to be finalized, which includes functions such as deallocating variables, resetting flags from *initialized* to *non-initialized*, etc. Initialization and finalization functions are each performed once per run, before the first call to the physics and after the last call to the physics, respectively. They may not contain thread-dependent or block-dependent information.

The initialization and finalization can be invoked for a single group, or for the entire suite. In both cases, subroutines `ccpp_physics_init` and `ccpp_physics_finalize` are used and the arguments passed to those subroutines determine the type of initialization.

Subroutine `ccpp_physics_init`

This subroutine is part of the CCPP API and is auto-generated. A typical call to `ccpp_physics_init` is:

```
call ccpp_physics_init(cdata, suite_name, [group_name], ierr=ierr)
```

Subroutine `ccpp_physics_finalize`

This subroutine is part of the CCPP API and is auto-generated. A typical call to `ccpp_physics_finalize` is:

```
call ccpp_physics_finalize(cdata, suite_name, [group_name], ierr=ierr)
```

6.4.5 Initializing and Finalizing the time step

The time step initialization typically consists of updating quantities that depend on the valid time, for example solar insulation angle, aerosol emission rates and other values obtained from climatologies. Like the physics initialization and finalization steps, the time step intialization and finalization steps need access to the entire data of an MPI task and may not contain thread-dependent or block-dependent information.

Subroutine `ccpp_physics_timestep_init`

This subroutine is part of the CCPP API and is auto-generated.A typical call to `ccpp_physics_timestep_init` is:

```
call ccpp_physics_timestep_init(cdata, suite_name, [group_name], ierr=ierr)
```

Subroutine `ccpp_physics_timestep_finalize`

This subroutine is part of the CCPP API and is auto-generated. A typical call to `ccpp_physics_timestep_finalize` is:

```
call ccpp_physics_timestep_finalize(cdata, suite_name, [group_name], ierr=ierr)
```

6.5 Host Caps

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 (as is done for the relatively simple SCM), it is recommended to separate the *cap* in its own module for clarity and simplicity (as is done for the UFS Atmosphere). While the details of implementation will be specific to each host model, the host model *cap* is responsible for the following general functions:

- Allocating memory for variables needed by physics
  - All variables needed to communicate between the host model and the physics, and all variables needed to communicate among physics schemes, need to be allocated by the host model. The latter, for example for interstitial variables used exclusively for communication between the physics schemes, are typically allocated in the *cap*.
- Allocating and initializing the `cdata` structure(s) and setting the suite name (suite initialization)
- Providing interfaces to call the CCPP
  - 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_physics_init`, `ccpp_physics_timestep_init`, `ccpp_physics_run`, `ccpp_physics_timestep_finalize` and `ccpp_physics_finalize`, and handle any errors returned. *Listing 6.7* provides an example where the host cap consists of three subroutines `physics_init` (which consists of the suite initialization and CCPP physics init phase), `physics_run` (which internally performs the CCPP time step init, run, and time step finalize phases), and `physics_finalize` (which consists of the suite finalization and CCPP physics finalize phase).

```
module example_ccpp_host_cap

 use ccpp_types,         only: ccpp_t
 use ccpp_static_api,    only: ccpp_physics_init,             &
                               ccpp_physics_timestep_init,    &
```

(continues on next page)

(continued from previous page)

```
                              ccpp_physics_run,               &
                              ccpp_physics_timestep_finalize, &
                              ccpp_physics_finalize

  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 cdata
   cdata%blk_no = 1
   cdata%thrd_no = 1

   ! Initialize CCPP physics (run all _init routines)
   call ccpp_physics_init(cdata, suite_name=trim(ccpp_suite_name),      &
                          ierr=ierr)

 end subroutine physics_init

 subroutine physics_run(ccpp_suite_name, group)
   ! Optional argument group can be used to run a group of schemes      &
   ! defined in the SDF. Otherwise, run entire suite.
   character(len=*),           intent(in) :: ccpp_suite_name
   character(len=*), optional, intent(in) :: group

   integer :: ierr
   ierr = 0

   if (present(group)) then
      call ccpp_physics_timestep_init(cdata,                           &
                           suite_name=trim(ccpp_suite_name),           &
                           group_name=group, ierr=ierr)
      call ccpp_physics_run(cdata, suite_name=trim(ccpp_suite_name),    &
                           group_name=group, ierr=ierr)
      call ccpp_physics_timestep_finalize(cdata,                       &
                           suite_name=trim(ccpp_suite_name),           &
                           group_name=group, ierr=ierr)
   else
      call ccpp_physics_timestep_init(cdata,                           &
                           suite_name=trim(ccpp_suite_name), ierr=ierr)
      call ccpp_physics_run(cdata, suite_name=trim(ccpp_suite_name),    &
                           ierr=ierr)
      call ccpp_physics_timestep_finalize(cdata,                       &
                           suite_name=trim(ccpp_suite_name), ierr=ierr)
   end if
```

(continues on next page)

(continued from previous page)

```
 end subroutine physics_run

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

   ! Finalize CCPP physics (run all _finalize routines)
   call ccpp_physics_finalize(cdata, suite_name=trim(ccpp_suite_name),  &
                              ierr=ierr)

   ! Reset cdata
   cdata%blk_no = -999
   cdata%thrd_no = -999

 end subroutine physics_finalize

end module example_ccpp_host_cap
```

Listing 6.7: Fortran template for a CCPP host model cap. After each call to ``ccpp_physics_*``, the host model should check the return code ``ierr`` and handle any errors (omitted for readability).

Readers are referred to the actual implementations of the cap functions in the CCPP-SCM and the UFS for further information. For the SCM, the cap functions are implemented in:

- `ccpp-scm/scm/src/scm.F90`
- `ccpp-scm/scm/src/scm_type_defs.F90`
- `ccpp-scm/scm/src/scm_setup.F90`
- `ccpp-scm/scm/src/scm_time_integration.F90`

For the UFS, the cap functions can be found in `ufs-weather-model/FV3/ccpp/driver/CCPP_driver.F90`.

CHAPTER

SEVEN

CCPP CODE MANAGEMENT

7.1 Organization of the Code

This chapter describes the organization of the code, provides instruction on the GitHub workflow and the code review process, and outlines the release procedure. It is assumed that the reader is familiar with using basic GitHub features. A GitHub account is necessary if a user would like to make and contribute code changes to the *CCPP*.

The repository and code organization differs for *CCPP Framework* and *CCPP Physics*.

7.1.1 CCPP Framework

The CCPP Framework code base can be found in the authoritative repository in the *NCAR* GitHub organization (https://github.com/NCAR/ccpp-framework). This repository is public and can be viewed, downloaded, or cloned by users without needing a GitHub account.

Developers seeking to contribute code to the CCPP should create a GitHub account and set up a personal fork in order to introduce changes to the official code base via a Pull Request (PR) on GitHub (see *Creating Forks*).

The following is the directory structure for the `ccpp-framework` repository:

```
├── doc                    # Documentation for design/implementation and developers guide
│   ├── DevelopersGuide
│   ├── HelloWorld         # Toy model to use of the CCPP Framework
│   └── img
├── logging                # Logging handler for future capgen.py
├── schema                 # XML scheme for suite definition files
├── scripts                # Scripts for ccpp_prebuild.py, metadata parser, etc.
│   ├── conversion_tools
│   ├── fortran_tools
│   └── parse_tools
├── src                    # CCPP framework source code
├── stub                   # CCPP stub build directory[1]
├── test                   # Unit/system testing framework for future capgen.py
│   ├── advection_test
│   ├── capgen_test
│   ├── hash_table_tests
│   └── unit_tests
└── tests                  # System testing framework for ccpp_prebuild.py
```

[1] see Section 8.5

7.1.2 CCPP Physics

Because the CCPP Physics repository accepts contributions coming from multiple host models and applications, the code and repository organization is a bit more complex. The main "authoritative" code base for CCPP Physics can be found in the NCAR GitHub organization (https://github.com/NCAR/ccpp-physics). This repository is public and can be viewed, downloaded, or cloned by users without needing a GitHub account. However, in most cases code changes are not applied to this repository directly: Each *host model* or application (aside from the *SCM*) maintains its own Application Fork that accepts changes to CCPP Physics specifically in the context of that application. Code managers regularly sync changes from the Application Forks to the authoritative CCPP Physics repository in order to ensure a unified CCPP Physics code base. For more information about Application Forks, see the GitHub Wiki Page.

Developers should create a personal fork from the appropriate Application Fork in order to introduce changes to the official code base via a Pull Request (PR) on GitHub (see *Creating Forks*). Currently, the only Application Fork is for the UFS, so users should fork from there.

The following is the directory structure for the `ccpp-physics` repository (condensed version):

```
├── physics                 # CCPP physics source code and metadata files
│   ├── docs                # Scientific documentation (doxygen)
│   │   ├── img             # Figures for doxygen
│   │   └── pdftxt          # Text files for documentation
└── tools                   # Tools used by CI system for basic checks (encoding ...)
```

7.2 GitHub Workflow (setting up development repositories)

The CCPP development practices make use of the GitHub forking workflow. For users not familiar with this concept, this website provides some background information and a tutorial.

7.2.1 Creating Forks

The GitHub forking workflow relies on forks (personal copies) of the shared repositories on GitHub. A personal fork needs to be created only once, and only for repositories that users will contribute changes to. The following steps describe how to create a fork for CCPP development.

1. Go to the repository you wish to fork, and make sure you are signed in to your GitHub account.
   - For CCPP Framework changes, this should be the authoritative repository (https://github.com/NCAR/ccpp-framework)
   - **For CCPP Physics changes, this should be the Application Fork corresponding to your host model of interest**
     - UFS Fork (https://github.com/ufs-community/ccpp-physics)
2. Select the "fork" button in the upper right corner.
   - If you have already created a fork, this will take you to your fork.
   - If you have not yet created a fork, this will create one for you.

Note: If you already have a fork for a different CCPP Physics repository and so can not create a new one, contact the code managers via GitHub discussions (https://github.com/NCAR/ccpp-physics/discussions)

7.2.2 Checking out the Code

Instructions are provided here for the ccpp-physics repository assuming development intended for use in UFS Applications. The instructions for the ccpp-framework repository are analogous but should start from the main repository in the NCAR GitHub Organization (https://github.com/NCAR/ccpp-framework).

The process for checking out the CCPP is described in the following, assuming access via https (using a personal access token) rather than ssh. If you are using an ssh key instead, you should replace instances of `https://github.com/` with `git@github.com:` in repository URLs.

Start by checking out the UFS Application Fork:

```
git clone https://github.com/ufs-community/ccpp-physics
cd ccpp-physics
git remote rename origin upstream
```

In the above commands we have also renamed the "origin" repository to "upstream" within this clone. This will be required if you plan on making changes and contributing them back to your fork, but is otherwise unnecessary. This step prevents accidentally pushing changes to the main repository rather than your fork later on.

From here you can view the available branches in the ccpp-physics repository with the `git branch` command:

```
git fetch --all
git branch -a

* ufs/dev
  remotes/upstream/HEAD -> upstream/ufs/dev
  remotes/upstream/ufs/dev
```

As you can see, you are placed on the `ufs/dev` branch by default; this is the most recent version of the development code in the ccpp-physics repository. In the ccpp-framework repository, the default branch is named `main`. All new development should start from the default branch, but if you would like to view code from another branch this is simple with the `git checkout` command.

```
git checkout release/public-v6

branch 'release/public-v6' set up to track 'upstream/release/public-v6'.
Switched to a new branch 'release/public-v6'
```

Note: Never used git or GitHub before? Confused by what all this means or why we do it? Check out this presentation from the UFS SRW Training workshop for a "from basic principles" explanation!

After this command, git has checked out a local copy of the remote branch `upstream/release/public-v6` named `release/public-v6`. To return to the ufs/dev branch, simply use `git checkout ufs/dev`.

