
v4 CCPP Technical Documentation

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

Nov 16, 2020

CONTENTS

1	CCPP Overview	1
1.1	How to Use this Document	4
2	CCPP-Compliant Physics Parameterizations	7
2.1	General Rules	8
2.2	Metadata Rules	9
2.3	Input/output Variable (argument) Rules	11
2.4	Coding Rules	12
2.5	Parallel Programming Rules	13
2.6	Scientific Documentation Rules	13
2.6.1	Doxygen Comments and Commands	14
2.6.2	Doxygen Documentation Style	14
2.6.3	Doxygen Configuration	20
2.6.4	Including metadata information	21
2.6.5	Using Doxygen	23
3	CCPP Configuration and Build Options	25
4	Constructing Suites	29
4.1	Suite Definition File	29
4.1.1	Groups	29
4.1.2	Subcycling	30
4.1.3	Order of Schemes	30
4.2	Interstitial Schemes	31
4.3	SDF Examples	31
4.3.1	Simplest Case: Single Group and no Subcycling	31
4.3.2	Case with Multiple Groups	32
4.3.3	Case with Subcycling	32
4.3.4	GFS v16beta Suite	32
5	Autogenerated Physics <i>Caps</i>	37
5.1	Dynamic Build <i>Caps</i>	37
5.2	Static Build <i>Caps</i>	40
5.3	Automatic unit conversions	44
6	Host Side Coding	47
6.1	Variable Requirements on the Host Model Side	47
6.2	Metadata for Variable in the Host Model	47
6.3	CCPP Variables in the SCM and UFS Atmosphere Host Models	50
6.4	CCPP API	51
6.4.1	Data Structure to Transfer Variables between Dynamics and Physics	52

6.4.2	Adding and Retrieving Information from cdata (dynamic build option)	53
6.4.3	Initializing and Finalizing the CCpp	54
6.4.4	Running the physics	54
6.4.5	Initializing and Finalizing the Physics	55
6.5	Host Caps	56
6.5.1	SCM Host Cap	58
6.5.2	UFS Atmosphere Host Cap	59
7	CCPP Code Management	63
7.1	Organization of the Code	63
7.1.1	Authoritative Repositories	63
7.1.2	Directory Structure of ccpp/framework	63
7.1.3	Directory Structure of ccpp/physics	64
7.2	GitHub Workflow (setting up development repositories)	64
7.2.1	Creating Forks	64
7.2.2	Checking out the Code	64
7.3	Committing Changes to your Fork	65
7.4	Contributing Code, Code Review Process	66
7.4.1	Creating a PR	66
8	Technical Aspects of the CCpp Prebuild	69
8.1	<i>Prebuild</i> Script Function	69
8.2	Script Configuration	71
8.3	Running ccpp_prebuild.py	72
8.4	Troubleshooting	73
9	Tips for Adding a New Scheme	79
10	Acronyms	83
11	Glossary	85
	Index	87

CCPP OVERVIEW

Note: The information in this document corresponds to the CCPP v4 release. To obtain the latest available Technical Documentation go to <https://ccpp-techdoc.readthedocs.io/en/latest/>.

Ideas for this project originated within the Earth System Prediction Capability (ESPC) physics interoperability group, 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 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 architecture of the CCPP and its connection to a host model is shown in [Figure 1.1](#). There are two distinct parts to the CCPP: 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 host model needs to have functional documentation 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 model¹, 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.

There are multiple options to build the CCPP (see more detail in [Chapter 3](#)). For example, with the CCPP dynamic build, all the CCPP-compliant parameterizations are compiled into a library which is linked to the host model at runtime. Conversely, with the CCPP static build, one or more physics suites are compiled into a library and linked to the host model when it is compiled. The dynamic build favors flexibility as users can select the parameterizations and their order at runtime, while the static build favors performance as it provides superior execution time and a smaller memory footprint. The type of build defines several differences in the creation and use of the auto-generated code, many of which are not exposed to the user. The differences pertain to the interfaces between CCPP-Framework and the physics (physics *caps*) and the host model (host model *cap*), as well as in the procedures for calling the physics. In addition, the building option varies with choice of the host model. The only CCPP build option supported for use with the CCPP Single-Column Model v4.0 or with the UFS Medium-Range Weather Application v1.0 is the static build. The dynamic build will be phased out in a future release of the CCPP.

¹ As of this writing, the CCPP has been validated with two host models: the CCPP Single Column Model (SCM) and the atmospheric component of NOAA’s Unified Forecast System (UFS) (hereafter the UFS Atmosphere) that utilizes the Finite-Volume Cubed Sphere (FV3) dycore. The CCPP can be utilized both with the global and standalone regional 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.

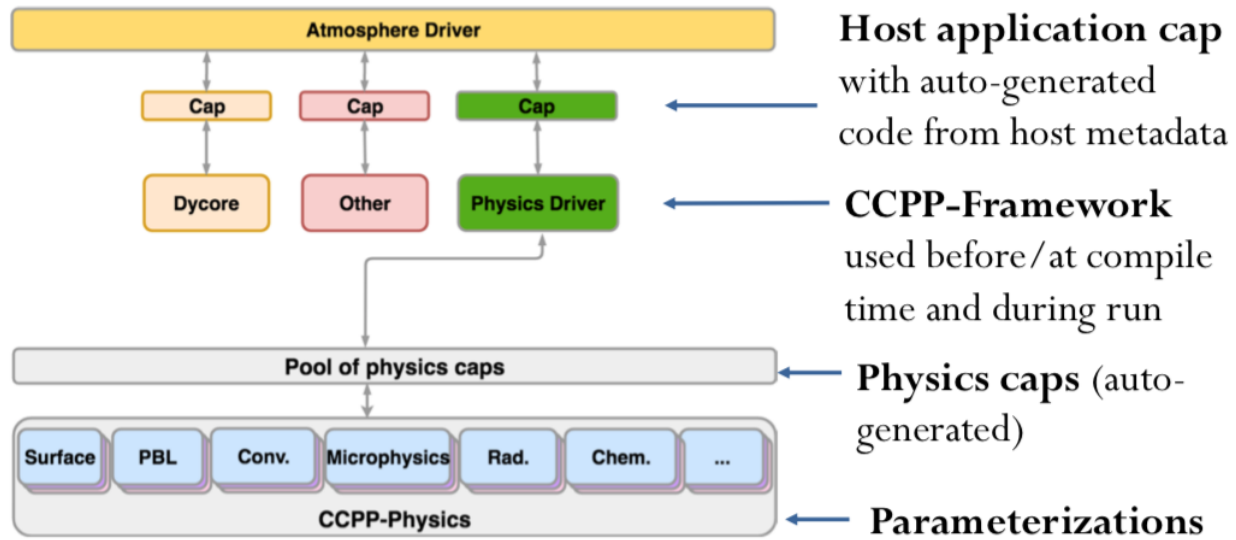


Fig. 1.1: Architecture of the CCPP and its connection to a host model, represented here as the driver for an atmospheric model (yellow box). The dynamical core (dycore), physics, and other aspects of the model (such as coupling) are connected to the driving host through the pool of physics caps. The 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-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>.

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 release, issued in March 2020, contains suite GFS_v15p2, which is an updated version of the operational GFS v15 and replaces suite GFS_v15. It also contains three developmental suites: csawmg has minor updates, GSD_v1 is an update over the previously released GSD_v0, and GFS_v16beta is the target suite for implementation in the upcoming operational GFSv16 (it replaces suite GFSv15plus). Additionally, there are two new suites, GFS_v15p2_no_nsst and GFS_v16beta_no_nsst, which are variants that treat the sea surface temperature more simply. These variants are recommended for use when the initial conditions do not contain all fields needed to initialize the more complex Near Sea Surface Temperature (NSST) scheme. The *CCPP Scientific Documentation* describes the suites and their parameterizations in detail.

The CCPP is governed by the groups that contribute to its development. The governance of the CCPP-Physics is currently led by NOAA, and the DTC works with EMC and the Next Generation Global Prediction System (NG-GPS) Program Office to determine which 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/>

Common Community Physics Package (CCPP) Ecosystem

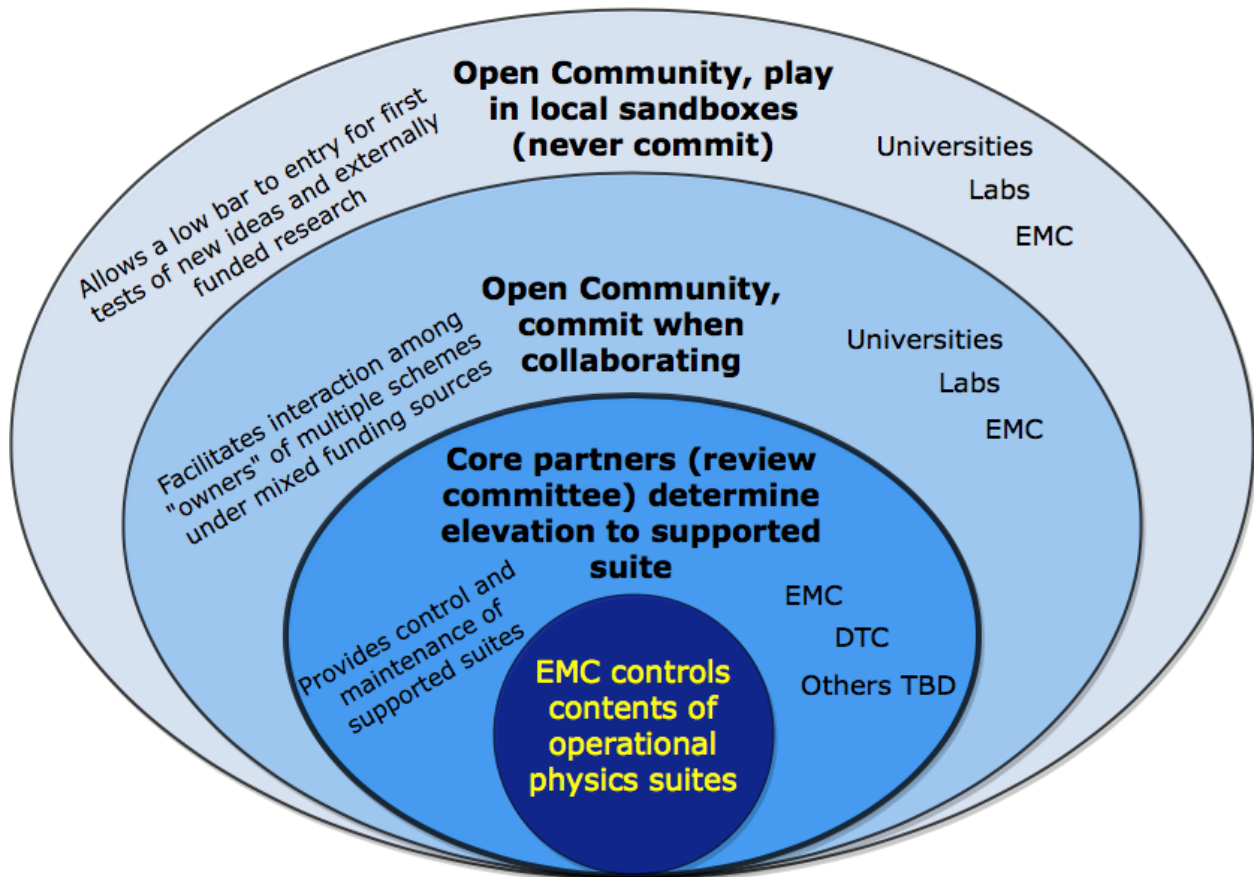


Fig. 1.2: CCPP ecosystem.

ccpp-framework/wiki and <https://dtcenter.org/gmtb/users/ccpp>). Please direct all inquiries to gmtb-help@ucar.edu.

Table 1.1: Suites supported in the CCpp

	Operational	Experimental			Variants	
	GFS_v15p2	GFS_v16beta	csawmg	GSD_v1	GFS_v15p2_no_nsst	GFS_v16beta_no_nsst
Microphysics	GFDL	GFDL	M-G3	Thompson	GFDL	GFDL
PBL	K-EDMF	TKE EDMF	K-EDMF	saMYNN	K-EDMF	TKE EDMF
Deep convection	saSAS	saSAS	CSAW	GF	saSAS	saSAS
Shallow convection	saSAS	saSAS	saSAS	saMYNN and saSAS	saSAS	saSAS
Radiation	RRTMG	RRTMG	RRTMG	RRTMG	RRTMG	RRTMG
Surface layer	GFS	GFS	GFS	GFS	GFS	GFS
Gravity Wave Drag	uGWD	uGWD	uGWD	uGWD	uGWD	uGWD
Land surface	Noah	Noah	Noah	RUC	Noah	Noah
Ozone	NRL 2015	NRL 2015	NRL 2015	NRL 2015	NRL 2015	NRL 2015
H ₂ O	NRL 2015	NRL 2015	NRL 2015	NRL 2015	NRL 2015	NRL 2015
Ocean	NSST	NSST	NSST	NSST	sfc_ocean	sfc_ocean

The suites that are currently supported in the CCpp are listed in the second row. The types of parameterization are denoted in the first column, where H₂O represents the stratospheric water vapor parameterization. The GFS_v15p2 suite includes the GFDL microphysics, a Eddy-Diffusivity Mass Flux (K-EDMF) planetary boundary layer (PBL) scheme, scale-aware (sa) Simplified Arakawa-Schubert (SAS) convection, Rapid Radiation Transfer Model for General Circulation Models (RRTMG) radiation, the GFS surface layer, the unified gravity wave drag (uGWD), the Noah Land Surface Model (LSM), the 2015 Navy Research Laboratory (NRL) ozone and stratospheric water vapor schemes, and the NSST ocean scheme. The three developmental suites are candidates for future operational implementations. The GFS_v16beta suite is the same as the GFS_v15p2 suite except using the Turbulent Kinetic Energy (TKE)-based EDMF PBL scheme. The Chikira-Sugiyama (csawmg) suite uses the Morrison-Gottelman 3 (M-G3) microphysics scheme and Chikira-Sugiyama convection scheme with Arakawa-Wu extension (CSAW). The NOAA Global Systems Division (GSD) v1 suite (GSD_v1) includes Thompson microphysics, scale-aware Mellor-Yamada-Nakanishi-Niino (saMYNN) PBL and shallow convection, Grell-Freitas (GF) deep convection schemes, and the Rapid Update Cycle (RUC) LSM. The two variants use the sfc_ocean scheme instead of the NSST scheme.

