
CCPP Training Documentation

Release v1.0

May 09, 2019

CONTENTS

1	Introduction	1
1.1	How To Use This Document	1
2	CCPP Overview	3
3	CCPP-Compliant Physics Parameterizations	7
3.1	General rules	7
3.2	Input/output variable (argument) rules	10
3.3	Coding rules	11
3.4	Parallel programming rules	11
4	Scientific Documentation Rules	13
4.1	Doxygen comments and commands	13
4.2	Doxygen Documentation Style	14
4.2.1	Doxygen files	14
4.2.2	Doxygen Overview Page	15
4.2.3	Physics Scheme Pages	15
4.2.4	Doxygen Modules	17
4.2.5	Bibliography	19
4.2.6	Equations	19
4.3	Doxygen Configuration	20
4.3.1	Configuration file	20
4.3.2	Diagrams	21
4.4	Using Doxygen	22
5	Configuring and Building Options (Weiwei)	23
6	Building and Running Host Models (Weiwei)	25
7	Adding a new scheme (Weiwei)	27
8	Glossary	29
	Bibliography	31
	Index	33

INTRODUCTION

This document contains training material for the Common Community Physics Package (CCPP). It describes the:

- CCPP overview
- physics schemes and interstitials
- suite definition files
- CCPP-compliant parameterizations
- adding a new scheme/suite
- host-side coding
- fundamentals of obtaining, pre-building, building and running the CCPP with NEMSfv3gfs
- CCPP code management and governance

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

Please send questions and comments to the help desk: gmb-help@ucar.edu

1.1 How To Use This Document

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!
AaBbCc123	What you type contrasted with on-screen computer output	host\$ su
%	Command-line prompt	% cd \$TOP_DIR

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

```
% mkdir ${TOP_DIR}
```


CCPP OVERVIEW

Ideas for this project originated within the Earth System Prediction Capability (ESPC) Physics Interoperability group, which has representatives from NCAR, the Navy, NOAA Research, 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 NOAA 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 2.1. There are two distinct parts to the CCPP: a library of physical parameterizations that conforms to selected standards (*CCPP-Physics*) and an infrastructure (*CCPP-Framework*) that enables connecting the physics to host models.