If you wish to make changes that you will eventually contribute back to the public code base, you should always create a new "feature" branch that will track those particular changes.

```
git checkout upstream/ufs/dev
git checkout -b feature/my_new_local_development_branch
```

Note: By checking out the remote `upstream/ufs/dev` branch directly, you will be left in a so-called 'detached HEAD' state. This will prompt git to show you a scary-looking warning message, but it can be ignored so long as you

follow it by the second command above to create a new branch.

You can now make changes to the code, and commit those changes locally using `git commit` in order to track

Once you are ready to contribute the code back to the main (`upstream`) ccpp-physics repository, you need to create a pull request (PR) (see *Creating a pull request*). In order to do so, you first need to create your own fork of this repository (see *Creating Forks*) and configure your fork as an additional remote destination, which we typically label as *origin*. For example:

```
git remote add origin https://github.com/YOUR_GITHUB_USER/ccpp-physics
git fetch origin
```

Then, push your local branch to your fork:

```
git push origin my_local_development_branch
```

For each repository/submodule, you can check the configured remote destinations and all existing branches (remote and local):

```
git remote -v show
git remote update
git branch -a
```

As opposed to branches without modifications described in step 3, changes to the upstream repository can be brought into the local branch by pulling them down. For example (where a local branch is checked out):

```
cd ccpp-physics
git remote update
git pull upstream ufs/dev
```

7.3 Committing Changes to your Fork

Once you have your fork set up to begin code modifications, you should check that the cloned repositories upstream and origin are set correctly:

```
git remote -v
```

This should point to your fork as *origin* and the repository you cloned as *upstream*:

```
origin            https://github.com/YOUR_GITHUB_USER/ccpp-physics (fetch)
origin            https://github.com/YOUR_GITHUB_USER/ccpp-physics (push)
upstream   https://github.com/ufs-community/ccpp-physics (fetch)
upstream   https://github.com/ufs-community/ccpp-physics (push)
```

Also check what branch you are working on:

```
git branch
```

This command will show what branch you have checked out on your fork:

```
* features/my_local_development_branch
  ufs/dev
```

After making modifications and testing, you can commit the changes to your fork. First check what files have been modified:

```
git status
```

This git command will provide some guidance on what files need to be added and what files are "untracked". To add new files or stage modified files to be committed:

```
git add filename1 filename2
```

At this point it is helpful to have a description of your changes to these files documented somewhere, since when you commit the changes, you will be prompted for this information. To commit these changes to your local repository and push them to the development branch on your fork:

```
git commit
git push origin features/my_local_development_branch
```

When this is done, you can check the status again:

```
git status
```

This should show that your working copy is up to date with what is in the repository:

```
On branch features/my_local_development_branch
Your branch is up to date with 'origin/features/my_local_development_branch'.
nothing to commit, working tree clean
```

At this point you can continue development or create a PR as discussed in *Creating a Pull Request*.

7.4 Contributing Code, Code Review Process

Once your development is mature, and the testing has been completed, you are ready to create a PR using GitHub to propose your changes for review.

7.4.1 Creating a Pull Request

Go to the github.com web interface, and navigate to your repository fork and branch. In most cases, this will be in the ccpp-physics repository, hence the following example:

- Navigate to: https://github.com/<yourusername>/ccpp-physics
- Use the drop-down menu on the left-side to select a branch to view your development branch
- Use the button just right of the branch menu, to start a "New Pull Request"
- Fill in a short title (one line)
- Fill in a detailed description, including reporting on any testing you did
- Click on "Create pull request"

If your development also requires changes in other repositories, you must open PRs in those repositories as well. In the PR message for each repository, please note the associated PRs submitted to other repositories.

Several people (aka CODEOWNERS) are automatically added to the list of reviewers on the right hand side. Once the PR has been approved, the change is merged to ufs/dev by one of the code owners. If there are pending conflicts, this

means that the code is not up to date with the trunk. To resolve those, pull the target branch from upstream as described above, solve the conflicts and push the changes to the branch on your fork (this also updates the PR).

Note: GitHub offers a "Draft pull request" feature that allows users to push their code to GitHub and create a draft PR. Draft PRs cannot be merged and do not automatically initiate notifications to the CODEOWNERS, but allow users to prepare the PR and flag it as "ready for review" once they feel comfortable with it. To open a draft rather than a ready-for-review PR, select the arrow next to the green "Create pull request" button, and select "Create draft pull request". Then continue the above steps as usual.

CHAPTER

EIGHT

TECHNICAL ASPECTS OF THE CCPP *PREBUILD*

8.1 *Prebuild* Script Function

The *CCPP prebuild* script `ccpp-framework/scripts/ccpp_prebuild.py` is the central piece of code in the *CCPP Framework* that connects the *host model* with the *CCPP Physics* schemes (see 3.1). This script must be run before compiling the CCPP Physics library and the host model cap. This may be done manually or as part of a host model build-time script. Both the *UFS* and *SCM* have incorporated the calls to `ccpp_prebuild.py` in their build systems.

The CCPP *prebuild* script automates several tasks based on the information collected from the metadata on the host model side and from the individual physics schemes (`.meta` files; see Figure 8.1):

- Compiles a list of variables provided by the host model.
- Compiles a list of variables required to run all schemes in the CCPP Physics pool.
- Matches these variables by their `standard_name`, checks for missing variables and mismatches of their attributes (e.g., units, rank, type, kind). Performs automatic unit conversions if a mismatch of units is detected between a scheme and the host model (see Section 5.2 for details).
- Filters out unused variables for a given *suite*.
- Autogenerates software caps as appropriate:
  - The script generates *caps* for the suite as a whole and physics *groups* as defined in the input *SDF*s; in addition, the CCPP API for the build is generated.
- Populates makefiles with kind/type definitions, schemes, caps. Statements to compile the CCPP API are included as well.

8.2 Script Configuration

To connect the CCPP with a host model `XYZ`, a Python-based configuration file for this model must be created in the host model's repository. The easiest way is to copy an existing configuration file for the SCM in sub-directory `ccpp/config` of the ccpp-scm repository. The configuration in `ccpp_prebuild_config.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 8.1 contains an example for the CCPP-SCM prebuild config. Here, both `ccpp-framework` and `ccpp-physics` are located in directories `ccpp/framework` and `ccpp/physics` of the top-level directory of the host model, and `ccpp_prebuild.py` is executed from the same top-level directory.

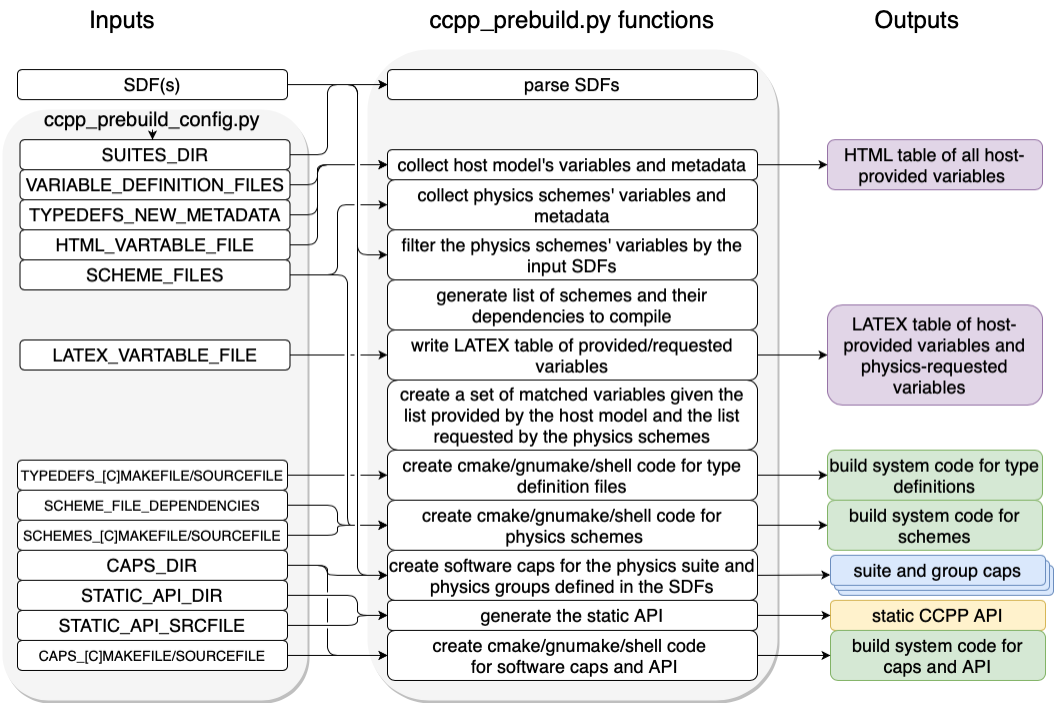


Fig. 8.1: *Schematic of tasks automated by the* `ccpp_prebuild.py` *script and associated inputs and outputs. The majority of the input is controlled through the host-model dependent* `ccpp_prebuild_config.py` *, whose user-editable variables are included as all-caps within the* `ccpp_prebuild_config.py` *bubble. Outputs are color-coded according to their utility: purple outputs are informational only (useful for developers, but not necessary to run the code), yellow outputs are used within the host model, blue outputs are connected to the physics, and green outputs are used in the model build.*

```
# Host model identifier
HOST_MODEL_IDENTIFIER = "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/scm_type_defs.F90',
    'scm/src/scm_physical_constants.F90',
    ]

 # How parent variables (module variables, derived data types)
 # are referenced in the model
 TYPEDEFS_NEW_METADATA = {
     'ccpp_types' : {
         'ccpp_types' : '',
         'ccpp_t' : 'cdata',
         },
     'GFS_typedefs' : {
         'GFS_typedefs' : '',
         'GFS_control_type' : 'physics%Model',
         },
     }

# Add all physics scheme files relative to basedir
SCHEME_FILES = {
    'ccpp/physics/physics/GFS_DCNV_generic.f90' ,
    'ccpp/physics/physics/sfc_sice.f',
    }

 # Default build dir, relative to current working directory,
 # if not specified as command-line argument
 DEFAULT_BUILD_DIR = 'scm/bin'

 # Auto-generated makefile/cmakefile snippets that contain all type definitions
 TYPEDEFS_MAKEFILE   = '{build_dir}/ccpp/physics/CCPP_TYPEDEFS.mk'
 TYPEDEFS_CMAKEFILE  = '{build_dir}/ccpp/physics/CCPP_TYPEDEFS.cmake'
 TYPEDEFS_SOURCEFILE = '{build_dir}/ccpp/physics/CCPP_TYPEDEFS.sh'

 # Auto-generated makefile/cmakefile snippets that contain all schemes
 SCHEMES_MAKEFILE = '{build_dir}/ccpp/physics/CCPP_SCHEMES.mk'
 SCHEMES_CMAKEFILE = '{build_dir}/ccpp/physics/CCPP_SCHEMES.cmake'
 SCHEMES_SOURCEFILE = '{build_dir}/ccpp/physics/CCPP_SCHEMES.sh'

 # Auto-generated makefile/cmakefile snippets that contain all caps
 CAPS_MAKEFILE = '{build_dir}/ccpp/physics/CCPP_CAPS.mk'
 CAPS_CMAKEFILE = '{build_dir}/ccpp/physics/CCPP_CAPS.cmake'
 CAPS_SOURCEFILE = '{build_dir}/ccpp/physics/CCPP_CAPS.sh'

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

 # Directory where the suite definition files are stored
 SUITES_DIR = 'ccpp/suites'
```

(continues on next page)