1.1 How to Use this Document

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

- 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

For the latest version of the released code, please visit the [DTC Website](#)

Please send questions and comments to the help desk: gmtb-help@ucar.edu. When using the CCpp with NOAA's Unified Forecast System, questions can also be posted in the UFS Forum at <https://forums.ufscommunity.org/>.

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

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.bashrc</code> Use <code>ls -a</code> to list all files. host\$ You have mail!

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

```
mkdir ${TOP_DIR}
```


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. For example, if a given set 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 that sits on top of the actual scheme and provides 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 put MG2 and MG3 as 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_hrrr.F90` in the `ccpp-physics/physics` directory.
- To deal with optional arguments. A driver can check whether optional arguments have been provided by the host model to either write out a message and return an error code or call a subroutine with or without optional arguments. For example, see `mp_thompson_hrrr.F90`, `radsw_main.f`, or `radlw_main.f` 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` in the `ccpp-physics/physics` directory.

Schemes in the CCPP are classified into two categories: primary schemes and interstitial schemes. Primary schemes are the major parameterizations, such as PBL, microphysics, convection, radiation, surface layer parameterizations, etc. 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.

2.1 General Rules

A CCPP-compliant scheme is in the form of Fortran modules. [Listing 2.1](#) contains the template for a CCPP-compliant scheme (ccpp/framework/doc/DevelopersGuide/scheme_template.F90), which includes three essential components: the `_init`, `_run`, and `_finalize` subroutines. 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

  subroutine scheme_template_init ()
end subroutine scheme_template_init

  subroutine scheme_template_finalize()
end subroutine scheme_template_finalize

!> \section arg_table_scheme_template_run Argument Table
!! \htmlinclude scheme_template_run.html
!!

  subroutine scheme_template_run (errmsg, errflg)

    implicit none

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

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

    continue

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

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

    !--- actual code
    ! add your code here

    ! in case of errors, set errflg to a value != 0,
    ! assign a 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 `_init`, `_run`, and `_finalize` subroutines.

More details are found below:

- Each scheme must be in its own module and must include three (`_init`, `_run`, and `_finalize`) subroutines (entry

points). The module name and the subroutine names must be consistent with the scheme name. The `_init` and `_finalize` subroutines are run automatically when the CCPP-Physics are initialized and finalized, respectively. These two 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). The `_run` subroutine contains the code to execute the scheme.

- Each `.f` or `.F90` file with one or more CCPP entry point schemes must be accompanied by a `.meta` file containing metadata about the arguments to the scheme(s). For more information, see [Section 2.2](#).
- Non-empty schemes must be preceded by the three lines below. These are markup comments used by Doxygen, the software employed to create the scientific documentation, to insert an external file containing metadata information (in this case, `schemename_run.html`) in the documentation. See more on this topic in [Section 2.6](#).

```
!> \section arg_table_schemename_run Argument Table
!! \htmlinclude schemename_run.html
!!
```

- All external information required by the scheme must be passed in via the argument list. Statements such as `'use EXTERNAL_MODULE'` should not be used for passing in data and all physical constants should go through the argument list.
- Note that standard names, variable names, module names, scheme names and subroutine names are all case sensitive.
- Interstitial modules (`scheme_pre` and `scheme_post`) can be included if any part of the physics scheme must be executed before (`_pre`) or after (`_post`) the module `scheme` defined above.

2.2 Metadata Rules

- Metadata files (`.meta`) are in a relaxed config file format and contain metadata for one or more CCPP entry point schemes. There should be one `.meta` file for each `.f` or ```F90``` file.
- For each CCPP compliant scheme, the `.meta` file should have 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`, `DDT`, or `host`.
- For empty schemes, the three lines above are sufficient. For non-empty schemes, 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>
optional = <optional>
```

- 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`, and `optional`.
- 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)
```

- [Listing 2.2](#) contains the template for a CCpp-compliant scheme (`ccpp/framework/doc/DevelopersGuide/scheme_template.meta`),

```
[ccpp-arg-table]
  name = ozphys_init
  type = scheme

#####
[ccpp-arg-table]
  name = ozphys_finalize
  type = scheme

#####
[ccpp-arg-table]
  name = ozphys_run
  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_flag
  long_name = error flag for error handling in CCpp
  units = flag
  dimensions = ()
  type = integer
  intent = out
  optional = F
```

Listing 2.2: Fortran template for a metadata file accompanying a CCpp-compliant scheme.

2.3 Input/output Variable (argument) Rules

- Variables available for CCpp physics schemes are identified by their unique `standard_name`. While an effort is made to comply with existing `standard_name` definitions of the Climate and Forecast (CF) conventions (<http://cfconventions.org>), additional names are used in the CCpp (see below for further information).
- A list of available standard names and an example of naming conventions can be found in `ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_${HOST}.pdf`, where `${HOST}` is the name of the host model. Running the CCpp *prebuild* script (described in [Chapter 8](#)) 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.
- 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
```

- 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 2.4](#)).
- At present, only two types of variable definitions are supported by the CCpp-framework:
 - Standard Intrinsic Fortran variables are preferred (`character`, `integer`, `logical`, `real`). 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). 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 loop counter can be obtained from CCPP as an `intent(in)` variable (see a [mandatory list of variables](#) that are provided by the CCPP-Framework and/or the host model for this and other purposes).

2.4 Coding Rules

- Code must comply to modern Fortran standards (Fortran 90/95/2003).
- Labeled `end` statements should be used for modules, subroutines and functions, 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.
- `goto` statements are not allowed.
- `common` blocks are not allowed.
- 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 set `errflg` to a value other than 0, for example:

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

- Schemes are not allowed to abort/stop the program.
- Schemes are not allowed to perform I/O operations except for reading lookup tables or other information needed to initialize the scheme, 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#heading=h.97v79689onyd).

2.5 Parallel Programming Rules

Most often shared memory (OpenMP: Open Multi-Processing) and MPI (Message Passing Interface) communication are done outside the physics in which case the physics looping 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:

- 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` and `_finalize` phase for the purpose of computing, reading or writing scheme-specific data that is independent of the host model's data decomposition. An example is the initial read of a lookup table of aerosol properties by one or more MPI processes, and its subsequent broadcast to all processes. Several restrictions apply:
 - The implementation of reading and writing of data must be scalable to perform efficiently from a few to millions of tasks.
 - The MPI communicator must be provided by the host model as an `intent (in)` argument in the argument list (*see list of mandatory variables*).
 - The use of `MPI_COMM_WORLD` is not allowed.
- 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.

```
#ifndef 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 DTC helpdesk (gmtb-help@ucar.edu).

2.6 Scientific Documentation Rules

Technically, scientific documentation is not needed for a parameterization to work with the CCpp. However, scientific and technical documents are important for code maintenance and for fostering understanding among stakeholders. As such, it is required of physics schemes 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.org/GMTB/v4.0/sci_doc. 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.

2.6.1 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 purpose is present 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 “<” just 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 !< earliest year 2D CO2 data available
```

2.6.2 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 Fortran source code file level. 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_driver.F90
!! This file is scale-aware Grell-Freitas cumulus scheme driver.
```

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:

 calprecipitype.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 diagnostics 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 file or module. They are external text files that generate pages with a high-level scientific overview and typically contain a longer description of a project or suite. You can refer to any source code entity from within the page.

The DTC maintains a main page, created by the Doxygen command “`\\mainpage`”, containing an overall description and background of the CCpp. Physics developers do not have to edit the file with the mainpage, which has a user-visible title, but not label:

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

(continues on next page)

(continued from previous page)

```
...
*/
```

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 in `suite_FV3_GFS_v15p2.xml.txt`:

The GFS v15p2 physics suite uses the parameterizations in the following order, as defined in

```
\c FV3_GFS_v15p2 :
- \ref fast_sat_adj
- \ref GFS_RRTMG
- \ref GFS_SFCLYR
- \ref GFS_NSST
- \ref GFS_NOAH
- \ref GFS_SFCSICE
- \ref GFS_HEDMF
- \ref cires_ugwp
- \ref GFS_RAYLEIGH
- \ref GFS_OZPHYS
- \ref GFS_H2OPHYS
- \ref GFS_SAMFdeep
- \ref GFS_SAMFshal
- \ref GFDL_cloud
- \ref GFS_CALPRECIPTYPE
- \ref STOCHY_PHYS
```

The HTML result is [here](#). You can see that the “-” signs before “@ref” generate a list with bullets. Doxygen command “\c” displays its argument using a typewriter font.

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”. Scheme pages are stored in the `ccpp-physics/physics/docs/pdftxt` directory. Each 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
```

(continues on next page)

(continued from previous page)

*/

The physics scheme page will often describe the following:

1. Description section ("\\section"), which usually includes:

- Scientific origin and scheme history ("\\cite")
- Key features and differentiating points
- A picture is worth a thousand words ("\\image")

To insert images into doxygen documentation, you'll need to have your images ready 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. Intraphysics Communication Section ("\\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 table in the Fortran code for the scheme.

3. General Algorithm Section ("\\section")

The general description of the algorithm will be in this section. It is created by inserting a reference link ("\\ref") in the Fortran code for the scheme.

The symbols "\/**" and "*/" need to be the first and last entries of the page. See an example of GFS Scale-Aware Simplified Arakawa-Schubert (sa-SAS) Deep Convection Scheme page in the previous 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 module list 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.
```

(continues on next page)

(continued from previous page)

```
!! ...
!!\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
```

- For subroutines that are non-empty, 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). Note that empty subroutines, as is sometimes the case for `init` and `finalize` subroutines, do not require the inclusion of a file with metadata information. 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
!! \param[in] deltax  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,                &
    & z1,zm,thvx,q1,t1,u1,v1,hpbl,kpbl,                        &
    & sflx,ustar,wstar,xf,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 repository, 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 [link](#) 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://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.2
```

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

2.6.3 Doxygen Configuration

Configuration File

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

If starting from scratch, you can generate a default configuration file using the command:

```
doxygen -g <config_file>
```

Then you can edit the default configuration file to serve your needs. The default file includes plenty of comments to explain all the options. Some of the important things you need to pay attention to are:

- The name of your project:

```
PROJECT_NAME = 'your project name'
```

- The input files (relative to the directory where you run doxygen):

```
INPUT =
```

The following lines should be listed here: the doxygen mainpage text file, the scheme pages, and the source codes to be contained in the output. The order in which schemes are listed determines the order in the HTML result.

- The directory where to put the documentation (if you leave it empty, then the documentation will be created in the directory where you run doxygen):

```
OUTPUT_DIRECTORY = doc
```

- The type of documentation you want to generate (HTML, LaTeX and/or something else):

```
GENERATE_HTML = YES
```

If HTML is chosen, the following tells doxygen where to put the documentation relative to the `OUTPUT_DIRECTORY`:

```
HTML_OUTPUT = html
```

and

```
HTML_FILE_EXTENSION = .html
```

determines the extension of the files.

- Other important settings for a Fortran code project are:


```

OPTIMIZE_FOR_FORTRAN      =    YES
EXTENSION_MAPPING         =    .f=FortranFree      \
                          .F90=FortranFree        \
                          .f90=FortranFree
LAYOUT_FILE               =    ccpp_dox_layout.xml
CITE_BIB_FILES            =    library.bib
FILE_PATTERN              =    *.f                \
                          *.F90                  \
                          *.f90                  \
                          *.txt
GENERATE_TREEVIEW         =    yes

```

Doxygen files for layout (`ccpp_dox_layout.xml`), a HTML style (`ccpp_dox_extra_style.css`), and 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 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:

2.6.4 Including metadata information

As described above, a table of metadata information should be included in the documentation for every CCpp entry-point scheme. Before doxygen is run, the table for each scheme must be manually created in separate files in HTML format, with one file per non-empty scheme. The HTML files are included in the Fortran files using the doxygen 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`. The syntax for running this script from the directory above where the CCpp is installed is:

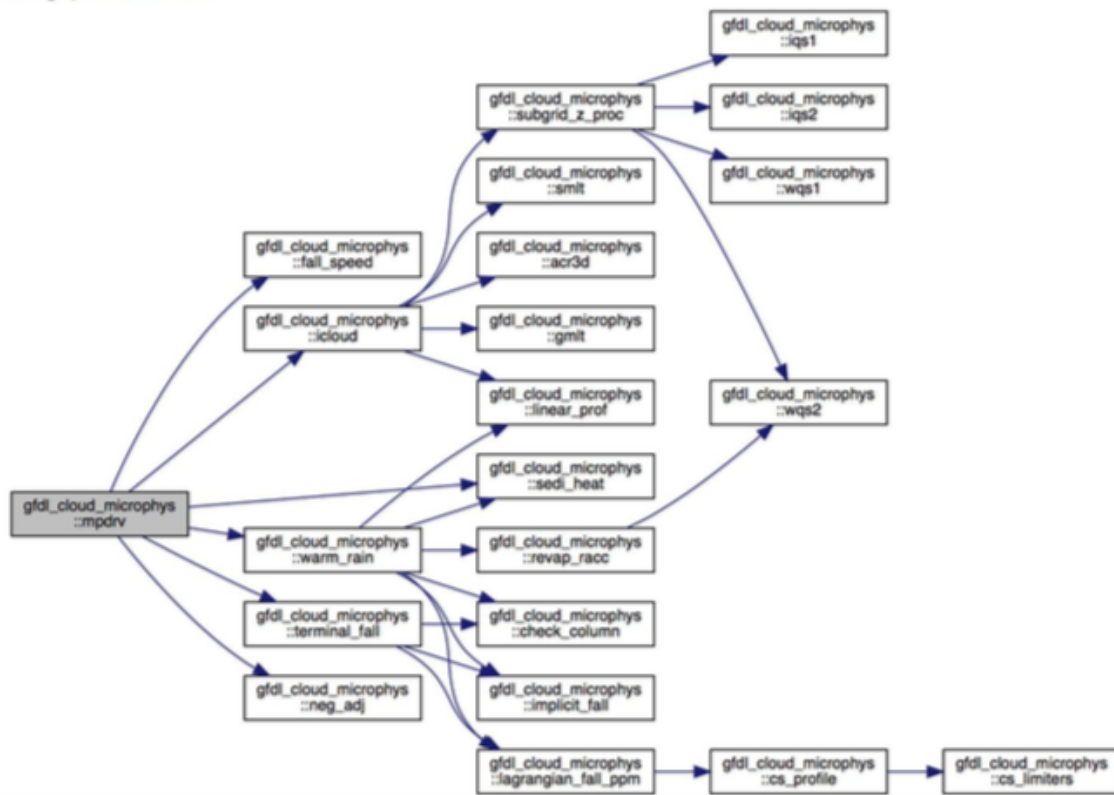
```

./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.