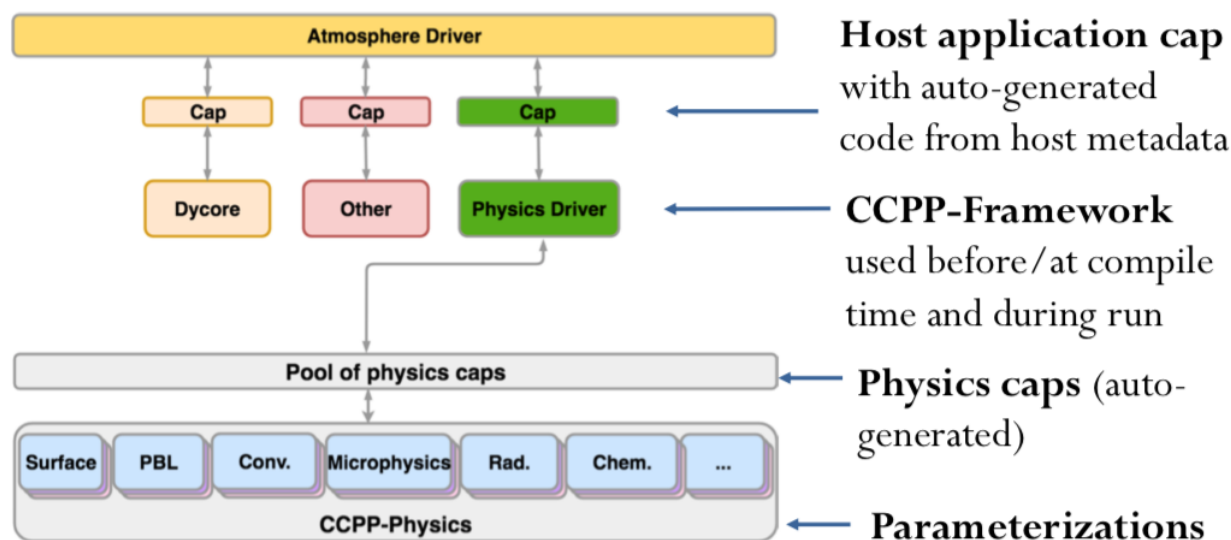


Fig. 2.1: Architecture of the CCPP and its connection with a host model, here represented as the driver for an atmospheric model (yellow box). The dynamical core (dycore), the physics, and other aspects of the model (such as coupling) are connected to the driving host through the pool of physics caps. The CCPP-Physics is denoted by the gray box at the bottom of the physics, and encompasses the parameterizations, which are accompanied by parameterization caps.

The host model also 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 model, and check whether they are available (otherwise an error is issued). This process has greatly served to expose the variables passed between physics and dynamics, and also to clarify how suites exchange information among parameterizations. During runtime, the CCPP-Framework is responsible for communicating the necessary variables between the host model and the parameterizations.

Building the CCPP involves a *prebuild* step, and an actual build step. As of this writing, the CCPP has been connected with the GMTB Single Column Model (SCM) and with various configurations of NOAA's Unified Forecast System (UFS). As described in detail later in this document, there are multiple modes of building the CCPP along with the UFS, and the user should choose the mode that best matches their needs. With the CCPP dynamic build, all CCPP-compliant parameterizations are compiled into a library which is linked to the host model at runtime. Conversely, with the CCPP static build, a single suite is compiled into a library and linked to the host model when it is compiled. The dynamic build favors flexibility, as users can choose 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. When the CCPP is used with the SCM, the dynamic build is used.

The CCPP Physics is envisioned to contain parameterizations and suites that are used operationally in the UFS, as well as parameterizations that are under development for possible transition to operations in the future. By distributing and supporting the CCPP to the scientific community, a large number of scientists can experiment with and innovate physics. Figure 2.2 shows the CCPP ecosystem, a significant effort in transition of research to operations that can benefit physics developers, users, and NOAA operations.

The CCPP-Physics and its associated CCPP-Framework are developed as open source codes, 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>).



Fig. 2.2: CCPP ecosystem

The first public release of CCPP took place in April 2018 and included all parameterizations of the operational GFS v14, along with the ability to connect to the GMTB SCM. The second public release of the CCPP took place in August of 2018 and additionally included the physics suite tested in advance of the GFS v15 implementation. Since

then, additional parameterizations have been made CCPP-compliant, in order to encompass the suites that are under consideration for GFS v16. A summary of the suites supported in the CCPP can be found in Table 2.1. In addition to the schemes listed, it is expected that, as time goes on, other parameterizations will be considered for inclusion in the CCPP.

The CCPP is governed by the groups that contribute to its development. Governance for the CCPP-Physics is currently led by NOAA, and the GMTB works with EMC and the NGGPS Program Office to determine which schemes and suites should be included and supported. Governance for the CCPP-Framework is done jointly by NOAA and NCAR, and more information can be found at <https://github.com/NCAR/ccpp-framework/wiki>. Additional information can also be found on the GMTB website at <https://dtcenter.org/gmtb/users/ccpp>. Please direct all inquiries to gmtb-help@ucar.edu.