(continued from previous page)

```
# Directory where to write static API to
STATIC_API_DIR = 'scm/src/'
STATIC_API_SRCFILE = 'scm/src/CCPP_STATIC_API.sh'

# Directory for writing HTML pages generated from metadata files
METADATA_HTML_OUTPUT_DIR = 'ccpp/physics/physics/docs'

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

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

Listing 8.1: CCPP prebuild config for SCM (shortened)

Although most of the variables in the `ccpp_prebuild_config.py` script are described by in-line comments in the code listing above and their use is described in Figure 8.1, some clarifying comments are in order. The `SCHEME_FILES` variable is a list of CCPP-compliant physics scheme entry/exit point source files only, their dependencies are not listed here (see Section 2.2 for how dependencies are included). `TYPEDEFS_NEW_METADATA` is a dictionary that, for each Fortran module name contained in the files in `VARIABLE_DEFINITION_FILES` (the key of the dictionary), contains a nested dictionary (the value) that describes how the module itself and the derived data types are referenced in the host model. For the module itself, an empty string is typically the correct choice. For each of the derived data types contained in the module, a Fortran variable expression is required, as shown in the listing above. This entry is needed to correctly identify and pass parent variables (derived data types) of variables that are needed by the physics to the auto-generated caps.

8.3 Running ccpp_prebuild.py

Once the configuration in `ccpp_prebuild_config.py` is complete, the `ccpp_prebuild.py` script can be run from a specific directory, dependent on the host model. For the SCM, this is the top level directory, i.e. the correct call to the script is `./ccpp/framework/scripts/ccpp_prebuild.py`. For the *UFS Atmosphere* host model, the script needs to be called from subdirectory `FV3/ccpp`, relative to the top-level `ufs-weather-model` directory. In the following, we use the SCM directory structure. Note that for both SCM and UFS, the `ccpp_prebuild.py` script is called automatically by the build system.

For developers adding a CCPP-compliant physics scheme, running `ccpp_prebuild.py` periodically is recommended to check that the metadata provided with the physics schemes matches what the host model provided. As alluded to above, the `ccpp_prebuild.py` script has six command line options, with the path to a host-model specific configuration file (`--config`) being the only required option:

`-h, --help` show this help message and exit

`--config` `PATH_TO_CONFIG/config_file` path to CCPP *prebuild* configuration file

`--clean` remove files created by this script, then exit

`--verbose` enable verbose output

`--debug` enable additional checks on array sizes

`--suites` SUITES SDF(s) to use (comma-separated, without path)

An example invocation of running the script (called from the SCM's top level directory) would be:

```
./ccpp/framework/scripts/ccpp_prebuild.py \
  --config=./ccpp/config/ccpp_prebuild_config.py \
```

(continues on next page)

(continued from previous page)

```
  --suites=FV3_GFS_v16 \
  --verbose
```

which uses a configuration script located at the specified path. The `--verbose` option can be used for more verbose output from the script.

The *SDF*(s) to compile into the executable can be specified using the `--suites` command-line argument. Such files are included with the SCM and ufs-weather-model repositories, and must be included with the code of any host model to use the CCPP. An example of a build using two SDFs is:

```
./ccpp/framework/scripts/ccpp_prebuild.py \
  --config=./ccpp/config/ccpp_prebuild_config.py \
  --suites=FV3_GFS_v16,RRFS_v1beta
```

Note: If the `--suites` option is omitted, all suites will be compiled into the executable.

The `--debug` command-line argument enables additional checks on array sizes inside the auto-generated software caps, prior to entering any of the schemes.

If the CCPP *prebuild* step is successful, the last output line will be:

```
INFO: CCPP prebuild step completed successfully.
```

To remove all files created by `ccpp_prebuild.py`, for example as part of a host model's `make clean` functionality, execute the same command as before, but with `--clean` appended:

```
./ccpp/framework/scripts/ccpp_prebuild.py --config=./ccpp/config/ccpp_prebuild_config.py␣
↪\
--suites=FV3_GFS_v16,RRFS_v1beta --clean
```

8.4 Troubleshooting

If invoking the `ccpp_prebuild.py` script fails, some message other than the success message will be written to the terminal output. Specifically, the terminal output will include informational logging messages generated from the script and any error messages written to the Python logging utility. Some common errors (minus the typical logging output and traceback output) and solutions are described below, with non-bold font used to denote aspects of the message that will differ depending on the problem encountered. This is not an exhaustive list of possible errors, however. For example, in this version of the code, there is no cross-checking that the metadata information provided corresponds to the actual Fortran code, so even though `ccpp_prebuild.py` may complete successfully, there may be related compilation errors later in the build process. For further help with an undescribed error, you can make a post in the appropriate GitHub discussions forum for *CCPP Physics* (https://github.com/NCAR/ccpp-physics/discussions) or *CCPP Framework* (https://github.com/NCAR/ccpp-framework/discussions).

1. **`ERROR: Configuration file erroneous/path/to/config/file not found`**

   - Check that the path entered for the `--config` command line option points to a readable configuration file.

2. **`KeyError: 'erroneous_scheme_name'` when using the `--suites` option**

   - This error indicates that a scheme within the supplied *SDF*s does not match any scheme names found in the SCHEME_FILES variable of the supplied configuration file that lists scheme source files. Double check that the scheme's source file is included in the SCHEME_FILES list and that the scheme name

that causes the error is spelled correctly in the supplied SDFs and matches what is in the source file (minus any `*_timestep_init`, `*_init`, `*_run`, `*_finalize`, `*_timestep_finalize` suffixes).

3. `CRITICAL: Suite definition file` erroneous/path/to/SDF.xml `not found`.

   `Exception: Parsing suite definition file` erroneous/path/to/SDF.xml `failed`.

   - Check that the path `SUITES_DIR` in the CCPP prebuild config and the names entered for the `--suites` command line option are correct.

4. `INFO: Parsing metadata tables for variables provided by host model ...`

   `IOError: [Errno 2] No such file or directory:` 'erroneous_file.f90'

   - Check that the paths specified in the `VARIABLE_DEFINITION_FILES` of the supplied configuration file are valid and contain CCPP-compliant host model snippets for insertion of metadata information. (see *example*)

5. **`Exception: Error parsing variable entry` "erroneous variable metadata table entry data" `in argument table` variable_metadata_table_name**

   - Check that the formatting of the metadata entry described in the error message is OK.

6. **`Exception: New entry for variable` var_name `in argument table` variable_metadata_table_name `is incompatible with existing entry:`**

   `Existing: Contents of <mkcap.Var object at 0x10299a290> (* = mandatory for compatibility):`
   `standard_name` = var_name *
   `long_name` =
   `units` = various *
   `local_name` =
   `type` = real *
   `rank` = (:,:,:) *
   `kind` = kind_phys *
   `intent` = none
   `active` = T
   `target` = None
   `container` = MODULE_X TYPE_Y
   `vs. new: Contents of <mkcap.Var object at 0x10299a310> (* = mandatory for compatibility):`
   `standard_name` = var_name *
   `long_name` =
   `units` = frac *
   `local_name` =
   `type` = real *
   `rank` = (:,:) *
   `kind` = kind_phys *
   `intent` = none
   `active` = T
   `target` = None
   `container` = MODULE_X TYPE_Y

   - This error is associated with a variable that is defined more than once (with the same *standard name*) on the host model side. Information on the offending variables is provided so that one can provide different standard names to the different variables.

7. **Exception: Scheme name differs from module name: module_name= "X" vs. scheme_name= "Y"**

    - Make sure that each scheme in the errored module begins with the module name and ends in either `*_timestep_init`, `*_init`, `*_run`, `*_finalize`, or `*_timestep_finalize`.

8. **Exception: New entry for variable var_name in argument table of subroutine scheme_subroutine_name is incompatible with existing entry:**

    ```
    existing: Contents of <mkcap.Var object at 0x10299a290> (* = mandatory for
    compatibility):
        standard_name = var_name *
        long_name =
        units = various *
        local_name =
        type = real *
        rank = (:,:,:) *
        kind = kind_phys *
        intent = none
        active = T
        target = None
        container = MODULE_X TYPE_Y
    vs. new: Contents of <mkcap.Var object at 0x10299a310> (* = mandatory for
    compatibility):
        standard_name = var_name *
        long_name =
        units = frac *
        local_name =
        type = real *
        rank = (:,:) *
        kind = kind_phys *
        intent = none
        active = T
        target = None
        container = MODULE_X TYPE_Y
    ```

    - This error is associated with physics scheme variable metadata entries that have the same standard name with different mandatory properties (either units, type, rank, or kind currently – those attributes denoted with a *). This error is distinguished from the error described in 8 above, because the error message mentions "in argument table of subroutine" instead of just "in argument table".

9. **ERROR: Variable X requested by MODULE_Y SCHEME_Z SUBROUTINE_A not provided by the model**

    ```
    Exception: Call to compare_metadata failed.
    ```

    - A variable requested by one or more physics schemes is not being provided by the host model. If the variable exists in the host model but is not being made available for the CCPP, an entry must be added to one of the host model variable metadata sections.

10. **ERROR: error, variable X requested by MODULE_Y SCHEME_Z SUBROUTINE_A cannot be identified unambiguously. Multiple definitions in MODULE_Y TYPE_B**

    - A variable is defined in the host model variable metadata more than once (with the same standard name). Remove the offending entry or provide a different standard name for one of the duplicates.

11. **ERROR: incompatible entries in metadata for variable var_name:**

```
provided:  Contents of <mkcap.Var object at 0x104883210> (* = mandatory for
compatibility):
     standard_name = var_name *
     long_name =
     units = K *
     local_name =
     type = real *
     rank = *
     kind = kind_phys *
     intent = none
     active = T
     target = None
     container =
requested:  Contents of <mkcap.Var object at 0x10488ca90> (* = mandatory for
compatibility):
     standard_name = var_name *
     long_name =
     units = none *
     local_name =
     type = real *
     rank = *
     kind = kind_phys *
     intent = in
     active = T
     target = None
     container =
```

12. **Exception: Call to compare_metadata failed.**

    - This error indicates a mismatch between the attributes of a variable provided by the host model and what is requested by the physics. Specifically, the units, type, rank, or kind don't match for a given variable standard name. Double-check that the attributes for the provided and requested mismatched variable are accurate. If after checking the attributes are indeed mismatched, reconcile as appropriate (by adopting the correct variable attributes either on the host or physics side).

Note: One error that the `ccpp_prebuild.py` script will not catch is if a physics scheme lists a variable in its actual (Fortran) argument list without a corresponding entry in the subroutine's variable metadata. This will lead to a compilation error when the autogenerated scheme cap is compiled:

```
Error:  Missing actual argument for argument 'X' at (1)
```

8.5 CCPP Stub Build

New in version 6.0, CCPP includes a *stub* capability, which will build the appropriate basic software caps needed for the compilation of the *host model*, but not include any of the physics itself. This can be useful for host model debugging, testing "dry" dynamics with no parameterizations, and other use cases where building the whole CCPP physics library would be unnecessary. Currently this capability is only supported for the *UFS Atmosphere*.

To create the stub software caps, rather than using the host configuration file as described above, users can use the provided stub config file `ccpp/framework/stub/ccpp_prebuild_config.py`. From the `ccpp/framework/stub` directory, the prebuild script can be called in this manner to use the CCPP stub build:

```
../scripts/ccpp_prebuild.py --config=ccpp_prebuild_config.py
cmake . 2>&1 | tee log.cmake
make 2>&1 | tee log.make
```

The rest of the UFS Atmosphere build can proceed as normal.

8.6 CCPP Physics Variable Tracker

New in version 6.0, CCPP includes a tool that allows users to track a given variable's journey through a specified physics suite. This tool, `ccpp-framework/scripts/ccpp_track_variables.py`, given a *suite definition file* and the *standard name* of a variable, will output the list of subroutines that use this variable – in the order that they are called – as well as the variable's Fortran *intent* (`in`, `out`, or `inout`) within that subroutine. This can allow the user to more easily determine where specific errors, biases, or other influences on a specific variable or variables might originate from within the physics suite. The `--help` option will give a basic rundown of how to use the script:

```
./ccpp_track_variables.py --help
usage: ccpp_track_variables.py [-h] -s SDF -m METADATA_PATH -c CONFIG -v VARIABLE [--
↪debug]

optional arguments:
  -h, --help            show this help message and exit
  -s SDF, --sdf SDF     suite definition file to parse
  -m METADATA_PATH, --metadata_path METADATA_PATH
                        path to CCPP scheme metadata files
  -c CONFIG, --config CONFIG
                        path to CCPP prebuild configuration file
  -v VARIABLE, --variable VARIABLE
                        variable to track through CCPP suite
  --debug               enable debugging output
```

For this initial implementation, this script must be executed from within a *host model*, and must be called from the same directory that the `ccpp_prebuild.py` script is called from. This first example is called using the *UFS Atmosphere* as a host model, from the directory `ufs-weather-model/FV3/ccpp`:

```
framework/scripts/ccpp_track_variables.py -c=config/ccpp_prebuild_config.py \
  -s=suites/suite_FV3_RRFS_v1beta.xml -v air_temperature_of_new_state -m ./physics/
↪physics/
For suite suites/suite_FV3_RRFS_v1beta.xml, the following schemes (in order for each␣
↪group) use the variable air_temperature_of_new_state:
In group physics
  GFS_suite_stateout_reset_run (intent out)
```

(continues on next page)

(continued from previous page)

```
  dcyc2t3_run (intent in)
  GFS_suite_stateout_update_run (intent out)
  ozphys_2015_run (intent in)
  get_phi_fv3_run (intent in)
  GFS_suite_interstitial_3_run (intent in)
  GFS_MP_generic_pre_run (intent in)
  mp_thompson_pre_run (intent in)
  mp_thompson_run (intent inout)
  mp_thompson_post_run (intent inout)
  GFS_MP_generic_post_run (intent in)
  maximum_hourly_diagnostics_run (intent in)
In group stochastics
  GFS_stochastics_run (intent inout)
```

In the example above, we can see that the variable `air_temperature_of_new_state` is used in the FV3_RRFS_v1beta suite by several microphysics-related schemes, as well as by the stochastics *parameterization*.

To learn more about a given subroutine, you can search the physics source code within the `ccpp-physics` repository, or you can consult the CCPP Scientific Documentation: typing the subroutine name into the search bar should lead you to further information about the subroutine and how it ties into its associated physics scheme. In addition, because of the naming conventions for subroutines in CCPP-compliant physics schemes, we can typically see which scheme, as well as which *phase* within that scheme, is associated with the listed subroutine, without having to consult any further documentation or source code. For example, the `mp_thompson_run` subroutine is part of the Thompson microphysics scheme, specifically the *run* phase of that scheme.

This second example is called using the *SCM* as a host model:

```
ccpp/framework/scripts/ccpp_track_variables.py --config=ccpp/config/ccpp_prebuild_config.
↪py \
   -s=ccpp/suites/suite_SCM_GFS_v17_p8.xml -v surface_friction_velocity_over_land -m ./
↪ccpp/physics/physics/
For suite ccpp/suites/suite_SCM_GFS_v17_p8.xml, the following schemes (in order for each␣
↪group) use the variable surface_friction_velocity_over_land:
In group physics
  GFS_surface_composites_pre_run (intent inout)
  sfc_diff_run (intent inout)
  noahmpdrv_run (intent inout)
  sfc_diff_run (intent inout)
  noahmpdrv_run (intent inout)
  GFS_surface_composites_post_run (intent in)
```

In the example above, we can see that the variable `wind_speed_at_lowest_model_layer` is used in a few subroutines, two of which (`sfc_diff_run` and `noahmpdrv_run` are listed twice). This is not an error! The two repeated subroutines are part of a scheme called in a *subcycle*, and so they are called twice in this cycle as designated in the SDF. The `ccpp_track_variables.py` script lists the subroutines in the exact order they are called (within each *group*), including subcycles.

Some standard names can be exceedingly long and hard to remember, and it is not always convenient to search the full list of standard names for the exact variable you want. Therefore, this script will also return matches for partial variable names. In this example, we will look for the variable "velocity", which is not a standard name of any variable, and see what it returns:

```
framework/scripts/ccpp_track_variables.py --config=config/ccpp_prebuild_config.py \
   -s=suites/suite_FV3_GFS_v16.xml -v velocity -m ./physics/physics/
```

(continues on next page)

(continued from previous page)

```
Variable velocity not found in any suites for sdf suites/suite_FV3_GFS_v16.xml

ERROR:ccpp_track_variables:Variable velocity not found in any suites for sdf suites/
↪suite_FV3_GFS_v16.xml

Did find partial matches that may be of interest:

In GFS_surface_composites_pre_run found variable(s) ['surface_friction_velocity',
↪'surface_friction_velocity_over_water', 'surface_friction_velocity_over_land',
↪'surface_friction_velocity_over_ice']
In sfc_diff_run found variable(s) ['surface_friction_velocity_over_water', 'surface_
↪friction_velocity_over_land', 'surface_friction_velocity_over_ice']
In GFS_surface_composites_post_run found variable(s) ['surface_friction_velocity',
↪'surface_friction_velocity_over_water', 'surface_friction_velocity_over_land',
↪'surface_friction_velocity_over_ice']
In cires_ugwp_run found variable(s) ['angular_velocity_of_earth']
In samfdeepcnv_run found variable(s) ['vertical_velocity_for_updraft', 'cellular_
↪automata_vertical_velocity_perturbation_threshold_for_deep_convection']
```

While the script did not find the variable specified, it did find several partial matches – `surface_friction_velocity`, `surface_friction_velocity_over_water`, `surface_friction_velocity_over_land`, etc. – as well as the subroutines they were found in. You can then use this more specific information to refine your next query:

```
framework/scripts/ccpp_track_variables.py --config=config/ccpp_prebuild_config.py \
   -s=suites/suite_FV3_GFS_v16.xml -v surface_friction_velocity -m ./physics/physics/
For suite suites/suite_FV3_GFS_v16.xml, the following schemes (in order for each group)␣
↪use the variable surface_friction_velocity:
In group physics
  GFS_surface_composites_pre_run (intent in)
  GFS_surface_composites_post_run (intent inout)
```

CHAPTER

NINE

TIPS FOR ADDING A NEW SCHEME

This chapter contains a brief description on how to add a new *scheme* to the *CCPP Physics* pool.

- Identify the variables required for the new scheme and check if they are already available for use in the CCPP by checking the metadata information in `GFS_typedefs.meta` or by perusing file `ccpp-framework/doc/DevelopersGuide/CCPP_VARIABLES_{FV3,SCM}.pdf` generated by `ccpp_prebuild.py`.
  - If the variables are already available, they can be invoked in the scheme's metadata file and one can skip the rest of this subsection. If the variable required is not available, consider if it can be calculated from the existing variables in the CCPP. If so, an *interstitial scheme* (such as `scheme_pre`; see more in Chapter 2) can be created to calculate the variable. However, the variable must be defined but not initialized in the *host model* as the memory for this variable must be allocated on the host model side. Instructions for how to add variables to the host model side is described in Chapter 6.

    Note: The *CCPP framework* is capable of performing automatic unit conversions between variables provided by the host model and variables required by the new scheme. See Section 5.2 for details.

  - If new namelist variables need to be added, the `GFS_control_type` DDT should be used. In this case, it is also important to modify the namelist file `input.nml` to include the new variable.
  - It is important to note that not all data types are persistent in memory. Most variables in the interstitial data type are reset (to zero or other initial values) at the beginning of a physics *group* and do not persist from one *set* to another or from one group to another. The diagnostic data type is periodically reset because it is used to accumulate variables for given time intervals. However, there is a small subset of interstitial variables that are set at creation time and are not reset; these are typically dimensions used in other interstitial variables.

    Note: If the value of a variable must be remembered from one call to the next, it should not be in the interstitial or diagnostic data types.

  - If information from the previous timestep is needed, it is important to identify if the host model readily provides this information. For example, in the Model for Prediction Across Scales (MPAS), variables containing the values of several quantities in the preceding timesteps are available. When that is not the case, as in the *UFS Atmosphere*, interstitial schemes are needed to compute these variables. As an example, the reader is referred to the GF convective scheme, which makes use of interstitials to obtain the previous timestep information.
  - Consider allocating the new variable only when needed (i.e. when the new scheme is used and/or when a certain control flag is set). If this is a viable option, following the existing ex-

amples in `GFS_typedefs.F90` and `GFS_typedefs.meta` for allocating the variable and setting the `active` attribute in the metadata correctly.

- If an entirely new variable needs to be added, consult the CCPP *standard names* dictionary and the rules for creating new standard names at https://github.com/escomp/CCPPStandardNames. If in doubt, use the GitHub discussions page in the CCPP Framework repository (https://github.com/ncar/ccpp-framework) to discuss the suggested new standard name(s) with the CCPP developers.
- Examine scheme-specific and *suite* interstitials to see what needs to be replaced/changed; then check existing scheme interstitial and determine what needs to replicated. Identify if your new scheme requires additional interstitial code that must be run before or 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 *SDF*.
- Follow the guidelines outlined in Chapter 2 to make your scheme CCPP-compliant. Make sure to use an uppercase suffix `.F90` to enable C preprocessing.
- Locate the CCPP *prebuild* configuration files for the target host model, for example:
  - `ufs-weather-model/FV3/ccpp/config/ccpp_prebuild_config.py` for the *UFS Atmosphere*
  - `ccpp-scm/ccpp/config/ccpp_prebuild_config.py` for the *SCM*
- Add the new scheme to the Python dictionary in `ccpp_prebuild_config.py` using the same path as the existing schemes:

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

- Place new scheme in the same location as existing schemes in the CCPP directory structure, e.g., `../some_relative_path/new_scheme.F90`.
- Edit the SDF and add the new scheme at the place it should be run. SDFs are located in
  - `ufs-weather-model/FV3/ccpp/suites` for the UFS Atmosphere
  - `ccpp-scm/ccpp/suites` for the SCM
- Before running, check for consistency between the namelist and the SDF. There is no default consistency check between the SDF and the namelist unless the developer adds one. Errors may result in segmentation faults in running something you did not intend to run if the arrays are not allocated.
- Test and debug the new scheme:
  - Typical problems include segmentation faults related to variables and array allocations.
  - Make sure the SDF and namelist are compatible. Inconsistencies may result in segmentation faults because arrays are not allocated or in unintended scheme(s) being executed.
  - A scheme called GFS_debug (`GFS_debug.F90`) may be added to the SDF where needed to print state variables and interstitial variables. If needed, edit the scheme beforehand to add new variables that need to be printed.
  - Check the *prebuild* script for success/failure and associated messages; run the *prebuild* script with the *–debug* and *–verbose* flags.
  - Compile code in DEBUG mode, run through debugger if necessary (gdb, Allinea DDT, totalview, . . .).
  - Use memory check utilities such as valgrind.
  - Double-check the metadata file associated with your scheme to make sure that all information, including standard names and units, correspond to the correct local variables.

- Done. Note that no further modifications of the build system are required, since the *CCPP Framework* will autogenerate the necessary makefiles that allow the host model to compile the scheme.

CHAPTER

TEN

PARAMETERIZATION-SPECIFIC OUTPUT

10.1 Overview

When used with *UFS* and the *SCM*, the *CCPP* offers the capability of outputting tendencies of temperature, zonal wind, meridional wind, ozone, and specific humidity produced by the *parameterizations* of selected *suites*. This capability is useful for understanding the behavior of the individual parameterizations in terms of magnitude and spatial distribution of tendencies, which can help model developers debug, refine, and tune their *schemes*.

The CCPP also enables outputting two-dimensional (2D) or three-dimensional (3D) arbitrary diagnostics from the parameterizations. This capability is targeted to model developers who may benefit from analyzing intermediate quantities computed in one or more parameterizations. One example of a desirable diagnostic is tendencies from sub-processes within a parameterization, such as the tendencies from condensation, evaporation, sublimation, etc. from a microphysics parameterization. The output is done using CCPP-provided 2D- and 3D arrays, and the developer can fill positions 1, 2, .., N of the array. Important aspects of the implementation are that memory is only allocated for the necessary positions of the array and that all diagnostics are output on physics model levels. An extension to enable output on radiation levels may be considered in future implementations.

These capabilities have been tested and are expected to work with the following suites:

- SCM: GFS_v16, GFS_v17_p8, RAP, RRFS_v1beta, WoFS, HRRR
- ufs-weather-model (regional): GFS_v16, RRFS_v1beta, WoFS, HRRR

10.2 Tendencies

This section describes the tendencies available, how to set the model to prepare them and how to output them. It also contains a list of frequently-asked questions in Section 10.2.6.

10.2.1 Available Tendencies

The model can produce tendencies for temperature, wind, and all non-chemical tracers (see Table 10.1) for several different schemes. Not all schemes produce all tendencies. For example, the orographic and convective gravity wave drag (GWD) schemes produce tendencies of temperature and wind, but not of tracers. Similarly, only the planetary boundary layer (PBL), deep and shallow convection, and microphysics schemes produce specific humidity tendencies. Some PBL and convection schemes will have tendencies for tracers, and others won't.

In addition to the tendencies from specific schemes, the output includes tendencies from all photochemical processes, all physics processes, and all non-physics processes (last three rows of Table 10.2). Examples of non-physical processes are dynamical core processes such as advection and nudging toward climatological fields.

In the supported suites, there are two types of schemes that produce ozone tendencies: PBL and ozone photochemistry. The total tendency produced by the ozone photochemistry scheme (NRL 2015 scheme) is subdivided by subprocesses:

production and loss (combined as a single subprocess), quantity of ozone present in the column above a grid cell, influences from temperature, and influences from mixing ratio. For more information about the NRL 2015 ozone photochemistry scheme, consult the CCPP Scientific Documentation.

There are numerous tendencies in CCPP, and you need to know which ones exist for your configuration to enable them. The model will output a list of available tendencies for your configuration if you run with diagnostic tendencies enabled. To avoid overusing memory, you should enable just one tendency, which is available for all suites, the non-physics (ie. dynamics) tendency of temperature. Details of how to do this, and how to use the information, is below.

10.2.2 Enabling Tendencies

For performance reasons, the preparation of tendencies for output is off by default in the UFS and can be turned on via a set of namelist options. Since the SCM is not operational and has a relatively tiny memory footprint, these tendencies are turned on by default in the SCM.

There are three namelist variables associated with this capability: `ldiag3d`, `qdiag3d`, and `dtend_select`. These are set in the `&gfs_physics_nml` portion of the namelist file `input.nml`.

- `ldiag3d` enables tendencies for state variables (horizontal wind and temperature)
- `qdiag3d` enables tendencies for tracers; `ldiag3d` must also be enabled
- `dtend_select` enables only a subset of the tendencies turned on by `ldiag3d` and `qdiag3d`

If `dtend_select` is not specified, the default is to select all tendencies enabled by the settings of `ldiag3d` and `qdiag3d`.

Note that there is a fourth namelist variable, `lssav`, associated with the output of parameterization-specific information. The value of `lssav` is overwritten to true in the code, so the value used in the namelist is irrelevant.

While the tendencies output by the SCM are instantaneous, the tendencies output by the UFS are averaged over the number of hours specified by the user in variable `fhzero` in the `&gfs_physics_nml` portion of the namelist file `input.nml`. Variable `fhzero` must be an integer (it cannot be zero).

This example namelist selects all tendencies from microphysics processes, and all tendencies of temperature. The naming convention for `dtend_select` is explained in the next section.