Here is the call graph for this function:



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

To generate a complete Scientific Documentation, documentation, 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:

```
./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. Option `-m` also requires to specify `-o`, while option `-c` will ignore `-o`. For more information, use

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

2.6.5 Using Doxygen

In order to generate the doxygen-based documentation, one needs to follow five steps:

1. Have the doxygen executable installed on your computer. For the NOAA machine Hera and the NCAR machine Cheyenne, the doxygen executable resides in `/usr/bin`, which should be in your `$PATH`. If you need to install doxygen in another location, add the following line into the `.cshrc` file in your home directory:

```
alias doxygen /path/to/doxygen
```

Source your `.cshrc` file.

2. Document your code, including doxygen main page, scheme pages and inline comments within 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 referred to in the physics suites.
5. Create or edit a doxygen configuration file to control what doxygen pages, source files and bibliography file get parsed, how the source files get parsed, and to customize the output.
6. Run doxygen from directory `ccpp/physics/physics/docs` using the command line to specify the doxygen configuration file as an argument:

```
$doxygen $PATH_TO_CONFIG_FILE/<config_file>
```

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 on creating the scientific documentation, contact the DTC helpdesk at gmtb-help@ucar.edu.

CCPP CONFIGURATION AND BUILD OPTIONS

While the *CCPP-Framework* code can be compiled independently, the *CCPP-Physics* code can only be used within a host modeling system that provides the variables and the kind, type, and DDT definitions. As such, it is advisable to integrate the CCPP configuration and build process with the host model's. 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](#).

For this release, the SCM and the UFS Atmosphere are supported for use with the CCPP static build (the dynamic build option is not supported because it will be deprecated soon). In the case of the UFS Atmosphere as the host model, there are several build options ([Figure 3.1](#)). The choice can be specified through command-line options supplied to the `compile.sh` script for manual compilation or through a regression test (RT) configuration file. Detailed instructions for building the code are discussed in `Chapter %s`.

The relevant options for building CCPP with the UFS Atmosphere can be described as follows:

- **Without CCPP (CCPP=N):** The code is compiled without CCPP and runs using the original UFS Atmosphere physics drivers, such as `GFS_physics_driver.F90`. This option entirely bypasses all CCPP functionality and is only used for RT against the unmodified UFS Atmosphere codebase.
- **With CCPP (CCPP=Y):** The code is compiled with CCPP enabled and restricted to CCPP-compliant physics. That is, any parameterization to be called as part of a suite must be available in CCPP. Physics scheme selection and order is determined at runtime by an external suite definition file (SDF; see [Chapter 4](#) for further details on the SDF). The existing physics-calling code `GFS_physics_driver.F90` and `GFS_radiation_driver.F90` are bypassed altogether in this mode and any additional code needed to connect parameterizations within a suite previously contained therein is executed from the so-called CCPP-compliant “interstitial schemes”. One further option determines how the CCPP-compliant physics are called within the host model:
 - **Dynamic CCPP (STATIC=N):** This option is not supported; it allows choosing any physics schemes within the CCPP library at runtime by making adjustments to the CCPP SDF and the model namelist. This option carries computational overhead associated with the higher level of flexibility. Note that the *CCPP-Framework* and *CCPP-physics* are dynamically linked to the executable.
 - **Static CCPP (STATIC=Y):** The code is compiled with CCPP enabled and is restricted to CCPP-compliant physics defined by one or more SDFs used at compile time. This option is recommended for users interested in production-mode and operational applications, since it limits flexibility in favor of runtime performance and memory footprint. Note that the *CCPP-Framework* and *CCPP-physics* are statically linked to the executable.

For all options that activate the CCPP, the `ccpp_prebuild.py` Python script must be run. This may be done manually or as part of a host model build-time script. In the case of the SCM, `ccpp_prebuild.py` must be run manually, as it is not incorporated in that model's build system. In the case of the UFS Atmosphere, `ccpp_prebuild.py` is run automatically as a step in the build system, although it can be run manually for debugging purposes.

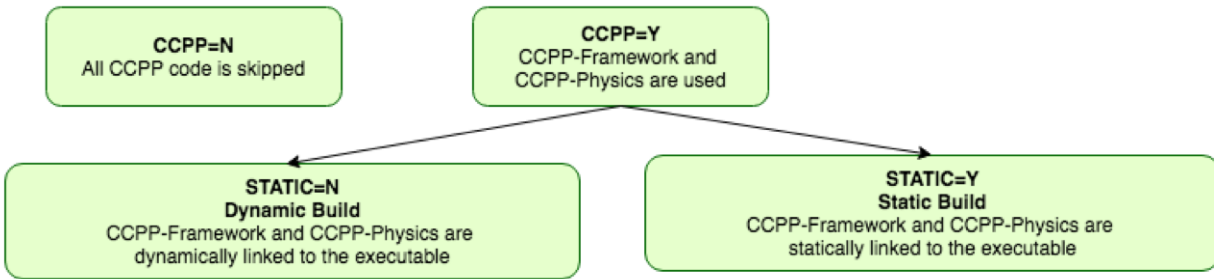


Fig. 3.1: Options for building the CCPP with the UFS Atmosphere. Runs where computational performance is valued over physics flexibility should use *STATIC=Y*.

The path to a host-model specific configuration file is the only required argument to `ccpp_prebuild.py`. 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. Figure 3.2 depicts the main functions of the `ccpp_prebuild.py` script for the dynamic build. Using information included in the configuration file, the script reads metadata associated with all physics variables provided by the host model and metadata associated with all physics variables requested by all physics schemes in the CCPP (at least those whose entry/exit point source files are listed in the configuration file). The script matches the variables and autogenerates software caps for each physics scheme entry point so that they are available to be called by the host model, depending on the contents of a SDF provided at runtime.

In case of a static build, the basic function of the script is the same, but the output (namely, autogenerated software caps) take a different, more computationally-efficient approach. For a general idea of how the static option is different, compare Figure 3.3 for the static case with Figure 3.2 for the dynamic case. As mentioned, during the build step, one or more SDFs are used as an additional input. The script parses the SDF(s) and only matches provided/requested variables that are used within the particular physics suite(s). Rather than autogenerate software caps for each physics scheme (regardless of request), in static mode, the script autogenerates software caps for the physics suite(s) as a whole and for each physics group as defined in the SDF(s). At runtime, a single SDF is used to select the suite that will be executed in the run. This arrangement allows for more 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 compared to the dynamic option and speeds up execution to be on par with physics called using the existing physics-calling code.

27

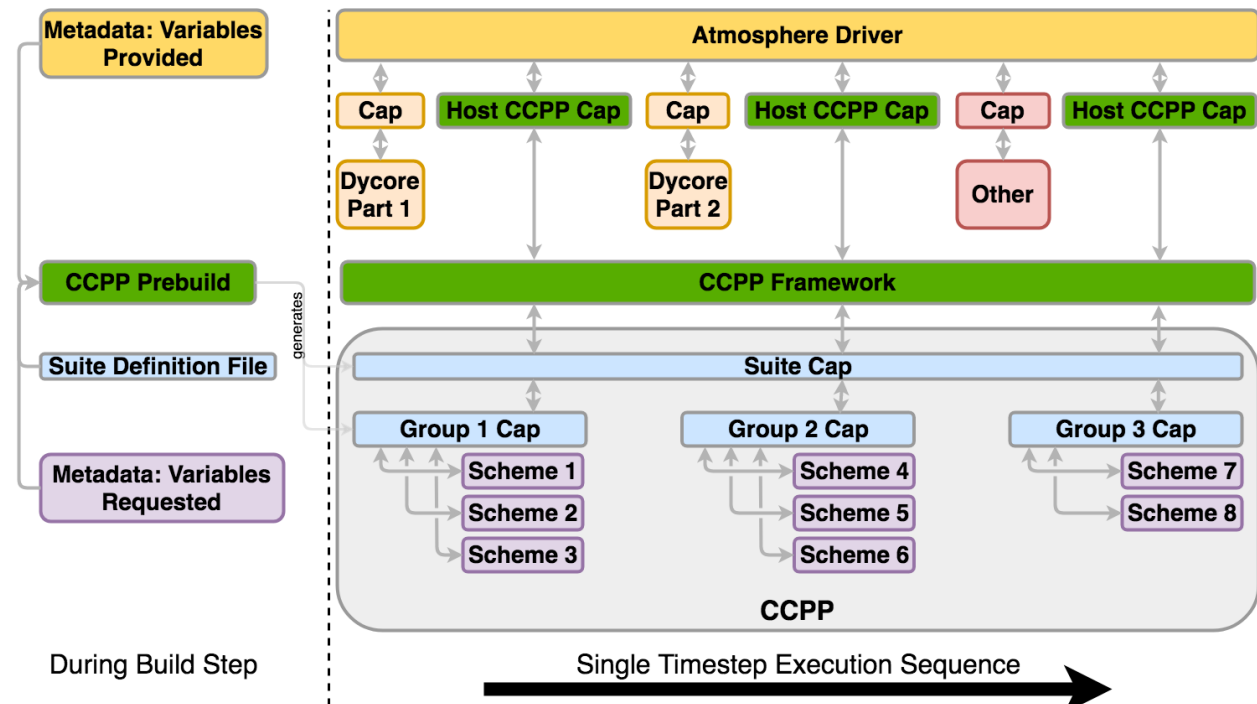


Fig. 3.3: As in Figure 3.2 , but for the **static** build using a single SDF. In this case, software caps are autogenerated for the suite and physics groups (defined in the SDF provided to the `ccpp_prebuild.py` script) rather than for individual schemes. The suite must be defined via the SDF at prebuild 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.

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 schemes 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 name of the suite is listed at the top of the *SDF* 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 `time_vary` step, which is run only once when the model is first initialized.

```
<suite name="ABC" lib="ccppphys" ver="3.0.0">
  <!-- <init></init> -->
  <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>
```

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_run` call in the appropriate place in the host application. The remainder of the parameterization groups could be called using `ccpp_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_run` call. An example of this is found in the `FV3_GFS_v15.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>
  <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.

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. For the static build, reading the *SDF(s)* and defining the order of schemes for each suite happens at compile time. For the dynamic build, no *SDF* is used at compile time (Figure 4.1). Instead, at runtime the *SDF* is read and the order of schemes is determined.

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*.

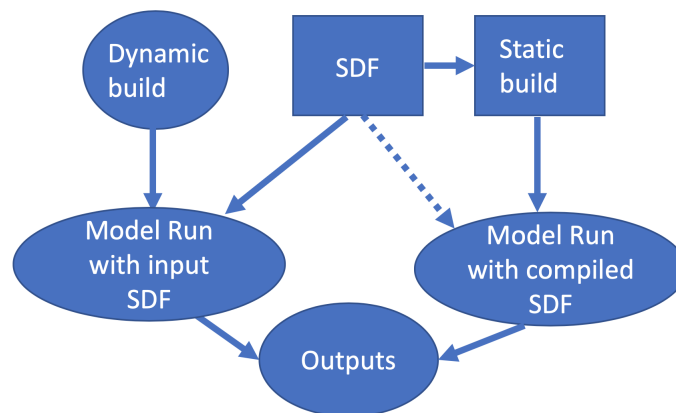


Fig. 4.1: Schematic of the use of the *SDF* in dynamic and static builds. Note that, for the static build, more than one *SDF* can be supplied at compile time, but only one can be used at runtime.

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 *SDF* `suite_Suite_A.xml` could contain the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite_A" lib="ccppphys" ver="3.0.0">
  ...
  <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>
      <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* file. 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" lib="ccppphys" ver="3.0.0">
  <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" lib="ccppphys" ver="3.0.0">
  <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 *UFS* Atmosphere, which employs various groups and subcycling:

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

<suite name="FV3_GFS_v16beta" lib="ccppphys" ver="3">
  <!-- <init></init> -->
  <group name="fast_physics">
    <subcycle loop="1">
      <scheme>fv_sat_adj</scheme>
```

(continues on next page)

(continued from previous page)

```

    </subcycle>
  </group>
  <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">
      <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>dcyc2t3_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>

```

(continues on next page)

(continued from previous page)

```

    <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>GFS_DCNV_generic_pre</scheme>
    <scheme>get_phi_fv3</scheme>
    <scheme>GFS_suite_interstitial_3</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>
  </subcycle>
</group>
<group name="stochastics">
  <subcycle loop="1">
    <scheme>GFS_stochastics</scheme>
  </subcycle>
</group>
<!-- <finalize></finalize> -->
</suite>

```

The suite name is FV3_GFS_v16beta. Five groups (fast_physics, time_vary, radiation, physics, and stochastics) 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.