Table 2.1: *Schemes and suites available in the CCPP*

Schemes/Suites	GFS v14	FV3GFSv1	FV3GFS v1.1	EMC/CPT	GSD
Microphysics	Zhao-Carr	GFDL	GFDL	aaMG3	aaThompson
PBL	EDMF	EDMF	TKE EDMF	EDMF	MYNN
Deep cu	saSAS	saSAS	saSAS	CSAW	GF
Shallow cu	saSAS	saSAS	saSAS	saSAS	MYNN
Radiation	RRTMG	RRTMG	RRTMG	RRTMG	RRTMG
Surface Layer	GFS	GFS	GFS	GFS	GFS
Land surface	Noah	Noah	Noah	Noah	RUC
Ozone	NRL 2006	NRL 2015	NRL 2015	NRL 2015	NRL 2015
H2O	None	NRL 2015	NRL 2015	NRL 2015	NRL 2015

The first row lists the suites currently available in the CCPP. The first column denotes the types of parameterizations, where cu stands for convective parameterization and H2O for stratospheric water vapor. The operational GFS V14 suite (column 1) used Zhao-Carr microphysics, Eddy-Diffusivity Mass Flux (EDMF) PBL, scale-aware Simplified Arakawa Schubert (saSAS) convection, Rapid Radiation Transfer Model for General Circulation Models (RRTMG) radiation, GFS surface layer, Noah LSM, Navy Research Laboratory (NRL) ozone, and no stratospheric water vapor scheme. The FV3GFS v1 suite, uses the GFDL microphysics and the NRL 2015 ozone and photolysis schemes. The other three suites are candidates for future operational implementations. The FV3GFS v1.1 suite uses Turbulent Kinetic Energy (TKE)-based EDMF, the EMC and Climate Process Team (CPT) suite uses the scale-aware (aa) Morrison-Gottelman 3 (MG3) microphysics and Chikira-Sugiyama convection with Arakawa-Wu extension (CSAW) deep convection, and the NOAA Global Systems Division (GSD) suite uses aerosol aware Thompson microphysics, Mellor-Yamada-Nakashini-Niino (MYNN) PBL and shallow convection, and Rapid Update Cycle (RUC) LSM.

CCPP-COMPLIANT PHYSICS PARAMETERIZATIONS

A basic description of the rules for a parameterization to be considered CCPP-compliant is summarized in this section (see also [HBCF18]).

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

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, they should be coded as two separate schemes to allow for 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 Morrison-Gottelman microphysics, in which a simple entrypoint 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.

However, there are some reasons that may justify the implementation of a driver:

- To preserve schemes that are also distributed outside of the CCPP. For example, the Thompson microphysics scheme is distributed both with the 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 and CCPP distributions.
- To deal with optional arguments.
- To perform unit conversions and array transformations, such as flip the vertical direction and rearrange index order. Note that, in the future, these capabilities will be included in the CCPP-Framework so that schemes do not have to perform these operations.

Somewhere here include info on schemes that have options, such as MG2 and MG3 are in the same scheme. MYNN has many options. So does GF. It is a grey area what is defined as a new scheme versus options within a scheme.

3.1 General rules

Listing 1 contains a Fortran template for a CCPP-compliant scheme, which can also be found in `ccpp/framework/doc/DevelopersGuide/scheme_template.F90`.

```

module scheme_template

    contains

    subroutine scheme_template_init ()
    end subroutine scheme_template_init

    subroutine scheme_template_finalize()
    end subroutine scheme_template_finalize

!> \section arg_table_scheme_template_run Argument Table
!! | local_name | standard_name      | long_name                                     |_
↪units | rank | type              | kind | intent | optional |
!! |-----|-----|-----|-----|-----|-----|
↪--|-----|-----|-----|-----|-----|
!! | errmsg      | ccpp_error_message | error message for error handling in CCPP |_
↪none | 0 | character | len=* | out      | F      |
!! | errflg      | ccpp_error_flag    | error flag for error handling in CCPP    |_
↪flag | 0 | integer   |      | out      | F      |
!!

    subroutine scheme_template_run (errmsg, errflg)

        implicit none

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

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

        continue

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

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

        !--- actual code
        ! add your code here

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

        return

    end subroutine scheme_template_run

end module scheme_template

```

- Each scheme must be in its own module (module name = scheme name) and must have three entry points (subroutines) starting with the name of the module:

```
module scheme
```

(continues on next page)

(continued from previous page)

```

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

```

The `_init` and `_finalize` routines are run automatically when the CCPP physics are initialized and finalized, respectively. These routines may be called more than once, depending on the host model's parallelization strategy, and as such must be idempotent (that is, the answer must be the same when the subroutine is called multiple times).