```
&gfs_physics_nml
  ldiag3d = .true. ! enable basic diagnostics
  qdiag3d = .true. ! also enable tracer diagnostics
  dtend_select = 'dtend*mp', 'dtend_temp_*' ! Asterisks (*) and question marks (?) have␣
↪the same meaning as shell globs
  ! The default for dtend_select is '*' which selects everything
  ! ... other namelist parameters ...
/
```

10.2.3 Tendency Names

Tendency variables follow the naming pattern below, which is used to enable calculation (`input.nml`) and output of the variable:

```
dtend_variable_process
```

The `dtend_` string stands for "diagnostic tendency" and is used to avoid variable name clashes. Replace `variable` with the short name of the tracer or state variable (see Table 10.1). Replace `process` with the short name of the process that is changing the variable (see Table 10.2). For example, microphysics (`mp`) temperature (`temp`) tendency is `dtend_temp_mp`.

The next section will tell you how to determine which tendency variables are available for your model.

Table 10.1: *Non-chemical tracer and state variables with tendencies. The second column is the* `variable` *part of* `dtend_variable_process`. *The Index column is the first index of* `dtidx`. *Hence "X Wind" is at* `dtend(:,:,dtidx(index_of_x_wind,:))`.

Variable	Short Name	Associated Namelist Variables	dtidx Index	Tendency Units
Temperature	temp	ldiag3d	index_of_temperature	K s^{-1}
X Wind	u	ldiag3d	index_of_x_wind	m s^{-2}
Y Wind	v	ldiag3d	index_of_y_wind	m s^{-2}
Water Vapor Specific Humidity	qv	qdiag3d	100+ntqv	kg kg^{-1} s^{-1}
Ozone Concentration	o3	qdiag3d	100+ntoz	kg kg^{-1} s^{-1}
Cloud Condensate or Liquid Water	liq_wat	qdiag3d	100+ntcw	kg kg^{-1} s^{-1}
Ice Water	ice_wat	qdiag3d	100+ntiw	kg kg^{-1} s^{-1}
Rain Water	rainwat	qdiag3d	100+ntrw	kg kg^{-1} s^{-1}
Snow Water	snowwat	qdiag3d	100+ntsw	kg kg^{-1} s^{-1}
Graupel	graupel	qdiag3d	100+ntgl	kg kg^{-1} s^{-1}
Cloud Amount	cld_amt	qdiag3d	100+ntclamt	kg kg^{-1} s^{-1}
Liquid Number Concentration	water_nc	qdiag3d	100+ntlnc	kg^{-1} s^{-1}
Ice Number Concentration	ice_nc	qdiag3d	100+ntinc	kg^{-1} s^{-1}
Rain Number Concentration	rain_nc	qdiag3d	100+ntrnc	kg^{-1} s^{-1}
Snow Number Concentration	snow_nc	qdiag3d	100+ntsnc	kg^{-1} s^{-1}
Graupel Number Concentration	graupel_nc	qdiag3d	100+ntgnc	kg^{-1} s^{-1}
Turbulent Kinetic Energy	sgs_tke	qdiag3d	100+ntke	J s^{-1}
Mass Weighted Rime Factor	q_rimef	qdiag3d	100+nqrimef	kg kg^{-1} s^{-1}
Number Concentration Of Water-Friendly Aerosols	liq_aero	qdiag3d	100+ntwa	kg^{-1} s^{-1}
Number Concentration Of Ice-Friendly Aerosols	ice_aero	qdiag3d	100+ntia	kg^{-1} s^{-1}
Oxygen Ion Concentration	o_ion	qdiag3d	100+nto	kg kg^{-1} s^{-1}
Oxygen Concentration	o2	qdiag3d	100+nto2	kg kg^{-1} s^{-1}

Table 10.2: *Processes that can change non-chemical tracer and state variables. The third column is the* `process` *part of* `dtend_variable_process`. *The dtidx index is second index of dtidx, hence "Deep Convection" is at* `dtend(:,:,dtidx(:, index_of_process_dcnv))`.

Process	diag_table Module Name	Short Name	dtidx Index
Planetary Boundary Layer	gfs_phys	pbl	index_of_process_pbl
Deep Convection	gfs_phys	deepcnv	index_of_process_dcnv
Shallow Convection	gfs_phys	shalcnv	index_of_process_scnv
Microphysics	gfs_phys	mp	index_of_process_mp
Convective Transport	gfs_phys	cnvtrans	index_of_process_conv_trans
Long Wave Radiation	gfs_phys	lw	index_of_process_longwave
Short Wave Radiation	gfs_phys	sw	index_of_process_shortwave
Orographic Gravity Wave Drag	gfs_phys	orogwd	index_of_process_orographic_gwd
Rayleigh Damping	gfs_phys	rdamp	index_of_process_rayleigh_damping
Convective Gravity Wave Drag	gfs_phys	cnvgwd	index_of_process_nonorographic_gwd
Production and Loss (Photochemical)	gfs_phys	prodloss	index_of_process_prod_loss
Ozone Mixing Ratio (Photochemical)	gfs_phys	o3mix	index_of_process_ozmix
Temperature-Induced (Photochemical)	gfs_phys	temp	index_of_process_temp
Overhead Ozone Column (Photochemical)	gfs_phys	o3column	index_of_process_overhead_ozone
Sum of Photochemical Processes	gfs_phys	photochem	index_of_process_photochem
Sum of Physics Processes (Including Photochemical)	gfs_phys	phys	index_of_process_physics
Sum of Non-Physics Processes	gfs_dyn	nophys	index_of_process_non_physics

10.2.4 Selecting Tendencies

With the many suites and many combinations of schemes, it is hard to say which variable/process combinations are available for your particular configuration. To find a list, enable diagnostics, but disable all tracer/process combinations except one:

```
&gfs_physics_nml
  ldiag3d = .true. ! enable basic diagnostics
  qdiag3d = .true. ! also enable tracer diagnostics
  dtend_select = 'dtend_temp_nophys' ! All configurations have non-physics temperature␣
↪tendencies
  ! ... other namelist parameters ...
/
```

After recompiling and running the model, you will see lines like this in the model's standard output stream:

```
dtend selected: gfs_phys dtend_qv_mp = water vapor specific humidity tendency due to␣
↪microphysics (kg kg-1 s-1)
dtend selected: gfs_phys dtend_liq_wat_mp = cloud condensate (or liquid water) tendency␣
↪due to microphysics (kg kg-1 s-1)
dtend selected: gfs_phys dtend_rainwat_mp = rain water tendency due to microphysics (kg␣
↪kg-1 s-1)
dtend selected: gfs_phys dtend_ice_wat_mp = ice water tendency due to microphysics (kg␣
↪kg-1 s-1)
```

(continues on next page)

(continued from previous page)

```
dtend selected: gfs_phys dtend_snowwat_mp = snow water tendency due to microphysics (kg␣
↪kg-1 s-1)
dtend selected: gfs_phys dtend_graupel_mp = graupel tendency due to microphysics (kg kg-
↪1 s-1)
dtend selected: gfs_phys dtend_cld_amt_mp = cloud amount integer tendency due to␣
↪microphysics (kg kg-1 s-1)
dtend selected: gfs_phys dtend_temp_phys = temperature tendency due to physics (K s-1)
dtend selected: gfs_dyn dtend_temp_nophys = temperature tendency due to non-physics␣
↪processes (K s-1)
```

There are three critical pieces of information in each line. Taking the third last line as an example,

1. `dtend_cld_amt_mp` – this is the name of the variable in `dtend_select`; for the UFS, it is also the name of the variable in the `diag_table`
2. `gfs_phys` – the `diag_table` module name (specific to the UFS, can be ignored for other models)
3. "cloud amount integer tendency due to microphysics" – meaning of the variable

Note that the `dtend_temp_nophys` differs from the others in that it is in the `gfs_dyn` module instead of `gfs_phys` because it sums non-physics processes. This is only relevant for the UFS.

Now that you know what variables are available, you can choose which to enable:

```
&gfs_physics_nml
  ldiag3d = .true. ! enable basic diagnostics
  qdiag3d = .true. ! also enable tracer diagnostics
  dtend_select = 'dtend*mp', 'dtend_temp_*' ! Asterisks (*) and question marks (?) have␣
↪the same meaning as shell globs
  ! The default for dtend_select is '*' which selects everything
  ! ... other namelist parameters ...
/
```

Note that any combined tendencies, such as the total temperature tendency from physics (`dtend_temp_phys`), will only include other tendencies that were calculated. Hence, if you only calculate PBL and microphysics tendencies then your "total temperature tendency" will actually just be the total of PBL and microphysics.

The third step is to enable output of variables, which will be discussed in the next section.

10.2.5 Outputting Tendencies

UFS

After enabling tendency calculation (using `ldiag3d`, `qdiag3d`, and `diag_select`), you must also enable output of those tendencies using the `diag_table`. Enter the new lines with the variables you want output. Continuing our example from before, this will enable output of some microphysics tracer tendencies, and the total tendencies of temperature:

```
"gfs_phys", "dtend_qv_mp",       "dtend_qv_mp",       "fv3_history", "all", .false.,
↪"none", 2
"gfs_phys", "dtend_liq_wat_mp",  "dtend_liq_wat_mp",  "fv3_history", "all", .false.,
↪"none", 2
"gfs_phys", "dtend_rainwat_mp",  "dtend_rainwat_mp",  "fv3_history", "all", .false.,
↪"none", 2
"gfs_phys", "dtend_ice_wat_mp",  "dtend_ice_wat_mp",  "fv3_history", "all", .false.,
```

(continues on next page)

(continued from previous page)

```
↪"none", 2
"gfs_phys", "dtend_snowwat_mp",  "dtend_snowwat_mp",  "fv3_history", "all", .false.,
↪"none", 2
"gfs_phys", "dtend_graupel_mp",  "dtend_graupel_mp",  "fv3_history", "all", .false.,
↪"none", 2
"gfs_phys", "dtend_cld_amt_mp",  "dtend_cld_amt_mp",  "fv3_history", "all", .false.,
↪"none", 2
"gfs_phys", "dtend_temp_phys",   "dtend_temp_phys",   "fv3_history", "all", .false.,
↪"none", 2
"gfs_dyn",  "dtend_temp_nophys", "dtend_temp_nophys", "fv3_history", "all", .false.,
↪"none", 2
```

Note that all tendencies, except non-physics tendencies, are in the `gfs_phys` diagnostic module. The non-physics tendencies are in the `gfs_dyn` module. This is reflected in the Table 10.2.

Note that some *host models*, such as the UFS, have a limit of how many fields can be output in a run. When outputting all tendencies, this limit may have to be increased. In the UFS, this limit is determined by variable `max_output_fields` in namelist section `&diag_manager_nml` in file `input.nml`.

Further documentation of the `diag_table` file can be found in the UFS Weather Model User's Guide.

When the model completes, the fv3_history file will contain these new variables.

SCM

The default behavior of the SCM is to output instantaneous values of all tendency variables, and `dtend_select` is not recognized. Tendencies are computed in file `scm_output.F90` in the subroutines output_init and output_append. If the values of `ldiag3d` or `qdiag3d` are set to false, the variables are still written to output but are given missing values.

10.2.6 FAQ

What is the meaning of error message `max_output_fields` was exceeded?

If the limit to the number of output fields is exceeded, the job may fail with the following message:

```
FATAL from PE    24: diag_util_mod::init_output_field: max_output_fields =          300␣
↪exceeded.  Increase via diag_manager_nml
```

In this case, increase `max_output_fields` in `input.nml`:

```
&diag_manager_nml
    prepend_date = .F.
    max_output_fields = 600
```

10.2.7 Why did I run out of memory when outputting tendencies?

Trying to output all tendencies may use more memory than is available on your system. Use `dtend_select` and choose your output variables carefully!

10.2.8 Why did I get a runtime logic error when outputting tendencies?

Setting `ldiag3d=F` and `qdiag3d=T` will result in an error message:

```
Logic error in GFS_typedefs.F90: qdiag3d requires ldiag3d
```

If you want to output tracer tendencies, you must set both `ldiag3d` and `qdiag3d` to T. Then use `diag_select` to enable only the tendencies you want. Make sure your `diag_table` matches your choice of tendencies specified through `diag_select`.

10.2.9 Why are my tendencies zero, even though the model says they are supported for my configuration?

For total physics or total photochemistry tendencies, see the next question.

The tendencies will be zero if they are never calculated. Check that you enabled the tendencies with appropriate settings of `ldiag3d`, `qdiag3d`, and `diag_select`.

Another possibility is that the tendencies in question really are zero. The list of "available" tendencies is set at the model level, where the exact details of schemes and suites are not known. This can lead to some tendencies erroneously being listed as available. For example, some PBL schemes have ozone tendencies and some don't, so some may have zero ozone tendencies. Also, some schemes don't have tendencies of state variables or tracers. Instead, they modify different variables, which other schemes use to affect the state variables and tracers. Unfortunately, not all of the 3D variables in CCPP have diagnostic tendencies.

10.2.10 Why are my total physics or total photochemistry tendencies zero?

There are three likely reasons:

- You forgot to enable calculation of physics tendencies. Make sure `ldiag3d` and `qdiag3d` are T, and make sure `diag_select` selects physics tendencies.
- The suite did not enable the `phys_tend` scheme, which calculates the total physics and total photochemistry tendencies.
- You did not enable calculation of the individual tendencies, such as ozone. The `phys_tend` sums those to make the total tendencies.

10.3 Output of Auxiliary Arrays from CCPP

The output of diagnostics from one or more parameterizations involves changes to the namelist and code changes in the parameterization(s) (to load the desirable information onto the CCPP-provided arrays and to add them to the subroutine arguments) and in the parameterization metadata descriptor file(s) (to provide metadata on the new subroutine arguments). In the UFS, the namelist is used to control the temporal averaging period. These code changes are intended to be used by scientists during the development process and are not intended to be incorporated into the authoritative code. Therefore, developers must remove any code related to these additional diagnostics before submitting a pull request to the `ccpp-physics` repository.

The auxiliary diagnostics from CCPP are output in arrays:

- aux2d - auxiliary 2D array for outputting diagnostics
- aux3d - auxiliary 3D array for outputting diagnostics

and dimensioned by:

- naux2d - number of 2D auxiliary arrays to output for diagnostics
- naux3d - number of 3D auxiliary arrays to output diagnostics

At runtime, these arrays will be written to the output files. Note that auxiliary arrays can be output from more than one parameterization in a given run.

The UFS and SCM already contain code to declare and initialize the arrays:

- dimensions are declared and initialized in `GFS_typedefs.F90`
- metadata for these arrays and dimensions are defined in `GFS_typedefs.meta`
- arrays are populated in `GFS_diagnostics.F90` (UFS) or `scm_output.F90` (SCM)

The remainder of this section describes changes the developer needs to make in the physics code and in the host model control files to enable the capability. An example (Section 10.3.2) and FAQ (Section 10.3.2) are also provided.

10.3.1 Enabling the auxiliary arrays capability

Physics-side changes

In order to output auxiliary arrays, developers need to change at least the following two files within the physics (see also example in Section 10.3.2):

- **A CCPP entrypoint scheme (Fortran source code)**
  - Add array(s) and its/their dimension(s) to the list of subroutine arguments
  - Declare array(s) with appropriate intent and dimension(s). Note that array(s) do not need to be allocated by the developer. This is done automatically in `GFS_typedefs.F90`.
  - Populate array(s) with desirable diagnostic for output
- **Associated CCPP metadata files for modified scheme(s)**
  - Add entries for the array(s) and its/their dimension(s) and provide metadata

Host-side changes

UFS

For the UFS, developers have to change the following two files on the host side (also see example provided in Section 10.3.2)

- **Namelist file `input.nml`**
  - Specify how many 2D and 3D arrays will be output using variables `naux2d` and `naux3d` in section `&gfs_physics_nml`, respectively. The maximum allowed number of arrays to output is 20 2D and 20 3D arrays.
  - Specify whether the output should be for instantaneous or time-averaged quantities using variables `aux2d_time_avg` and `aux_3d_time_avg`. These arrays are dimensioned `naux2d` and `naux3d`, respectively, and, if not specified in the namelist, take the default value F.

  - Specify the period of averaging for the arrays using variable `fhzero` (in hours).

- **File `diag_table`**
  - Enable output of the arrays at runtime.
  - 2D and 3D arrays are written to the output files.

SCM

Typically, in a 3D model, 2D arrays represent variables with two horizontal dimensions, e.g. x and y, whereas 3D arrays represent variables with all three spatial dimensions, e.g. x, y, and z. For the SCM, these arrays are implicitly 1D and 2D, respectively, where the "y" dimension is 1 and the "x" dimension represents the number of independent columns (typically also 1). For continuity with the UFS Atmosphere, the naming convention 2D and 3D are retained, however. With this understanding, the namelist files can be modified as in the UFS:

- **Namelist file `input.nml`**
  - Specify how many 2D and 3D arrays will be output using variables `naux2d` and `naux3d` in section `&gfs_physics_nml`, respectively. The maximum allowed number of arrays to output is 20 2D and 20 3D arrays.
  - Unlike the UFS, only instantaneous values are output. Time-averaging can be done through post-processing the output. Therefore, the values of `aux2d_time_avg` and `aux_3d_time_avg` should not be changed from their default false values. As such, the namelist variable `fhzero` has no effect in the SCM.

10.3.2 Recompiling and Examples

The developer must recompile the code after making the source code changes to the CCPP scheme(s) and associated metadata files. Changes in the namelist and diag table can be made after compilation. At compile and runtime, the developer must pick suites that use the scheme from which output is desired.

An example for how to output auxiliary arrays is provided in the rest of this section. The lines that start with "+" represent lines that were added by the developer to output the diagnostic arrays. In this example, the developer modified the Grell-Freitas (GF) cumulus scheme to output two 2D arrays and one 3D array. The 2D arrays are `aux_2d (:,1)` and `aux_2d(:,2)`; the 3D array is `aux_3d(:,:,1)`. The 2D array `aux2d(:,1)` will be output with an averaging in time in the UFS, while the `aux2d(:,2)` and `aux3d` arrays will not be averaged.

In this example, the arrays are populated with bogus information just to demonstrate the capability. In reality, a developer would populate the array with the actual quantity for which output is desirable.