- fv_sat_adj: Saturation adjustment (for the UFS Atmosphere only)
- 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 surface 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
- `dcyc2t3_post`: Postprocessor for the mapping of the radiative fluxes and heating rates from the coarser radiation timestep onto the model's more frequent time steps
- `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)
- `GFS_GWD_generic_pre`: Preprocessor for the orographic gravity wave drag
- `satmedmfvdifq`: Scale-aware TKE-based moist eddy-diffusion mass-flux
- `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
- `GFS_DCNV_generic_pre`: Preprocessor for the GFS deep convective schemes

- `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
- `GFS_stochastics`: GFS stochastics scheme: Stochastic Kinetic Energy Backscatter (SKEB), Perturbed boundary layer specific humidity (SHUM), or Stochastically Perturbed Physics Tendencies (SPPT)

AUTOGENERATED PHYSICS 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, and a large fraction of code that can be included in the host model *cap*. The host model *cap* connects the framework (Physics Driver) to the host model and must be created manually incorporating the autogenerated code. This chapter describes the physics *caps*, while the host model *caps* are described in [Chapter 6](#). Note that the dynamic build, which can be used with the SCM and with the UFS Atmosphere, produces individual physics scheme *caps*, while the static build (for UFS Atmosphere only) produces group and suite *caps*. The physics *caps* autogenerated by `ccpp_prebuild.py` reside in the directory defined by the `CAPS_DIR` variable (see example in [Listing 8.1](#)).

5.1 Dynamic Build Caps

With the dynamic build, the CCPP-Framework and physics are dynamically linked to the executable to allow maximum runtime flexibility. A *cap* is generated using the metadata associated with the arguments of each scheme and the metadata associated with the variables provided by the host model (see left side of [Figure 3.2](#)). These metadata variables are described in the `.meta` files associated with the host model and the schemes. The CCPP *prebuild* step for the dynamic build performs the following tasks:

- Checks requested vs provided variables by `standard_name`.
- Checks units, rank, type for consistency. Perform unit conversions if a mismatch of units is detected and the required conversion has been implemented (see [Section 5.3](#) for details).
- Creates Fortran code that adds pointers to the host model variables and stores them in the `ccpp-data` structure (`ccpp_fields*.inc`). A hash table that is part of `cdata` is populated with key = `standard_name` of a variable and value = location of that variable in memory (i.e. a c-pointer).
- Creates one *cap* per physics scheme.
- Populates makefiles with schemes and *caps*.

The *prebuild* step will produce the following files for the UFS Atmosphere model:

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

```
ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_FV3.tex
```

- Makefile snippets that contain all *caps* to be compiled:

```
ccpp/physics/CCPP_CAPS.{cmake,mk}
```

- Makefile snippets that contain all schemes (and their dependencies) to be compiled:

```
ccpp/physics/CCPP_SCHEMES.{cmake,mk}
```

- List of variables provided by host model:

```
ccpp/physics/CCPP_VARIABLES_FV3.html
```

- One *cap* per physics scheme:

```
ccpp/physics/*_cap.F90
```

- *.inc files that contain module use and ccpp_field_add statements that populate the ccpp data type (cdata) with the necessary information on where (in memory) to find required variables:

```
FV3/atmos_cubed_sphere/driver/fvGFS/ccpp_modules_{fast,slow}_physics.inc
FV3/atmos_cubed_sphere/driver/fvGFS/ccpp_fields_{fast,slow}_physics.inc
FV3/ipd/ccpp_modules_{fast,slow}_physics.inc
FV3/ipd/ccpp_fields_{fast,slow}_physics.inc
```

The variables added to *_fast_physics.inc do not use GFS_typedefs.F90 or CCPP_data.F90.

- Autogenerated code to include in host model *caps* (called TARGET FILES) via CPP (C preprocessor) directives:

```
FV3/ipd/IPD_CCPP_driver.F90 for slow physics
FV3/atmos_cubed_sphere/driver/fvGFS/atmosphere.F90 for fast physics
```

For each *cap*, ccpp_prebuild.py generates “use” statements based on the host model template. Only the public *caps* (init, run and finalize) are exposed (see code example below). Each *cap* consists of a module containing three functions. For example, scheme_pre_cap.F90 would contain the functions scheme_pre_init_cap, scheme_pre_run_cap and scheme_pre_finalize_cap, which perform the functions below.

- Declare data types cptr, cdims and ckind.
- Create a pointer to the Fortran data type cdata.
- Call ccpp_field_get for each variable in the metadata file for the scheme and pull data from the cdata structure.

The index defined in each call speeds up memory access by avoiding a binary search, since variables are no longer searched by name; the order of the data in cdata are known.

- Call the corresponding scheme entry-point at the end with an explicit argument list.

For example, the autogenerated scheme *cap* for rrtmg_lw_pre_cap.F90 is shown in [Listing 5.1](#).

```
module rrtmg_lw_pre_cap
  use, intrinsic :: iso_c_binding, only: c_f_pointer, &
    c_ptr, c_int32_t
  use :: ccpp_types, only: ccpp_t, CCPP_GENERIC_KIND
  use :: ccpp_fields, only: ccpp_field_get
  use :: ccpp_errors, only: ccpp_error, ccpp_debug
  use :: rrtmg_lw_pre, only: rrtmg_lw_pre_run, &
    rrtmg_lw_pre_init, rrtmg_lw_pre_finalize
  ! Other modules required, e.g. type definitions
  use GFS_typedefs, only: GFS_control_type, GFS_grid_type, &
    GFS_sfcprop_type, GFS_radtend_type
  use machine, only: kind_phys
  implicit none
  private
  public :: rrtmg_lw_pre_run_cap, rrtmg_lw_pre_init_cap, &
```

(continues on next page)

(continued from previous page)

```

        rrtmg_lw_pre_finalize_cap
contains
function rrtmg_lw_pre_init_cap(ptr) bind(c) result(ierr)
    integer(c_int32_t)      :: ierr
    type(c_ptr), intent(inout) :: ptr
    type(ccpp_t), pointer    :: cdata
    type(c_ptr)             :: cptr
    integer, allocatable     :: cdims(:)
    integer                 :: ckind
    ierr = 0
    call c_f_pointer(ptr, cdata)
    call rrtmg_lw_pre_init()
end function rrtmg_lw_pre_init_cap

function rrtmg_lw_pre_run_cap(ptr) bind(c) result(ierr)
    integer(c_int32_t)      :: ierr
    type(c_ptr), intent(inout) :: ptr
    type(ccpp_t), pointer    :: cdata
    type(c_ptr)             :: cptr
    integer, allocatable     :: cdims(:)
    integer                 :: ckind
    type(GFS_control_type), pointer    :: Model
    type(GFS_grid_type), pointer       :: Grid
    type(GFS_sfcprop_type), pointer    :: Sfcprop
    type(GFS_radtend_type), pointer    :: Radtend
    integer, pointer :: im
    real(kind_phys), pointer :: tsfg(:)
    real(kind_phys), pointer :: tsfa(:)
    ierr = 0
    call c_f_pointer(ptr, cdata)
    call ccpp_field_get(cdata, 'GFS_control_type_instance', cptr, &
        ierr=ierr, kind=ckind, index=2)
    call c_f_pointer(cptr, Model)
    call ccpp_field_get(cdata, 'GFS_grid_type_instance', cptr, &
        ierr=ierr, kind=ckind, index=6)
    call c_f_pointer(cptr, Grid)
    call ccpp_field_get(cdata, 'GFS_sfcprop_type_instance', &
        cptr, ierr=ierr, kind=ckind, index=10)
    call c_f_pointer(cptr, Sfcprop)
    call ccpp_field_get(cdata, 'GFS_radtend_type_instance', &
        cptr, ierr=ierr, kind=ckind, index=9)
    call c_f_pointer(cptr, Radtend)
    call ccpp_field_get(cdata, 'horizontal_loop_extent', im, &
        ierr=ierr, kind=ckind, index=390)
    call ccpp_field_get(cdata, &
        'surface_ground_temperature_for_radiation', &
        tsfg, ierr=ierr, dims=cdims, kind=ckind, index=770)
    deallocate(cdims)
    call ccpp_field_get(cdata, &
        'surface_air_temperature_for_radiation', &
        tsfa, ierr=ierr, dims=cdims, kind=ckind, index=724)
    deallocate(cdims)
    call rrtmg_lw_pre_run(Model=Model, Grid=Grid, &
        Sfcprop=Sfcprop, Radtend=Radtend, im=im, &
        tsfg=tsfg, tsfa=tsfa, &
        errmsg=cdata%errmsg, errflg=cdata%errflg)
    ierr=cdata%errflg

```

(continues on next page)

(continued from previous page)

```

end function rrtmg_lw_pre_run_cap
function rrtmg_lw_pre_finalize_cap(ptr) bind(c) result(ierr)
  integer(c_int32_t)      :: ierr
  type(c_ptr), intent(inout) :: ptr
  type(ccpp_t), pointer    :: cdata
  type(c_ptr)             :: cptr
  integer, allocatable     :: cdims(:)
  integer                 :: ckind
  ierr = 0
  call c_f_pointer(ptr, cdata)
  call rrtmg_lw_pre_finalize()
end function rrtmg_lw_pre_finalize_cap
end module rrtmg_lw_pre_cap

```

Listing 5.1: Condensed version of the autogenerated scheme cap rrtmg_lw_pre_cap.F90 for the dynamic build. Note the calls to ccpp_field_get for each variable.

The fields accessed from `cdata` are determined by the metadata for the arguments of the scheme. In this example, `rrtmg_lw_pre_init` and `rrtmg_lw_pre_finalize` are empty subroutines, i.e. they have no arguments passed in or out, no metadata section in the `.meta` file, and no calls to `ccpp_field_get`. However, `rrtmg_lw_pre_run` has a metadata section in file `rrtmg_lw_pre.meta`, so `ccpp_field_get` is called for each variable described in the metadata section and the value put into the call to `rrtmg_lw_pre_run`.

5.2 Static Build Caps

With a static build, the CCpp-Framework and physics are statically linked to the executable. This allows the best performance and efficient memory use. Similar to the dynamic build, the static build requires metadata provided by the host model and variables requested from the physics scheme. Unlike a dynamic build where all variables are kept and pulled multiple times for various parameterizations, a static build only keeps variables for specified suites, and therefore requires one or more SDFs (see left side of [Figure 3.3](#)) as arguments to the `ccpp_prebuild.py` script. The CCpp *prebuild* step for the static build 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 5.3](#) for details).
- Filter unused schemes and variables.
- Create Fortran code for the static Application Programming Interface (API) that replaces the dynamic API (CCPP-Framework). The hash table used by the dynamic build to store variables in memory is left empty.
- Create *caps* for groups and suite(s).
- Populate makefiles with schemes and *caps*.

The *prebuild* step for the static build will produce the following files for the UFS Atmosphere:

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

```
ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_FV3.tex
```

- Makefile snippets that contain all *caps* to be compiled:

```
ccpp/physics/CCPP_CAPS.{cmake,mk}
```

- Makefile snippets that contain all schemes to be compiled:

```
ccpp/physics/CCPP_SCHEMES.{cmake,mk}
```

- List of variables provided by host model:

```
ccpp/physics/CCPP_VARIABLES_FV3.html
```

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

```
ccpp/physics/ccpp_{suite_name}_{group_name}_cap.F90
```

- *Cap* for each suite:

```
ccpp/physics/ccpp_{suite_name}_cap.F90
```

- Autogenerated API for static build that replaces the dynamic API (aka CCpp-Framework), the interface is identical between the two APIs:

```
FV3/gfsphysics/CCPP_layer/ccpp_static_api.F90
```

- Same TARGET FILES as for the dynamic build

ccpp_static_api.F90 replaces the entire dynamic CCpp-Framework with an equivalent interface, which contains subroutines ccpp_physics_init, ccpp_physics_run 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, run, finalize}. The subroutine ccpp_physics_init from the autogenerated code using suites FV3_GFS_v15 and FV3_CPT_v0 is shown in [Listing 5.2](#).

```
subroutine ccpp_physics_init(cdata, suite_name, group_name, ierr)
  use ccpp_types, only : ccpp_t
  implicit none
  type(ccpp_t),          intent(inout) :: cdata
  character(len=*),      intent(in)   :: suite_name
  character(len=*), optional, intent(in) :: group_name
  integer,               intent(out)  :: ierr
  ierr = 0
  if (trim(suite_name)=="FV3_GFS_v15") then
    if (present(group_name)) then
      if (trim(group_name)=="fast_physics") then
        ierr = FV3_GFS_v15_fast_physics_init_cap(cdata=cdata, CCPP_interstitial=CCPP_
        →interstitial)
      else if (trim(group_name)=="time_vary") then
        ierr = FV3_GFS_v15_time_vary_init_cap(GFS_Interstitial=GFS_Interstitial, &
        cdata=cdata,GFS_Data=GFS_Data, GFS_Control=GFS_Control)
      else if (trim(group_name)=="radiation") then
        ierr = FV3_GFS_v15_radiation_init_cap()
      else if (trim(group_name)=="physics") then
        ierr = FV3_GFS_v15_physics_init_cap(cdata=cdata, GFS_Control=GFS_Control)
      else if (trim(group_name)=="stochastics") then
        ierr = FV3_GFS_v15_stochastics_init_cap()
      else
        write(cdata%errmsg, '(*(a))') "Group " // trim(group_name) // " not found"
        ierr = 1
      end if
    else
      ierr = FV3_GFS_v15_init_cap(GFS_Interstitial=GFS_Interstitial, cdata=cdata,GFS_
      →Control=GFS_Control, &
```

(continues on next page)

(continued from previous page)

```

        GFS_Data=GFS_Data, CCpp_interstitial=CCpp_interstitial)
    end if
else if (trim(suite_name)=="FV3_CPT_v0") then
    if (present(group_name)) then
        if (trim(group_name)=="time_vary") then
            ierr = FV3_CPT_v0_time_vary_init_cap(GFS_Interstitial=GFS_Interstitial, &
                cdata=cdata,GFS_Data=GFS_Data, GFS_Control=GFS_Control)
        else if (trim(group_name)=="radiation") then
            ierr = FV3_CPT_v0_radiation_init_cap()
        else if (trim(group_name)=="physics") then
            ierr = FV3_CPT_v0_physics_init_cap(con_hfus=con_hfus, &
                GFS_Control=GFS_Control,con_hvap=con_hvap, &
                con_rd=con_rd,con_rv=con_rv,con_g=con_g, &
                con_ttp=con_ttp,con_cp=con_cp,cdata=cdata)
        else if (trim(group_name)=="stochastics") then
            ierr = FV3_CPT_v0_stochastics_init_cap()
        else
            write(cdata%errmsg, '(*a)') "Group " // trim(group_name) // " not found"
            ierr = 1
        end if
    else
        ierr = FV3_CPT_v0_init_cap(con_g=con_g, GFS_Data=GFS_Data,GFS_Control=GFS_
        ↪Control, &
            con_hvap=con_hvap,GFS_Interstitial=GFS_Interstitial, con_rd=con_rd,con_
        ↪rv=con_rv, &
            con_hfus=con_hfus, con_ttp=con_ttp,con_cp=con_cp,cdata=cdata)
    end if
else
    write(cdata%errmsg, '(*a)'), 'Invalid suite ' // trim(suite_name)
    ierr = 1
end if
cdata%errflg = ierr
end subroutine ccpp_physics_init

```

Listing 5.2: Code sample of subroutine `ccpp_physics_init` contained in the autogenerated file `ccpp_static_api.F90` for the multi-suite static build. This cap was generated using suites `FV3_GFS_v15` and `FV3_CPT_v0`. Examples of the highlighted functions are shown below in [Listing 5.3](#) and [Listing 5.4](#).

Note that if `group_name` is set, specified groups (i.e. `FV3_GFS_v15_physics_init_cap`) are called for the specified `suite_name`. These functions are defined in `ccpp_{suite_name}_{group_name}_cap.F90`, in this case `ccpp_FV3_GFS_v15_physics_cap.F90`. For example:

```

function FV3_GFS_v15_physics_init_cap(cdata,GFS_Control)&
    result(ierr)
    use ccpp_types, only: ccpp_t
    use GFS_typedefs, only: GFS_control_type
    implicit none
    integer :: ierr
    type(ccpp_t), intent(inout) :: cdata
    type(GFS_control_type), intent(in) :: GFS_Control
    ierr = 0
    if (initialized) return
    call lsm_noah_init(me=GFS_Control%me,isot=GFS_Control%isot,&
        ivegsr=vegsr=vegsr, nlunit=GFS_Control%nlunit, &
        errmsg=cdata%errmsg,errflg=cdata%errflg)
    if (cdata%errflg/=0) then
        write(cdata%errmsg, '(a)') "An error occurred in lsm_noah_init"
    end if
end function

```

(continues on next page)

(continued from previous page)

```

        ierr=cdata%errflg
        return
    end if
    call gfdl_cloud_microphys_init(me=GFS_Control%me, &
        master=GFS_Control%master,nlunit=GFS_Control%nlunit, &
        input_nml_file=GFS_Control%input_nml_file, &
        logunit=GFS_Control%logunit,fn_nml=GFS_Control%fn_nml, &
        imp_physics=GFS_Control%imp_physics, &
        imp_physics_gfdl=GFS_Control%imp_physics_gfdl, &
        do_shoc=GFS_Control%do_shoc, &
        errmsg=cdata%errmsg,errflg=cdata%errflg)
    if (cdata%errflg/=0) then
        write(cdata%errmsg,'(a)') "An error occured in &
            gfdl_cloud_microphys_init"
        ierr=cdata%errflg
        return
    end if
    initialized = .true.
end function FV3_GFS_v15_physics_init_cap

```

Listing 5.3: The FV3_GFS_v15_physics_init_cap contained in the autogenerated file ccpp_FV3_GFS_v15_physics_cap.F90 showing calls to the lsm_noah_init , and gfdl_cloud_microphys_init subroutines for the static build for suite 'FV3_GFS_v15' and group 'physics'.

If the group_name is not specified for a specified suite_name, the suite is called from the autogenerated ccpp_static_api.F90, which calls the init, run and finalize routines for each group. [Listing 5.4](#) is an example of FV3_GFS_v15_init_cap.

```

function FV3_GFS_v15_init_cap(GFS_Interstitial, &
    cdata,GFS_Control,GFS_Data,CCPP_interstitial) result(ierr)
    use GFS_typedefs, only: GFS_interstitial_type
    use ccpp_types, only: ccpp_t
    use GFS_typedefs, only: GFS_control_type
    use GFS_typedefs, only: GFS_data_type
    use CCPP_typedefs, only: CCPP_interstitial_type

    implicit none

    integer :: ierr
    type(GFS_interstitial_type), intent(inout) :: GFS_Interstitial(:)
    type(ccpp_t), intent(inout) :: cdata
    type(GFS_control_type), intent(inout) :: GFS_Control
    type(GFS_data_type), intent(inout) :: GFS_Data(:)
    type(CCPP_interstitial_type), intent(in) :: CCPP_interstitial

    ierr = 0
    ierr = FV3_GFS_v15_fast_physics_init_cap(cdata=cdata, CCPP_interstitial=CCPP_
    ↪interstitial)
    if (ierr/=0) return

    ierr = FV3_GFS_v15_time_vary_init_cap (GFS_Interstitial=GFS_Interstitial,
    ↪cdata=cdata, &
        GFS_Data=GFS_Data,GFS_Control=GFS_Control)
    if (ierr/=0) return

    ierr = FV3_GFS_v15_radiation_init_cap()
    if (ierr/=0) return

```

(continues on next page)

(continued from previous page)

```

ierr = FV3_GFS_v15_physics_init_cap(cdata=cdata, &
    GFS_Control=GFS_Control)
if (ierr/=0) return

ierr = FV3_GFS_v15_stochastics_init_cap()
if (ierr/=0) return
end function FV3_GFS_v15_init_cap

```

Listing 5.4: Condensed version of the FV3_GFS_v15_init_cap function contained in the autogenerated file ccpp_FV3_GFS_v15_cap.F90 showing calls to the group caps FV3_GFS_v15_fast_physics_init_cap, FV3_GFS_v15_time_vary_init_cap, etc. for the static build where a group name is not specified.

5.3 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. 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 static build.

```

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

! varl 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)

```

(continues on next page)

(continued from previous page)

```

! 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.

HOST SIDE CODING

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

In several places, references are made to an Interoperable Physics Driver (IPD). The IPD was originally developed by EMC and later expanded by NOAA GFDL with the goal of connecting GFS physics to various models. A top motivation for its development was the dycore test that led to the selection of FV3 as the dycore for the *UFS*. Designed in a fundamentally different way than the *CCPP*, the IPD will be phased out in the near future in favor of the CCPP as a single way to interface with physics in the UFS. To enable a smooth transition, several of the CCPP components must interact with the IPD and, as such, parts of the CCPP code in the UFS currently carry the tag “IPD”.

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`, `blk_no`, and `thrd_no`, which are allocated by the CCPP-Framework, as explained in [Section 6.4.1](#). A list of all variables required for the current pool of physics can be found in `ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_XYZ.pdf` (XYZ: SCM, FV3).

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 2.2](#)), 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 Variable 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`) with multiple metadata sections in each file (`[ccpp-arg-table]`) or just one. 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 attributes `optional` and `intent` must be set to `F` and `none`, respectively.
- The `local_name` of the variable must be set to the name the host model cap 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_reall
  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-arg-table]
  name = arg_table_example_vardefs
  type = module
[ex_int]
  standard_name = example_int
  long_name = ex. int
  units = none
  dimensions = ()
  type = integer
  kind =
[ex_real]
  standard_name = example_real
  long_name = ex. real
  units = m
  dimensions = (horizontal_dimension,vertical_dimension)
  type = real
  kind = kind=8
[ex_ddt]
  standard_name = example_ddt
  long_name = ex. ddt
  units = DDT
  dimensions = (horizontal_dimension,vertical_dimension)
  type = ex_ddt
  kind =
[ext]
  standard_name = example_ddt_instance
  long_name = ex. ddt inst
  units = DDT
  dimensions = (horizontal_dimension,vertical_dimension)
  type = ex_ddt
  kind =
[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_flag
  long_name = error flag for error handling in CCPP
  units = flag
  dimensions = ()
  type = integer

#####
[ccpp-arg-table]
  name = arg_table_example_ddt
  type = ddt
[ext%l]
  standard_name = example_flag
  long_name = ex. flag
  units = flag
  dimensions =
  type = logical
  kind =
[ext%r]

```

(continues on next page)

(continued from previous page)

```

standard_name = example_real3
long_name = ex. real
units = kg
dimensions = (horizontal_dimension, vertical_dimension)
type = real
kind = r15
[ext%r(,1)]
standard_name = example_slice
long_name = ex. slice
units = kg
dimensions = (horizontal_dimension, vertical_dimension)
type = real
kind = r15

```

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

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, DDTs are defined in `gmtb_scm_type_defs.f90` and `GFS_typedefs.F90`, and in the case of the UFS Atmosphere, they 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_interstitial_type` fields used to communicate variables among schemes in the slow physics group required to replace interstitial code in `GFS_{physics, radiation}_driver.F90` in CCPP
- `GFS_data_type` combined type of all of the above except `GFS_control_type` and `GFS_interstitial_type`
- `CCPP_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-arg-table]
    name = GFS_stateout_type
    type = ddt
[gq0(:, :, index_for_snow_water)]
    standard_name = snow_water_mixing_ratio_updated_by_physics
    long_name = moist (dry+vapor, no condensates) mixing ratio of snow water updated by_
    ↪ physics
    units = kg kg-1
    dimensions = (horizontal_dimension, vertical_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 bulk of the CCpp API is located in the CCpp-Framework, and is described in file `ccpp_api.F90`. Some aspects of the API differ between the dynamic and static build. In particular, subroutines `ccpp_physics_init`, `ccpp_physics_finalize`, and `ccpp_physics_run` (described below) are made public from `ccpp_api.F90` for the dynamic build, and are contained in `ccpp_static_api.F90` for the static build. Moreover, these subroutines take an additional argument (`suite_name`) for the static build. File `ccpp_static_api.F90` is auto-generated when the script `ccpp_prebuild.py` is run for the static build.

6.4.1 Data Structure to Transfer Variables between Dynamics and Physics

The roles of `cdata` structure in dealing with data exchange are not the same between the dynamic and the static builds of the CCpp. For the dynamic build, the `cdata` structure handles the data exchange between the host model and the physics schemes. `cdata` is a DDT containing a list of pointers to variables and their metadata and is persistent in memory.

For both the dynamic and static builds, the `cdata` structure is used for holding five 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 6.3](#)).

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

```
[ccpp-arg-table]
  name = ccpp_t
  type = scheme
[errflg]
  standard_name = ccpp_error_flag
  long_name = error flag for error handling in CCpp
  units = flag
  dimensions = ()
  type = integer
[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
[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 6.3: 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`, `blk_no`, and `thrd_no` can be passed to the schemes if required, but are not mandatory. For the static build of the CCPP, the `cdata` structure is only used to hold these five 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 may require `cdata` to be a one-dimensional array of the same size. Another example of a multi-dimensional array of `cdata` is in the SCM, which uses a one-dimensional `cdata` array for N independent columns.

Due to a restriction in the Fortran language, there are no standard pointers that are generic pointers, such as the C language allows. The CCPP system therefore has an underlying set of pointers in the C language that are used to point to the original data within the host application cap. The user does not see this C data structure, but deals only with the public face of the Fortran `cdata` DDT. The type `ccpp_t` is defined in `ccpp/framework/src/ccpp_types.meta` and declared in `ccpp/framework/src/ccpp_types.F90`.

6.4.2 Adding and Retrieving Information from `cdata` (dynamic build option)

Subroutines `ccpp_field_add` and `ccpp_field_get` are part of the CCPP-Framework and are used (in the dynamic build only) to load and retrieve information to and from `cdata`. The calls to `ccpp_field_add` are auto-generated by the script `ccpp_prebuild.py` and inserted onto the host model code via include files (i.e. `FV3/CCPP_layer/ccpp_fields_slow_physics.inc`) before it is compiled.

A typical call to `ccpp_field_add` is below, where the first argument is the instance of `cdata` to which the information should be added, the second argument is the `standard_name` of the variable, the third argument is the corresponding host model variable, the fourth argument is an error flag, the fifth argument is the units of the variable, and the last (optional) argument is the position within `cdata` in which the variable is expected to be stored.