```
diff --git a/physics/cu_gf_driver.F90 b/physics/cu_gf_driver.F90
index 927b452..aed7348 100644
--- a/physics/cu_gf_driver.F90
+++ b/physics/cu_gf_driver.F90
@@ -76,7 +76,8 @@ contains
                flag_for_scnv_generic_tend,flag_for_dcnv_generic_tend,          &
                du3dt_SCNV,dv3dt_SCNV,dt3dt_SCNV,dq3dt_SCNV,                    &
                du3dt_DCNV,dv3dt_DCNV,dt3dt_DCNV,dq3dt_DCNV,                    &
-               ldiag3d,qdiag3d,qci_conv,errmsg,errflg)
+               ldiag3d,qdiag3d,qci_conv,errmsg,errflg,                         &
+               naux2d,naux3d,aux2d,aux3d)
  !-----------------------------------------------------------------
       implicit none
```

(continues on next page)

(continued from previous page)

```
       integer, parameter :: maxiens=1
@@ -137,6 +138,11 @@ contains
    integer, intent(in   ) :: imfshalcnv
    character(len=*), intent(out) :: errmsg
    integer,          intent(out) :: errflg
+
+   integer, intent(in) :: naux2d,naux3d
+   real(kind_phys), intent(inout) :: aux2d(:,:)
+   real(kind_phys), intent(inout) :: aux3d(:,:,:)
+
 !  define locally for now.
    integer, dimension(im),intent(inout) :: cactiv
    integer, dimension(im) :: k22_shallow,kbcon_shallow,ktop_shallow
@@ -199,6 +205,11 @@ contains
   ! initialize ccpp error handling variables
      errmsg = ''
      errflg = 0
+
+     aux2d(:,1) = aux2d(:,1) + 1
+     aux2d(:,2) = aux2d(:,2) + 2
+     aux3d(:,:,1) = aux3d(:,:,1) + 3
+
 !
 ! Scale specific humidity to dry mixing ratio
 !
```

The `cu_gf_driver.meta` file was modified accordingly:

```
diff --git a/physics/cu_gf_driver.meta b/physics/cu_gf_driver.meta
index 99e6ca6..a738721 100644
--- a/physics/cu_gf_driver.meta
+++ b/physics/cu_gf_driver.meta
@@ -476,3 +476,29 @@
   type = integer
   intent = out
+[naux2d]
+  standard_name = number_of_2d_auxiliary_arrays
+  long_name = number of 2d auxiliary arrays to output (for debugging)
+  units = count
+  dimensions = ()
+  type = integer
+[naux3d]
+  standard_name = number_of_3d_auxiliary_arrays
+  long_name = number of 3d auxiliary arrays to output (for debugging)
+  units = count
+  dimensions = ()
+  type = integer
+[aux2d]
+  standard_name = auxiliary_2d_arrays
+  long_name = auxiliary 2d arrays to output (for debugging)
+  units = none
+  dimensions = (horizontal_loop_extent,number_of_3d_auxiliary_arrays)
```

(continues on next page)

(continued from previous page)

```
+  type = real
+  kind = kind_phys
+[aux3d]
+  standard_name = auxiliary_3d_arrays
+  long_name = auxiliary 3d arrays to output (for debugging)
+  units = none
+  dimensions = (horizontal_loop_extent,vertical_layer_dimension,number_of_3d_auxiliary_
↪arrays)
+  type = real
+  kind = kind_phys
```

The following lines were added to the `&gfs_physics_nml` section of the namelist file `input.nml`:

```
naux2d         = 2
naux3d         = 1
aux2d_time_avg = .true., .false.
```

Recall that for the SCM, `aux2d_time_avg` should not be set to true in the namelist.

Lastly, the following lines were added to the `diag_table` for UFS:

```
# Auxiliary output
"gfs_phys",    "aux2d_01",     "aux2d_01",      "fv3_history2d",  "all",  .false.,  "none
↪",  2
"gfs_phys",    "aux2d_02",     "aux2d_02",      "fv3_history2d",  "all",  .false.,  "none
↪",  2
"gfs_phys",    "aux3d_01",     "aux3d_01",      "fv3_history",    "all",  .false.,  "none
↪",
```

FAQ

How do I enable the output of diagnostic arrays from multiple parameterizations in a single run?

Suppose you want to output two 2D arrays from schemeA and two 2D arrays from schemeB. You should set the namelist to `naux2d=4` and `naux3d=0`. In the code for schemeA, you should populate `aux2d(:,1)` and `aux2d(:,2)`, while in the code for scheme B you should populate `aux2d(:,3)` and `aux2d(:,4)`.

CHAPTER

ELEVEN

DEBUGGING WITH CCPP

11.1 Introduction

In order to debug code efficiently with *CCPP*, it is important to remember the conceptual differences between traditional, physics-driver based approaches and the ones with CCPP.

Traditional, physics-driver based approaches rely on hand-written physics drivers that connect the different physical *parameterizations* together and often contain a large amount of "glue code" required between the parameterizations. As such, the physics drivers usually have access to all variables that are used by the physical parameterizations, while individual parameterizations only have access to the variables that are passed in. Debugging either happens on the level of the physics driver or inside physical parameterizations. In both cases, print statements are inserted in one or more places (e.g. in the driver before/after parameterizations to debug). In the CCPP, there are no hand-written physics drivers. Instead, the physical parameterizations are glued together by an *SDF* that lists the *primary physical parameterizations* and so-called *interstitial parameterizations* or interstitial schemes (containing the glue code, broken up into logical units) in the order of execution.

11.2 Two categories of debugging with CCPP

- **Debugging inside physical parameterizations**
  Debugging the actual physical parameterizations is identical in CCPP and in physics-driver based models. The parameterizations have access to the same data and debug print statements can be added in exactly the same way.
- **Debugging on a suite level**
  Debugging on a *suite* level, i.e. outside physical parameterizations, corresponds to debugging on the physics-driver level in traditional, physics-driver based models. In the CCPP, this can be achieved by using dedicated CCPP-compliant debugging schemes, which have access to all the data by requesting them via the metadata files. These schemes can then be called in any place in an SDF, except the `fast_physics` group, to produce the desired debugging output. The advantage of this approach is that debugging schemes can be moved from one place to another or duplicated by simply moving/copying a single line in the SDF before recompiling the code. The disadvantage is that different debugging schemes may be needed, depending on the *host model* and their data structures. For example, the *UFS* models use blocked data structures. The blocked data structures are commonly known as "GFS types", are defined in `GFS_typedefs.F90` and exposed to the CCPP in `GFS_typedefs.meta`. The rationale for this storage model is a better cache reuse by breaking up contiguous horizontal grid columns into N blocks with a predefined block size, and allocating each of the GFS types N times. For example, the 3-dimensional air temperature is stored as

```
GFS_data(nb)%Statein%tgrs(1:IM,1:LM) with blocks nb=1,...,N
```

Further, the UFS models run a subset of physics inside the dynamical core ("*fast physics*"), for which the host model data is stored inside the dynamical core and cannot be shared with the traditional ("*slow*") physics. As such, different debugging schemes are required for the `fast_physics` group.

11.3 CCPP-compliant debugging schemes for the UFS

For the UFS models, dedicated debugging schemes have been created by the CCPP developers. These schemes can be found in `FV3/ccpp/physics/physics/GFS_debug.F90`. Developers can use the schemes as-is or customize and add to them as needed. Also, several customization options are documented at the top of the file. These mainly deal with the amount/type of data/information output from arrays, and users can switch between them by turning on or off the corresponding preprocessor directives inside `GFS_debug.F90`, followed by recompiling.

11.3.1 Descriptions of the CCPP-compliant debugging schemes for the UFS

- **`GFS_diagtoscreen`**
  This scheme loops over all blocks for all GFS types that are persistent from one time step to the next (except `GFS_control`) and prints data for almost all constituents. The call signature and rough outline for this scheme is:

```
subroutine GFS_diagtoscreen_run (Model, Statein, Stateout, Sfcprop,␣
↪Coupling,     &
                                 Grid, Tbd, Cldprop, Radtend, Diag,␣
↪Interstitial, &
                                 nthreads, blkno, errmsg, errflg)
 ! Model / Control - only timestep information for now
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d, 'Model
↪%kdt', Model%kdt)
! Sfcprop
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d,
↪'Sfcprop%slmsk', Sfcprop%slmsk)
...
! Radtend
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d,
↪'Radtend%sfcfsw%upfxc', Radtend%sfcfsw(:)%upfxc)
...
!Tbd
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d, 'Tbd
↪%icsdsw', Tbd%icsdsw)
...
! Diag
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d, 'Diag
↪%srunoff', Diag%srunoff)
...
! Statein
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d,
↪'Statein%phii', Statein%phii)
! Stateout
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d,
↪'Stateout%gu0', Stateout%gu0)
...
! Coupling
```

(continues on next page)

(continued from previous page)

```
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d,
↪'Coupling%nirbmdi', Coupling%nirbmdi)
...
! Grid
call print_var(mpirank, omprank, blkno, Grid%xlat_d, Grid%xlon_d, 'Grid
↪%xlon', Grid%xlon)
...
end subroutine GFS_diagtoscreen_run
```

All output to `stdout/stderr` from this routine is prefixed with **'XXX: '** so that it can be easily removed from the log files using "grep -ve 'XXX: ' ..." if needed.

- **GFS_interstitialtoscreen**
  This scheme is identical to `GFS_diagtoscreen`, except that it prints data for all constituents of the `GFS_interstitial` derived data type only. As for `GFS_diagtoscreen`, the amount of information printed to screen can be customized using preprocessor statements, and all output to `stdout/stderr` from this routine is prefixed with **'XXX: '** so that it can be easily removed from the log files using "grep -ve 'XXX: ' ..." if needed.

- **GFS_abort**
  This scheme can be used to terminate a model run at some point in the call to the physics. It can be customized to meet the developer's requirements.

```
subroutine GFS_abort_run (Model, blkno, errmsg, errflg)
    use machine,               only: kind_phys
    use GFS_typedefs,          only: GFS_control_type
    implicit none

    !--- interface variables
    type(GFS_control_type),   intent(in   ) :: Model
    integer,                  intent(in   ) :: blkno
    character(len=*),         intent(  out) :: errmsg
    integer,                  intent(  out) :: errflg
    ! Initialize CCPP error handling variables
    errmsg = ''
    errflg = 0
    if (Model%kdt==1 .and. blkno==size(Model%blksz)) then
        if (Model%me==Model%master) write(0,*) "GFS_abort_run: ABORTING MODEL"
        call sleep(10)
        stop
    end if
 end subroutine GFS_abort_run
```

- **GFS_checkland**
  This routine is an example of a user-provided debugging scheme that is useful for solving issues with the fractional grid with the Rapid Update Cycle Land Surface Model (RUC LSM). All output to `stdout/stderr` from this routine is prefixed with **'YYY: '** (instead of 'XXX:'), which can be easily removed from the log files using "grep -ve 'YYY: ' ..." if needed.

```
subroutine GFS_checkland_run (me, master, blkno, im, kdt, iter, flag_iter, flag_
↪guess, &
                              flag_init, flag_restart, frac_grid, isot, ivegsrc,␣
↪stype, vtype, slope,        &
```

(continues on next page)

(continued from previous page)

```
                            soiltyp, vegtype, slopetyp, dry, icy, wet, lake,␣
↪ocean,                        &
                            oceanfrac, landfrac, lakefrac, slmsk, islmsk,␣
↪errmsg, errflg )
 ...
 do i=1,im
 !if (vegtype(i)==15) then
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, flag_iter(i)  :', i, blkno, flag_
↪iter(i)
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, flag_guess(i) :', i, blkno, flag_
↪guess(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, stype(i)      :', i, blkno,␣
↪stype(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, vtype(i)      :', i, blkno,␣
↪vtype(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, slope(i)      :', i, blkno,␣
↪slope(i)
     write(0,'(a,2i5,1x,i5)')   'YYY: i, blk, soiltyp(i)    :', i, blkno,␣
↪soiltyp(i)
     write(0,'(a,2i5,1x,i5)')   'YYY: i, blk, vegtype(i)    :', i, blkno,␣
↪vegtype(i)
     write(0,'(a,2i5,1x,i5)')   'YYY: i, blk, slopetyp(i)   :', i, blkno,␣
↪slopetyp(i)
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, dry(i)        :', i, blkno, dry(i)
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, icy(i)        :', i, blkno, icy(i)
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, wet(i)        :', i, blkno, wet(i)
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, lake(i)       :', i, blkno,␣
↪lake(i)
     write(0,'(a,2i5,1x,1x,l)') 'YYY: i, blk, ocean(i)      :', i, blkno,␣
↪ocean(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, oceanfrac(i)  :', i, blkno,␣
↪oceanfrac(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, landfrac(i)   :', i, blkno,␣
↪landfrac(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, lakefrac(i)   :', i, blkno,␣
↪lakefrac(i)
     write(0,'(a,2i5,1x,e16.7)')'YYY: i, blk, slmsk(i)      :', i, blkno,␣
↪slmsk(i)
     write(0,'(a,2i5,1x,i5)')   'YYY: i, blk, islmsk(i)     :', i, blkno,␣
↪islmsk(i)
     !end if
 end do
```

11.3.2 How to use these debugging schemes for the UFS

Below is an example for an SDF that prints debugging output from the standard/persistent GFS types and the interstitial type in two places in the radiation group before aborting. Remember that the model loops through each group N block number of times (with potentially M different threads), hence the need to configure `GFS_abort_run` correctly (in the above example, it aborts for the last block, which is either the last loop or in the last group of the threaded loop).

```
<?xml version="1.0" encoding="UTF-8"?>

<suite name="FV3_GFS_v16" version="1">
<!-- <init></init> -->
<group name="fast_physics">
  ...
</group>
<group name="time_vary">
  ...
</group>
<group name="radiation">
  <subcycle loop="1">
    <scheme>GFS_suite_interstitial_rad_reset</scheme>
    <scheme>GFS_diagtoscreen</scheme>
    <scheme>GFS_interstitialtoscreen</scheme>
    <scheme>GFS_rrtmg_pre</scheme>
    <scheme>GFS_radiation_surface</scheme>
    <scheme>rad_sw_pre</scheme>
    <scheme>rrtmg_sw</scheme>
    <scheme>rrtmg_sw_post</scheme>
    <scheme>rrtmg_lw_pre</scheme>
    <scheme>rrtmg_lw</scheme>
    <scheme>rrtmg_lw_post</scheme>
    <scheme>GFS_rrtmg_post</scheme>
    <scheme>GFS_diagtoscreen</scheme>
    <scheme>GFS_interstitialtoscreen</scheme>
    <scheme>GFS_abort</scheme>
  </subcycle>
</group>
<group name="physics">
  ...
</group>
<group name="stochastics">
  ...
</group>
<!-- <finalize></finalize> -->
</suite>
```

Note: Users should be aware that the additional debugging output slows down model runs. It is recommended to reduce the forecast length (as often done for debugging purposes) or increase the walltime limit to debug efficiently. Other options to reduce the size of the output written to stdout/stderr is to use fewer MPI tasks, no OpenMP threading, or to set the blocksize such that each MPI task only has one block.

11.3.3 How to customize the debugging schemes and the output for arrays in the UFS

At the top of `GFS_debug.F90`, there are customization options in the form of preprocessor directives (CPP `#ifdef` etc statements) and a brief documentation. Users not familiar with preprocessor directives are referred to the available documentation such as Using fpp Preprocessor Directives.

Currently three options exist: (1) full output of every element of each array if none of the #define preprocessor statements is used, (2) minimum, maximum, and mean value of arrays (default for GNU compiler), and (3) minimum, maximum, and 32-bit Adler checksum of arrays (default for Intel compiler). Note that Option (3), the Adler checksum calculation, cannot be used with gfortran (segmentation fault, bug in malloc?).

```
!> \file GFS_debug.F90
!!
!! This is the place to switch between different debug outputs.
!! - The default behavior for Intel (or any compiler other than GNU)
!!   is to print minimum, maximum and 32-bit Adler checksum for arrays.
!! - The default behavior for GNU is to minimum, maximum and
!!   mean value of arrays, because calculating the checksum leads
!!   to segmentation faults with gfortran (bug in malloc?).
!! - If none of the #define preprocessor statements is used,
!!   arrays are printed in full (this is often impractical).
!! - All output to stdout/stderr from these routines are prefixed
!!   with 'XXX: ' so that they can be easily removed from the log files
!!   using "grep -ve 'XXX: ' ..." if needed.
!! - Only one #define statement can be active at any time (per compiler)
!!
!! Available options for debug output:
!!
!!   #define PRINT_SUM: print minimum, maximum and mean value of arrays
!!
!!   #define PRINT_CHKSUM: minimum, maximum and 32-bit Adler checksum for␣
↪arrays
!!
#ifdef __GFORTRAN__
#define PRINT_SUM
#else
#define PRINT_CHKSUM
#endif
```

CHAPTER

TWELVE

ACRONYMS

Abbreviation	Explanation
aa	Aerosol-aware
API	Application Programming Interface
b4b	Bit-for-bit
CCPP	Common Community Physics Package
CF conventions	Climate and Forecast Metadata Conventions
CPP	C preprocessor
CPT	Climate Process Team
CSAW	Chikira-Sugiyama convection with Arakawa-Wu extension
DDT	Derived Data Type
dycore	Dynamical core
EDMF	Eddy-Diffusivity Mass Flux
EMC	Environmental Modeling Center
eps	Encapsulated PostScript
ESMF	The Earth System Modeling Framework
FMS	Flexible Modeling System
FV3	Finite-Volume Cubed Sphere
GF	Grell-Freitas convective scheme
GFDL	Geophysical Fluid Dynamics Laboratory
GFS	Global Forecast System
GSD	Global Systems Division
HEDMF	Hybrid eddy-diffusivity mass-flux
HTML	Hypertext Markup Language
ICAMS	Interagency Council for Advancing Meteorological Services
IPD	Interoperable Physics Driver
LSM	Land Surface Model
MG	Morrison-Gettelman
MP	Microphysics
MPAS	Model for Prediction Across Scales
MPI	Message Passing Interface
MYNN	Mellor-Yamada-Nakanishi-Niino
NCAR	National Center for Atmospheric Research
NEMS	National Oceanic and Atmospheric Administration (NOAA) Environmental Modeling System
NEMSfv3gfs	National Oceanic and Atmospheric Administration (NOAA) Environmental Modeling System using FV3 dynamic core and GFS physics
NGGPS	Next Generation Global Prediction System
NOAA	National Oceanic and Atmospheric Administration

continues on next page

Table 12.1 – continued from previous page

Abbreviation	Explanation
NRL	Naval Research Laboratory
NSST	Near Sea Surface Temperature ocean scheme
NUOPC	National Unified Operational Prediction Capability
NWP	Numerical Weather Prediction
OpenMP	Open Multi-Processing
PBL	Planetary Boundary Layer
png	Portable Network Graphic
PR	Pull request
PROD	Compiler optimization flags for production mode
REPRO	Compiler optimization flags for reproduction mode (bit-for-bit testing)
RRFS	Rapid Refresh Forecast System
RRTMG	Rapid Radiative Transfer Model for Global Circulation Models
RT	Regression test
RUC	Rapid Update Cycle
sa	Scale-aware
SAS	Simplified Arakawa-Schubert
SCM	Single Column Model
SDF	Suite Definition File
sfc	Surface
SHUM	Perturbed boundary layer specific humidity
SKEB	Stochastic Kinetic Energy Backscatter
SPPT	Stochastically Perturbed Physics Tendencies
TKE	Turbulent Kinetic Energy
TWP-ICE	Tropical Warm Pool International Cloud Experiment
UML	Unified Modeling Language
UFS	Unified Forecast System
WRF	Weather Research and Forecasting

CHAPTER

THIRTEEN

GLOSSARY

CCPP
: The topic of this technical guide, the Common Community Physics Package (CCPP) is a model-agnostic, well-vetted collection of codes containing atmospheric physical parameterizations and suites for use in NWP along with a framework that connects the physics to a *host model*

CCPP Framework
: The infrastructure that connects physics *schemes* with a *host model*; also refers to a software repository of the same name

CCPP Physics
: The pool of CCPP-compliant physics schemes; also refers to a software repository of the same name

Fast physics
: Physical parameterizations that require tighter coupling with the dynamical core than "slow" physics (due to the approximated processes within the parameterization acting on a shorter timescale) and that benefit from a smaller time step. The distinction is useful for greater accuracy, numerical stability, or both. In the UFS Atmosphere, a saturation adjustment is used in some suites and is called directly from the dynamical core for tighter coupling

Group
: A set of physics *schemes* within a suite definition file (SDF) that are called together without intervening computations from the *host application*. Groups are described in more detail in `Chapter %c`.

Group *cap*
: Autogenerated interface between a *group* of physics schemes and the *host model*. Caps are described in more detail in `Chapter %c`.

Host model
: A host model (or host application) is an atmospheric model or application that allocates memory, provides metadata for the variables passed into and out of the physics, and controls time-stepping

Interstitial scheme
: A modularized piece of code to perform data preparation, diagnostics, or other "glue" functions that allows primary schemes to work together as a suite. They can be categorized as "scheme-specific" or "suite-level". Scheme-specific interstitial schemes augment a specific primary scheme (to provide additional functionality). Suite-level interstitial schemes provide additional functionality on top of a class of primary schemes, connect two or more schemes together, or provide code for conversions, initializing sums, or applying tendencies, for example. The details of primary vs. interstitial schemes are described in more detail in `Chapter %c`.

NCAR
: The National Center for Atmospheric Research - a US federally funded research and development center (FFRDC) managed by the University Corporation for Atmospheric Research (UCAR) and funded by the National Science Foundation (NSF).

NEMS
: The NOAA Environmental Modeling System - a software infrastructure that supports NCEP/EMC's forecast

products. The coupling software is based on ESMF and the NUOPC layer.

Parameterization
The representation, in a dynamic model, of physical effects in terms of admittedly oversimplified parameters, rather than realistically requiring such effects to be consequences of the dynamics of the system (from the AMS Glossary)

Phase
A CCPP phase is one of five steps that each physics *scheme* can be broken down into. Phases are described in more detail in `Chapter %c`.

Physics *cap*
Generic name to refer to *suite* and *group physics caps*.

Physics Suite *cap*
Autogenerated interface between an entire *suite* of physics schemes and the *host model*. It consists of calls to autogenerated physics group caps. It may be used to call an entire suite at once or to call a specific group within a physics suite

Primary scheme
A parameterization, such as PBL, microphysics, convection, and radiation, that fits the traditionally-accepted definition, as opposed to an interstitial scheme

Scheme
A CCPP-compliant parameterization (*primary scheme*) or auxiliary code (*interstitial scheme*)

SDF
Suite Definition File (SDF) is an external file containing information about the construction of a physics *suite*. It describes the *schemes* that are called, in which order they are called, whether they are subcycled, and whether they are assembled into groups to be called together

Set
A collection of physics *schemes* that do not share memory (e.g. fast and slow physics)

SCM
The *CCPP* Single Column Model (SCM) is a simple 1D *host model* designed to be used with the CCPP Physics and Framework as a lightweight alternative to full 3D dynamical models for testing and development of physics *schemes* and *suites*. See the SCM User Guide for more information.

Slow physics
Physical parameterizations that can tolerate looser coupling with the dynamical core than "fast" physics (due to the approximated processes within the parameterization acting on a longer timescale) and that often use a longer time step. Such parameterizations are typically grouped and calculated together (through a combination of process- and time-splitting) in a section of an atmospheric model that is distinct from the dynamical core in the code organization

Standard name
Variable names based on CF conventions (http://cfconventions.org) that are uniquely identified by the *CCPP-compliant schemes* and provided by a *host model*. See Section 2.3 for more details.

Subcycling
Executing a physics *scheme* more frequently (with a shorter timestep) than the rest of the model physics or dynamics. See Section 4.1.2 for more details.

Suite
A collection of *primary physics schemes* and *interstitial schemes* that are known to work well together

UFS
A Unified Forecast System (UFS) is a community-based, coupled comprehensive Earth system modeling system. The UFS numerical applications span local to global domains and predictive time scales from sub-hourly analyses

to seasonal predictions. It is designed to support the Weather Enterprise and to be the source system for NOAA's operational numerical weather prediction applications

UFS Atmosphere
The atmospheric model component of the *UFS*. Its fundamental parts are the dynamical core and the physics

UFS Weather Model
The combined global/regional medium- to short-range weather-prediction model used in the *UFS* to create forecasts

INDEX