```
call ccpp_field_add(cdata, 'y_wind_updated_by_physics', GFS_Data(cdata%blk_no)
↳ %Stateout%gv0, ierr=ierr, units='m s-1', index=886)
```

For DDTs, the interface to `CCPP_field_add` is slightly different:

```
call ccpp_field_add(cdata, 'GFS_cldprop_type_instance', '', c_loc(GFS_Data(cdata%blk_
↳ no)%Cldprop), ierr=ierr, index=1)
```

where the first argument and second arguments bear the same meaning as in the first example, the third argument is the units (can be left empty or set to “DDT”), the fourth argument is the C pointer to the variable in memory, the fifth argument is an error flag, and the last (optional) argument is the position within `cdata` as in the first example.

Each new variable added to `cdata` is always placed at the next free position, and a check is performed to confirm that this position corresponds to the expected one, which in this example is 886. A mismatch will occur if a developer manually adds a call to `ccpp_field_add`, in which case a costly binary search is applied every time a variable is retrieved from memory. Adding calls manually is not recommended as all calls to `ccpp_fields_add` should be auto-generated.

The individual physics *caps* used in the dynamic build, which are auto-generated using the script `ccpp_prebuild.py`, contain calls to `ccpp_field_get` to pull data from the `cdata` DDT as a Fortran pointer to a variable that will be passed to the individual physics scheme.

6.4.3 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 Subroutine

The suite initialization subroutine, `ccpp_init`, takes three mandatory and two optional arguments. The mandatory arguments are the name of the suite (of type character), the name of the `cdata` variable that must be allocated at this point, and an integer used for the error status. Note that the suite initialization routine `ccpp_init` parses the SDF corresponding to the given suite name and initializes the state of the suite and its schemes. This process must be repeated for every element of a multi-dimensional `cdata`. For performance reasons, it is possible to avoid repeated reads of the SDF and to have a single state of the suite shared between the elements of `cdata`. To do so, specify an optional argument variable called `cdata_target = X` in the call to `ccpp_init`, where `X` refers to the instance of `cdata` that has already been initialized.

For a given suite name `XYZ`, the name of the suite definition file is inferred as `suite_XYZ.xml`, and the file is expected to be present in the current run directory. It is possible to specify the optional argument `is_filename=.true.` to `ccpp_init`, which will treat the suite name as an actual file name (with or without the path to it).

Typical calls to `ccpp_init` are below, where `ccpp_suite` is the name of the suite, and `ccpp_sdf_filepath` the actual SDF filename, with or without a path to it.

```
call ccpp_init(trim(ccpp_suite), cdata, ierr)
call ccpp_init(trim(ccpp_suite), cdata2, ierr, [cdata_target=cdata])
call ccpp_init(trim(ccpp_sdf_filepath), cdata, ierr, [is_filename=.true.] )
```

Suite Finalization Subroutine

The suite finalization subroutine, `ccpp_finalize`, takes two arguments, the name of the `cdata` variable that must be de-allocated at this point, and an integer used for the error status. A typical call to `ccpp_finalize` is below:

```
call ccpp_finalize(cdata, ierr)
```

If a specific data instance was used in a call to `ccpp_init`, as in the above example in [Section 6.4.3](#), then this data instance must be finalized last:

```
call ccpp_finalize(cdata2, ierr)
call ccpp_finalize(cdata, ierr)
```

6.4.4 Running the physics

The physics is invoked by calling subroutine `ccpp_physics_run`. This subroutine is part of the CCPP API and is included with the CCPP-Framework (for the dynamic build) or auto-generated (for the static build). This subroutine is capable of executing the physics with varying granularity, that is, a single scheme (dynamic build only), a single group, or an entire suite can be run with a single subroutine call. Typical calls to `ccpp_physics_run` are below, where `scheme_name` and `group_name` are optional and mutually exclusive (dynamic build), and where `suite_name` is mandatory and `group_name` is optional (static build).

Dynamic build:

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

Static build:

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

6.4.5 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 typically done for the entire domain, that is, they are not subdivided by blocks. 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.

The initialization and finalization can be invoked for a single parameterization (only in dynamic build), for a single group, or for the entire suite. In all cases, subroutines `ccpp_physics_init` and `ccpp_physics_finalize` are used and the arguments passed to those subroutines determine the type of initialization.

These subroutines should not be confused with `ccpp_init` and `ccpp_finalize`, which were explained previously.

Subroutine `ccpp_physics_init`

This subroutine is part of the CCpp API and is included with the CCpp-Framework (for the dynamic build) or auto-generated (for the static build). It cannot contain thread-dependent information but can have block-dependent information. Typical calls to `ccpp_physics_init` are below.

Dynamic build:

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

Static build:

```
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 included with the CCpp-Framework (for the dynamic build) or auto-generated (for the static build). Typical calls to `ccpp_physics_finalize` are below.

Dynamic build:

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

Static build:

```
call ccpp_physics_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 the *cdata* structure(s)
 - For the dynamic build, the *cdata* structure handles the data exchange between the host model and the physics schemes, while for the static build, *cdata* is utilized in a reduced capacity.
- Calling the suite initialization subroutine
 - The suite must be initialized using *ccpp_init*.
- Populating the *cdata* structure(s)
 - For the dynamic build, each variable required by the physics schemes must be added to the *cdata* structure (or to each element of a multi-dimensional *cdata*) on the host model side using subroutine *ccpp_field_add*. This is an automated task accomplished by inserting a preprocessor directive

```
#include ccpp_modules.inc
```

at the top of the *cap* (before implicit none) to load the required modules and a second preprocessor directive

```
#include ccpp_fields.inc
```

after the *cdata* variable and the variables required by the physics schemes are allocated and after the call to *ccpp_init* for this *cdata* variable. For the static build, this step can be skipped because the autogenerated *caps* for the physics (groups and suite *caps*) are automatically given memory access to the host model variables and they can be used directly, without the need for a data structure containing pointers to the actual variables (which is what *cdata* is).

Note: The CCpp-Framework supports splitting physics schemes into different sets that are used in different parts of the host model. An example is the separation between slow and fast physics processes for the GFDL microphysics implemented in the UFS Atmosphere: while the slow physics are called as part of the usual model physics, the fast physics are integrated in the dynamical core. The separation of physics into different sets is determined in the CCpp *prebuild* configuration for each host model (see [Chapter 5.1](#), and [Figure 8.1](#)), which allows to create multiple include files (e.g. *ccpp_fields_slow_physics.inc* and *ccpp_fields_fast_physics.inc* that can be used by different *cdata* structures in different parts of the model). This is a highly advanced feature and developers seeking to take further advantage of it should consult with GMTB first.

- 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_init*, *ccpp_physics_init*,

ccpp_physics_run, ccpp_physics_finalize and ccpp_finalize, and handle any errors returned See [Listing 6.4](#).

```

module example_ccpp_host_cap

use ccpp_api,           only: ccpp_t, ccpp_init, ccpp_finalize
use ccpp_static_api,    only: ccpp_physics_init, ccpp_physics_run,      &
                                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 the CCPP framework, parse SDF
        call ccpp_init(trim(ccpp_suite_name), cdata, ierr=ierr)
        if (ierr/=0) then
            write(*, '(a)') "An error occurred in ccpp_init"
            stop
        end if

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

    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_run(cdata, suite_name=trim(ccpp_suite_name),      &
                                    group_name=group, ierr=ierr)
        else
            call ccpp_physics_run(cdata, suite_name=trim(ccpp_suite_name),      &
                                    ierr=ierr)
        end if
        ! error handling as above

    end subroutine physics_run

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

```

(continues on next page)

(continued from previous page)

```

! Finalize CCpp physics (run all _finalize routines)
call ccpp_physics_finalize(cdata, suite_name=trim(ccpp_suite_name), &
                           ierr=ierr)
! error handling as above
call ccpp_finalize(cdata, ierr=ierr)
! error handling as above

end subroutine physics_finalize
end module example_ccpp_host_cap

```

Listing 6.4: Fortran template for a CCpp host model cap from `ccpp/framework/doc/DevelopersGuide/host_cap_template.F90`.

The following sections describe two implementations of host model caps to serve as examples. For each of the functions listed above, a description for how it is implemented in each host model is included.

6.5.1 SCM Host Cap

The only build type supported for the SCM v4 is the static build. The cap functions are mainly implemented in:

`gmtb-scm/scm/src/gmtb_scm.F90`

With smaller parts in:

`gmtb-scm/scm/src/gmtb_scm_type_defs.f90`

`gmtb-scm/scm/src/gmtb_scm_setup.f90`

`gmtb-scm/scm/src/gmtb_scm_time_integration.f90`

The host model *cap* is responsible for:

- Allocating memory for variables needed by physics

All variables and constants required by the physics have metadata provided on the host-side, `arg_table_physics_type` and `arg_table_gmtb_scm_physical_constants`, which are implemented in `gmtb_scm_type_defs.f90` and `gmtb_scm_physical_constants.f90`. To mimic the UFS Atmosphere and to hopefully reduce code maintenance, currently, the SCM uses GFS DDTs as sub-types within the physics DDT.

In `gmtb_scm_type_defs.f90`, the physics DDT has a create type-bound procedure (see subroutine `physics_create` and type `physics_type`), which allocates GFS sub-DDTs and other physics variables and initializes them with zeros. `physics%create` is called from `gmtb_scm.F90` after the initial SCM state has been set up.

- Allocating the `cdata` structure

The SCM uses a one-dimensional `cdata` array for `N` independent columns, i.e. in `gmtb_scm.F90`:

```
allocate(cdata(scm_state%n_cols))
```

- Calling the suite initialization subroutine

Within `scm_state%n_cols` loop in `gmtb_scm.F90` after initial SCM state setup and before first timestep, the suite initialization subroutine `ccpp_init` is called for each column with own instance of `cdata`, and takes three arguments, the name of the runtime SDF, the name of the `cdata` variable that must be allocated at this point, and `ierr`.

- Populating the cdata structure

Within the same `scm_state%n_cols` loop, but after the `ccpp_init` call, the `cdata` structure is filled in with real initialized values:

- `physics%Init_parm` (GFS DDT for setting up suite) are filled in from `scm_state%`
- call `GFS_suite_setup()`: similar to `GFS_initialize()` in the UFS Atmosphere, is called and includes:
- `%init/%create` calls for GFS DDTs
- initialization for other variables in physics DDT
- `init` calls for legacy non-ccpp schemes
- call `physics%associate()`: to associate pointers in physics DDT with targets in `scm_state`, which contains variables that are modified by the SCM “dycore” (i.e. forcing).
- Actual `cdata` fill in through `ccpp_field_add` calls:

```
#include "ccpp_fields.inc"
```

This include file is auto-generated from `ccpp/scripts/ccpp_prebuild.py`, which parses tables in `gmtb_scm_type_defs.f90`.

- Providing interfaces to call the CCpp

- Calling `ccpp_physics_init()`

Within the same `scm_state%n_cols` loop but after `cdata` is filled, the physics initialization routines (`*_init()`) associated with the physics suite, group, and/or schemes are called at each column.

- Calling `ccpp_physics_run()`

At the first timestep, if the forward scheme is selected (i.e. `scm_state%time_scheme == 1`), call `do_time_step()` to apply forcing and `ccpp_physics_run()` calls at each column; if the leapfrog scheme is selected (i.e. `scm_state%time_scheme == 2`), call `ccpp_physics_run()` directly at each column.

At a later time integration, call `do_time_step()` to apply forcing and `ccpp_physics_run()` calls at each column. Since there is no need to execute anything between physics groups in the SCM, the `ccpp_physics_run` call is only given `cdata` and an error flag as arguments.

- Calling `ccpp_physics_finalize()` and `ccpp_finalize()`

`ccpp_physics_finalize()` and `ccpp_finalize()` are called after the time loop at each column.

6.5.2 UFS Atmosphere Host Cap

For the UFS Atmosphere, there are slightly different versions of the host cap implementation depending on the desired build type (dynamic or static). Only the static build is publicly supported at this time, as the dynamic build will be deprecated soon. As discussed in [Chapter 8](#), these modes are controlled via appropriate strings included in the `MAKEOPTS` build-time argument. Within the source code, the three modes are executed within appropriate preprocessor directive blocks:

For any build that uses CCpp (dynamic or static):

```
#ifdef CCpp
#endif
```

For static (often nested within `#ifdef CCpp`):

```
#ifdef STATIC
#endif
```

The following text describes how the host cap functions listed above are implemented for the dynamic build only. Where the other modes of operation differ in their implementation, it will be called out.

- Allocating memory for variables needed by physics
 - Within the `atmos_model_init` subroutine of `atmos_model.F90`, the following statement is executed

```
allocate(IPD_Data)
```

`IPD_Data` is of `IPD_data_type`, which is defined in `IPD_typedefs.F90` as a synonym for `GFS_data_type` defined in `GFS_typedefs.F90`. This data type contains GFS-related DDTs (`GFS_statein_type`, `GFS_stateout_type`, `GFS_sfcprop_type`, etc.) as sub-types, which are defined in `GFS_typedefs.F90`.
- Allocating the `cdata` structures
- For the current implementation of the UFS Atmosphere, which uses a subset of fast physics processes tightly coupled to the dynamical core, three instances of `cdata` exist within the host model: `cdata_tile` to hold data for the fast physics, `cdata_domain` to hold data needed for all UFS Atmosphere blocks for the slow physics, and `cdata_block`, an array of `cdata` DDTs with dimensions of (number of blocks, number of threads) to contain data for individual block/thread combinations for the slow physics. All are defined as module-level variables in the `CCPP_data` module of `CCPP_data.F90`. The `cdata_block` array is allocated (since the number of blocks and threads is unknown at compile-time) as part of the 'init' step of the `CCPP_step` subroutine in `CCPP_driver.F90`. Note: Although the `cdata` containers are not used to hold the pointers to the physics variables for the static mode, they are still used to hold other CCpp-related information for that mode.
- Calling the suite initialization subroutine
 - Corresponding to the three instances of `cdata` described above, the `ccpp_init` subroutine is called within three different contexts, all originating from the `atmos_model_init` subroutine of `atmos_model.F90`:
 - For `cdata_tile` (used for the fast physics), the `ccpp_init` call is made from the `atmosphere_init` subroutine of `atmosphere.F90`. Note: when fast physics is used, this is the *first* call to `ccpp_init`, so it reads in the SDF and initializes the suite in addition to setting up the fields for `cdata_tile`.
 - For `cdata_domain` and `cdata_block` used in the rest of the physics, the 'init' step of the `CCPP_step` subroutine in `CCPP_driver.F90` is called. Within that subroutine, `ccpp_init` is called once to set up `cdata_domain` and within a loop for every block/thread combination to set up the components of the `cdata_block` array. Note: as mentioned in the CCpp API [Section 6.4](#), when fast physics is used, the SDF has already been read and the suite is already setup, so this step is skipped and the suite information is simply copied from what was already initialized (`cdata_tile`) using the `cdata_target` optional argument.
- Populating the `cdata` structures
- When the dynamic mode is used, the `cdata` structures are filled with pointers to variables that are used by physics and whose memory is allocated by the host model. This is done using `ccpp_field_add` statements contained in the autogenerated include files. For the fast physics, this include file is named

`ccpp_fields_fast_physics.inc` and is placed after the call to `ccpp_init` for `cdata_tile` in the `atmosphere_init` subroutine of `atmosphere.F90`. For populating `cdata_domain` and `cdata_block`, IPD data types are initialized in the `atmos_model_init` subroutine of `atmos_model.F90`. The `Init_parm` DDT is filled directly in this routine and `IPD_initialize` (pointing to `GFS_initialize` and for populating diagnostics and restart DDTs) is called in order to fill the GFS DDTs that are used in the physics. Once the IPD data types are filled, they are passed to the ‘init’ step of the `CCPP_step` subroutine in `CCPP_driver.F90` where `ccpp_field_add` statements are included in `ccpp_fields_slow_physics.inc` after the calls to `ccpp_init` for the `cdata_domain` and `cdata_block` containers.

- Note: for the static mode, filling of the `cdata` containers with pointers to physics variables is not necessary. This is because the autogenerated *caps* for the physics groups (that contain calls to the member schemes) can fill in the argument variables without having to retrieve pointers to the actual data. This is possible because the metadata about host model variables (that are known at `ccpp_prebuild` time) contain all the information needed about the location (DDTs and local names) to pass into the autogenerated *caps* for their direct use.
- Providing interfaces to call the CCpp
 - Calling `ccpp_physics_init`
 - In order to call the initialization routines for the physics, `ccpp_physics_init` is called in the `atmosphere_init` subroutine of `atmosphere.F90` after the included `ccpp_field_add` calls for the fast physics. For the slow physics, the ‘physics_init’ step of the `CCPP_step` subroutine in `CCPP_driver.F90` is invoked immediately after the call to the ‘init’ step in the `atmos_model_init` subroutine of `atmos_model.F90`. Within the ‘physics_init’ step, calls to `ccpp_physics_init` for all blocks are executed.
 - Note: for the static mode, `ccpp_physics_init` is autogenerated and contained within `ccpp_static_api.F90`. As mentioned in the [CCPP API Section 6.4](#), it can be called to initialize groups as defined in the SDFs or the suite as a whole, depending on whether a group name is passed in as an optional argument.
 - Calling `ccpp_physics_run`
 - For actually running the physics within the FV3 time loop, `ccpp_physics_run` is called from a couple of different places in the FV3 source code. For the fast physics, `ccpp_physics_run` is called for the fast physics group from the `Lagrangian_to_Eulerian` subroutine of `fv_mapz.F90` within the dynamical core. For the rest of the physics, the subroutine `update_atmos_radiation_physics` in `atmos_model.F90` is called as part of the FV3 time loop. Within that subroutine, the various physics steps (defined as groups within a SDF) are called one after the other. The ‘time_vary’ step of the `CCPP_step` subroutine within `CCPP_driver.F90` is called. Since this step is called over the entire domain, the call to `ccpp_physics_run` is done once using `cdata_domain` and the `time_vary` group. The ‘radiation’, ‘physics’, and ‘stochastics’ steps of the `CCPP_step` subroutine are called next. For each of these steps within `CCPP_step`, there is a loop over the number of blocks for calling `ccpp_physics_run` with the appropriate group and component of the `cdata_block` array for the current block and thread.
 - Note: The execution of calls to `ccpp_physics_run` is different for the three build types. For the static mode, `ccpp_physics_run` is called from `ccpp_static_api.F90` and contains autogenerated *caps* for groups and the suite as a whole as defined in the SDFs.
 - calling `ccpp_physics_finalize` and `ccpp_finalize`
 - At the conclusion of the FV3 time loop, calls to finalize the physics are executed. For the fast physics, `ccpp_physics_finalize` is called from the `atmosphere_end` subroutine of `atmosphere.F90`. For the rest of the physics, the ‘finalize’ step of the `CCPP_step` subroutine in `CCPP_driver.F90` is called from the `atmos_model_end` subroutine in `atmos_model.F90`. Within the ‘finalize’ step of `CCPP_step`, calls for `ccpp_physics_finalize` and

`ccpp_finalize` are executed for every thread and block for `cdata_block`. Afterward, `ccpp_finalize` is called for `cdata_domain` and lastly, `cdata_tile`. (That is, the calls to `ccpp_finalize` are in reverse order than the calls to `ccpp_initialize`.) In addition, `cdata_block` is also deallocated in the 'finalize' step of `CCPP_step`.

- Note: for the static mode, `ccpp_physics_finalize` is autogenerated and contained within `ccpp_static_api.F90`. As mentioned in the [CCPP API Section 6.4](#), it can be called to finalize groups as defined in the current SDFs or the suite as a whole, depending on whether a group name is passed in as an optional argument.

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.

7.1.1 Authoritative Repositories

There are two authoritative repositories for the CCPP:

<https://github.com/NCAR/ccpp-framework>

<https://github.com/NCAR/ccpp-physics>

Users have read-only access to these repositories and as such cannot accidentally destroy any important (shared) branches of these authoritative repositories. Both CCPP repositories are public (no GitHub account required) and may be used directly to read or create forks. Write permission is generally restricted, however.

The following branches are recommended for CCPP developers:

Repository	Branch name
https://github.com/NCAR/ccpp-physics	master
https://github.com/NCAR/ccpp-framework	master

7.1.2 Directory Structure of ccpp/framework

The following is the directory structure for the ccpp/framework (condensed version):

├─ cmake	# cmake files for building
├─ doc	# Documentation for design/implementation and developers_
└─ guide	
├─ DevelopersGuide	
└─ images	
└─ img	
├─ schemes	# Example ccpp_prebuild_config.py
├─ check	
├─ scripts	# Scripts for ccpp_prebuild.py, metadata parser, etc.
├─ fortran_tools	
└─ parse_tools	
└─ src	# CCPP framework source code

(continues on next page)

(continued from previous page)

└─ tests	# SDFs and code for testing
└─ test	
└─ nemsfv3gfs	# NEMSfv3gfs regression test scripts
└─ tests	# Development for framework upgrades

7.1.3 Directory Structure of ccpp/physics

The following is the directory structure for the ccpp/physics (condensed version):

└─ physics	# CCPP physics source code and metadata files
└─ docs	# Scientific documentation (doxygen)
└─ img	# Figures for doxygen
└─ pdftxt	# Text files for documentation

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. These forks need to be created only once, and only for directories that users will contribute changes to. The following steps describe how to create a fork for the example of the ccpp-physics submodule/repository:

Go to <https://github.com/NCAR/ccpp-physics> and make sure you are signed in as your GitHub user.

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 that the repo name in the upper left (blue) will be either “NCAR” or “your GitHub name” which tells you which fork you are looking at.

Note that personal forks are not required until a user wishes to make code contributions. The procedure for how to check out the code laid out below can be followed without having created any forks beforehand.

7.2.2 Checking out the Code

Instructions are provided here for the ccpp-physics repository. Similar steps are required for the ccpp-frameworkx repository. The process for checking out the CCPP is described in the following, assuming access via https rather than ssh. We strongly recommend setting up passwordless access to GitHub (see <https://help.github.com/categories/authenticating-to-github/>).

Start with checking out the main repository from the NCAR GitHub

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

Checking out remote branches means that your local branches are in a detached state, since you cannot commit directly to a remote branch. As long as you are not making any code modifications, this is not a problem. If during your development work changes are made to the corresponding upstream branch, you can simply navigate to this repository and check out the updated version:

```
git remote update
git checkout upstream/master
cd ..
```

However, if you are making code changes, you must create a local branch.

```
git checkout -b my_local_development_branch
```

Once you are ready to contribute the code to the upstream repository, you need to create a PR (see next section). In order to do so, you first need to create your own fork of this repository (see previous section) 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 remote update
```

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 dtc/develop
```

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/NCAR/ccpp-physics (fetch)
upstream        https://github.com/NCAR/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
   dtc/develop
   master
```

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 the next section.

7.4 Contributing Code, Code Review Process

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

7.4.1 Creating a PR

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 associate PRs submitted to other repositories.

Several people (aka CODEOWNERS) are automatically added to the list of reviewers on the right hand side. If others should be reviewing the code, click on the “reviewers” item on the right hand side and enter their GitHub usernames

Once the PR has been approved, the change is merged to master 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.

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 that connects the host model with the *CCPP-Physics* schemes (see [Figures 3.2](#) and [3.3](#)). 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. In the case of the SCM, `ccpp_prebuild.py` must be run manually, as it is not incorporated in that model's build system. In the case of `ufs-weather-model`, `ccpp_prebuild.py` can be run manually or automatically as a step in the build system.

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 required to run all schemes in the *CCPP-Physics* pool.
- Compiles a list of variables provided by the host model.
- Matches these variables by their `standard_name`, checks for missing variables and mismatches of their attributes (e.g., units, rank, type, kind) and processes information on optional variables. Performs automatic unit conversions if a mismatch of units is detected between a scheme and the host model (see [Section 5.3](#) for details).
- For the static build only, filters out unused variables for a given suite.
- For the dynamic build only, creates Fortran code (`ccpp_modules_*.inc`, `ccpp_fields_*.inc`) that stores pointers to the host model variables in the `cdata` structure.
- Autogenerates software caps as appropriate, depending on the build type.
 - If dynamic, the script generates individual caps for all physics schemes.
 - If static, the script generates caps for the suite as a whole and physics groups as defined in the input *SDFs*; in addition, the *CCPP* API for the static build is generated.
- Populates makefiles with schemes, scheme dependencies, and caps. For the static build, statements to compile the static *CCPP* API are included as well.

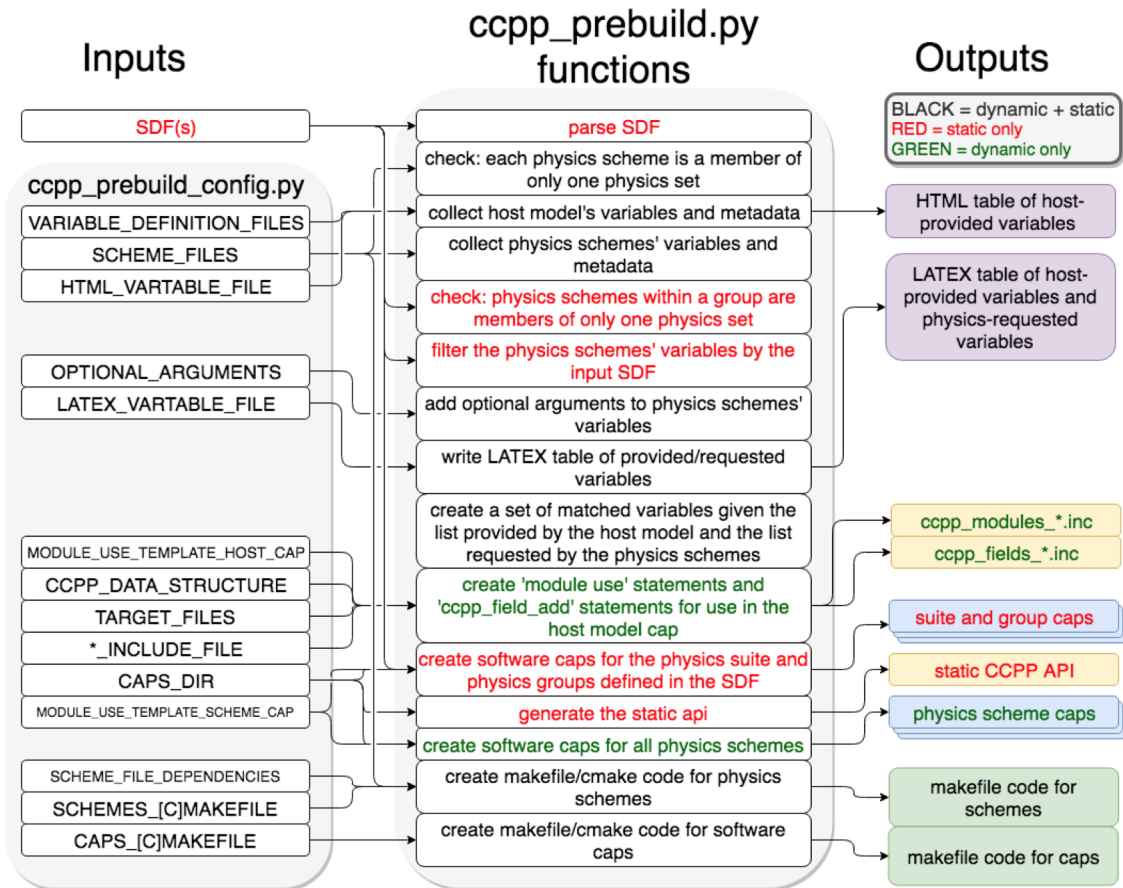


Fig. 8.1: Schematic of tasks automated by the `ccpp_prebuild.py` script and associated inputs and outputs. Red text denotes entries that are valid for the static build only, green text entries that are valid for the dynamic build only, and black text denotes entries valid for both the dynamic and static builds. 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.

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 TEST model in sub-directory `schemes/check` of the `ccpp-Framework` 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, it is assumed that 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 that `ccpp_prebuild.py` is executed from the same top-level directory.

```
# Host model identifier
HOST_MODEL_IDENTIFIER = "TEST"
# Add all files with metadata tables on the host model side,
# relative to basedir = top-level directory of host model
VARIABLE_DEFINITION_FILES = [
    'scm/src/gmtb_scm_type_defs.f90',
    'scm/src/gmtb_scm_physical_constants.f90'
]
# Add all physics scheme dependencies relative to basedir - note that the CCpp
# rules stipulate that dependencies are not shared between the schemes!
SCHEME_FILES_DEPENDENCIES = [] # can be empty
# Add all physics scheme files relative to basedir
SCHEME_FILES = {
    # Relative path : [ list of sets in which scheme may be called ]
    'ccpp/physics/physics/GFS_DCNV_generic.f90' : ['physics'],
    'ccpp/physics/physics/sfc_sice.f' : ['physics'],
}
# Auto-generated makefile/cmakefile snippets that contains all schemes
SCHEMES_MAKEFILE = 'ccpp/physics/CCPP_SCHEMES.mk'
SCHEMES_CMAKEFILE = 'ccpp/physics/CCPP_SCHEMES.cmake'
# CCpp host cap in which to insert the ccpp_field_add statements;
# determines the directory to place ccpp_{modules,fields}.inc
TARGET_FILES = ['scm/src/gmtb_scm.f90', ]
# Auto-generated makefile/cmakefile snippets that contains all caps
CAPS_MAKEFILE = 'ccpp/physics/CCPP_CAPS.mk'
CAPS_CMAKEFILE = 'ccpp/physics/CCPP_CAPS.cmake'
# Directory where to put all auto-generated physics caps
CAPS_DIR = 'ccpp/physics/physics'
# Directory where the suite definition files are stored
SUITES_DIR = 'ccpp/suites'

# Optional arguments - only required for schemes that use optional arguments.
# ccpp_prebuild.py will throw an exception if it encounters a scheme subroutine with
# optional arguments if no entry is made here. Possible values are:
OPTIONAL_ARGUMENTS = {
    #'subroutine_name_1' : 'all', #'subroutine_name_2' : 'none', #'subroutine_name_3' : [
    #'var1', 'var2'], }
# Names of Fortran include files in the host model cap (do not change);
# both files will be written to the directory of each target file
MODULE_INCLUDE_FILE = 'ccpp_modules.inc'
FIELDS_INCLUDE_FILE = 'ccpp_fields.inc'
# Directory where to write static API to
STATIC_API_DIR = 'scm/src'
```

(continues on next page)

(continued from previous page)

```
# HTML document containing the model-defined CCpp variables
HTML_VARIABLE_FILE = 'ccpp/physics/CCPP_VARIABLES.html'
# LaTeX document containing the provided vs requested CCpp variables
LATEX_VARIABLE_FILE = 'ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES.tex'
##### Template code to generate include files #####
# Name of the CCpp data structure in the host model cap;
# in the case of SCM, this is a vector with loop index i
CCPP_DATA_STRUCTURE = 'cdata(i)'

# EOF
```

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 regarding the `SCHEME_FILES` variable. This is a list of CCpp-compliant physics scheme entry/exit point source files. For each item in this list, a list of physics “sets” in which the scheme may be executed is included. A physics set refers to a collection of physics schemes that are able to be called together and executed in one software domain of a host model that do not share variables with schemes from another physics set. This feature was included to cater to the needs of the *UFS Weather Model*, which provides a clear-cut example of this concept. In this model, part of the microphysics scheme needed to be coupled more tightly with the dynamics, so this part of the microphysics code was put into a physics set labeled “fast_physics” which is executed within the dycore code. The variables in this physics set are distinct (in memory, due to a lack of a model variable registry) from variables used in the rest of the physics, which are part of the “slow_physics” set. In the future, it may be necessary to have additional sets, e.g. for chemistry or separate surface model components that do not share data/memory with other model components. For simpler models such as the CCpp SCM, only one physics set (labeled “physics”) is necessary. The concept of physics sets is different from physics “groups”, which are capable of sharing variables among their members and between groups but are used to organize schemes into sequential, callable units.

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 the top level directory. For the SCM, this script must be run to reconcile data provided by the SCM with data required by the physics schemes before compilation and to generate physics caps and makefile segments. For the *UFS Atmosphere* host model, the `ccpp_prebuild.py` script is called automatically by the *ufs-weather-model* build system when the *CCPP* build is requested (by running the *CCPP* regression tests or by passing the option `CCPP=Y` and others to the `compile.sh` script; see the compile commands defined in the *CCPP* regression test configurations for further details).

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. For the *UFS Atmosphere*, running `ccpp_prebuild.py` manually is identical to running it for the SCM (since the relative paths to their respective `ccpp_prebuild_config.py` files are identical), except it may be necessary to add the `--static` and `--suites` command-line arguments for the static option.

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 necessary input 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
--debug enable debugging output
```

--static enable a static build for a given suite definition file
 --suites SUITES SDF(s) to use (comma-separated, for static build only, without path)

So, the simplest possible invocation of the script (called from the host model's top level directory) would be:

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

↪py [--debug]
```

which assumes a dynamic build with a configuration script located at the specified path. The debug option can be used for more verbose output from the script.

For a static build (described above), where the *CCPP-Framework* and the physics libraries are statically linked to the executable and a set of one or more suites are defined at build-time, the --suites and --static options must be included. The *SDF*(s) should 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*. Unless the --static command-line argument is used with the script, it will assume dynamically linked libraries. An example of a static build using two *SDF*s is:

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

↪py --static \  

--suites=FV3_GFS_v15p2,FV3_GFS_v16beta
```

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 --static \  

--suites=FV3_GFS_v15p2,FV3_GFS_v16beta --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, please contact gmtb-help@ucar.edu.

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 --static and --suites options**

- 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 *SDF*s and matches what is in the source file (minus any *_init, *_run, *_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. ERROR: Scheme file `path/to/offending/scheme/source/file` belongs to multiple physics sets: `set1, set2`

Exception: Call to `check_unique_pset_per_scheme` failed.

- This error indicates that a scheme defined in the `SCHEME_FILES` variable of the supplied configuration file belongs to more than one set. Currently, a scheme can only belong to one physics set.

5. ERROR: Group `group1` contains schemes that belong to multiple physics sets: `set1, set2`

Exception: Call to `check_unique_pset_per_group` failed.

- This error indicates that one of the groups defined in the supplied *SDF*(s) contains schemes that belong to more than one physics set. Make sure that the group is defined correctly in the *SDF*(s) and that the schemes within the group belong to the same physics set (only one set per scheme is allowed at this time).

6. 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*)

7. **Exception:** Error parsing variable entry “erroneous variable metadata table entry data” in argument

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

8. **Exception:** New entry for variable `var_name` in argument table `variable_metadata_table_name` is i

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
optional = F
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
optional = F
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.

9. **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 *_init, *_run, or *_finalize.

10. **Exception:** Encountered closing statement "end" without descriptor (subroutine, module

- This script expects that subroutines and modules end with descriptor and name, e.g. 'end subroutine subroutine_name'.

11. **Exception:** New entry for variable var_name in argument table of subroutine scheme_subroutine

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
optional = F
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
optional = F
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”.

12. **ERROR: Check that all subroutines in module module_name have the same root name:**
i.e. scheme_A_init, scheme_A_run, scheme_A_finalize Here is a list of the subroutine names for scheme scheme_name: scheme_name_finalize, scheme_name_run
* All schemes must have *_init, *_run, *_finalize subroutines contained within its entry/exit point module.

13. **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.

14. **ERROR: error, variable X requested by MODULE_Y SCHEME_Z SUBROUTINE_A cannot be identified**

- 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.

15. **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
optional = F
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
optional = F
target = None
container =
```


16. 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)

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 5.3](#) 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.
- Examine scheme-specific and suite interstitials to see what needs to be replaced/changed; then check existing scheme interstitial and determine what needs to be 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 upper-case 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
 - `gmtb-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',  
...]
```

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

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

- 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
 - `gmtb-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 segment faults related to variables and array allocation.
 - Make sure 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 *prebuild* script for success/failure and associated messages.
 - Compile code in `DEBUG` mode, run through debugger if necessary (`gdb`, Allinea DDT, `totalview`, ...). See [Chapter %s](#) for information on debugging.
 - 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.

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
ESPC	Earth System Prediction Capability
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
IPD	Interoperable Physics Driver
LSM	Land Surface Model
MG	Morrison-Gottelman
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
NEMSV3gfs	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 10.1 – continued from previous page

Abbreviation	Explanation
NRL	Naval Research Laboratory
NSST	Near Sea Surface Temperature ocean scheme
NUOPC	National Unified Operational Prediction Capability
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)
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
VLab	Virtual Laboratory
WRF	Weather Research and Forecasting

GLOSSARY

CCPP Model agnostic, vetted, collection of codes containing atmospheric physical parameterizations and suites for use in NWP along with a framework that connects the physics to host models

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

Dynamic CCPP build A CCPP build type in which the *CCPP-Framework* and *CCPP-physics* libraries are dynamically linked to the executable and all CCPP-compliant schemes available in the library can be invoked at runtime. This build type is available for both the SCM and the UFS Atmosphere

“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

Group cap Autogenerated interface between a group of physics schemes and the host model. They are used only in the static CCPP build, and effectively replace the code from *CCPP-Framework* that provides the flexibility of the dynamic CCPP build

Host model/application An atmospheric model 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.

Multi-suite static CCPP build A static CCPP build type in which a set of physics suites is specified at compile time from which one can be chosen at runtime.

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.

NUOPC The National Unified Operational Prediction Capability is a consortium of Navy, NOAA, and Air Force modelers and their research partners. It aims to advance the weather modeling systems used by meteorologists, mission planners, and decision makers. NUOPC partners are working toward a common model architecture - a standard way of building models - in order to make it easier to collaboratively build modeling systems.

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 (AMS Glossary)

Physics *cap* Autogenerated interface between an individual physics scheme and the host model. Used only in the dynamic CCpp build

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

PROD Compiler flags used by NCEP for operational runs of the UFS Atmosphere and by EMC for regression tests of the code

REPRO Compiler flags used by EMC to guarantee reproducibility of the UFS Atmosphere code

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)

“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

Static CCpp build A build type in which the *CCPP-Framework* and the *CCPP-physics* libraries are statically linked to the executable and only the suites determined at compile time can be invoked at runtime. This build type is only available for the UFS Atmosphere

Subcycling Executing a physics scheme more frequently (with a shorter timestep) than the rest of the model physics or dynamics

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

Suite *cap* Autogenerated interface between an entire suite of physics schemes and the host model. They are used only in the static CCpp build and consist of calls to autogenerated group caps. They may be used to call an entire suite at once or to call a specific group within a physics suite

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 Global medium-range, weather-prediction model previously known as NEMSfv3gfs or FV3GFS used to create forecasts.

VLab Virtual Laboratory - a service and information technology framework, that enables NOAA employees and their partners to share ideas, collaborate, engage in software development, and conduct applied research (<https://www.nws.noaa.gov/mdl/vlab/>)

.xsd file extension XML schema definition

Symbols

"Fast" physics, [85](#)
"Slow" physics, [86](#)
.xsd file extension, [86](#)

C

CCPP, [85](#)
CCPP-Framework, [85](#)
CCPP-Physics, [85](#)

D

Dynamic CCPP build, [85](#)

G

Group, [85](#)
Group cap, [85](#)

H

Host model/application, [85](#)

I

Interstitial scheme, [85](#)

M

Multi-suite static CCPP build, [85](#)

N

NEMS, [85](#)
NUOPC, [85](#)

P

Parameterization, [86](#)
Physics cap, [86](#)
Primary scheme, [86](#)
PROD, [86](#)

R

REPRO, [86](#)

S

Scheme, [86](#)

SDF, [86](#)
Set, [86](#)
Standard_name, [86](#)
Static CCPP build, [86](#)
Subcycling, [86](#)
Suite, [86](#)
Suite cap, [86](#)

U

UFS, [86](#)
UFS Atmosphere, [86](#)
UFS Weather Model, [86](#)

V

VLab, [86](#)