- Additional modules (`scheme_pre` and `scheme_post`) can be used if there is any part of the physics scheme that must be executed before or after the module `scheme` defined above. These situations are described in more detail in Section 5.2 and 6.1.2. If additional modules are included, they also must have three entry points:

```

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

module scheme_post
  implicit none
  private
  public :: scheme_post_init, scheme_post_run, &
           scheme_post_finalize
  contains
    subroutine scheme_post_init()
    end subroutine scheme_post_init
    subroutine scheme_post_finalize()
    end subroutine scheme_post_finalize
    subroutine scheme_post_run()
    end subroutine scheme_post_run
end module scheme_post

```

- All CCPP entrypoint schemes need to be accompanied by a table that describes the arguments to the subroutine (see example in [Listing 1](#)). However, empty schemes (e.g., `scheme_template_init` in [Listing 1](#)) are excepted and need no argument table.
- The order of arguments in the table does not need to be the same as in the argument list of the subroutine, but that is preferable.
- The argument table must precede the entry point subroutine, and must start with `!> \section arg_table_subroutine_name Argument Table` and end with a line containing only `!!`

- All external information required by the scheme must be passed in via the argument list.
 - Statements such as `use EXTERNAL_MODULE` should not be used for passing in data and all physical constants should go through the argument list.
- If the width of an argument table exceeds 250 characters, the argument table should be wrapped. in C preprocessor directives:

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

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

3.2 Input/output variable (argument) rules

- Variables available for CCPP physics schemes are identified by their unique `standard_name`. While an effort is made to comply with existing `standard_name` definitions of the CF conventions (<http://cfconventions.org>), additional names are 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 3) 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 (units, rank, index ordering) must match the specifications on the host model side, but sub-slices can be used/added in the host model. For example, in `GFS_typedefs.F90`, tendencies can be split so they can be used in the necessary physics scheme:

```
!! | IPD_Data(nb)%Intdiag%dt3dt(:, :, 1) | cumulative_change_in_temperature_due_to_
↪longwave_radiation
!! | IPD_Data(nb)%Intdiag%dt3dt(:, :, 2) | cumulative_change_in_temperature_due_to_
↪shortwave_radiation_and_orographic_gravity_wave_drag
!! | IPD_Data(nb)%Intdiag%dt3dt(:, :, 3) | cumulative_change_in_temperature_due_to_PBL
```

- The two mandatory variables that every scheme must accept as `intent(out)` arguments are `errmsg` and `errflg` (see also coding rules).
- At present, only two types of variable definitions are supported by the CCPP framework:
 - Standard Intrinsic Fortran variables are preferred (character, integer, logical, real). For character variables, the length should be specified as `\`. All others can have a `kind` attribute of a kind type defined by the host model.
 - Derived data types (DDTs). While the use of DDTs is discouraged in general, some use cases may justify their application (e.g. DDTs for chemistry that contain tracer arrays 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 generality. Where possible, DDTs should be broken into components that could be usable for another scheme of the same type. Where DDTs currently exist within CCPP-compliant schemes, they are likely there due to expediency concerns and should eventually be phased out.

- 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 in the suite definition file (SDF; see also GMTB Single Column Model Technical Guide v2.1, chapter 6.1.3, <https://dtcenter.org/gmtb/users/ccpp/docs>), the loop counter can be obtained from CCPP as an `intent (in)` variable (see Listings 3.1 and 3.2 for a mandatory list of variables that are provided by the CCPP framework and/or the host model for this and other purposes).

3.3 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 none` is not allowed.
- 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/stderr`.
- Line lengths of up to 120 characters are suggested for better readability (exception: CCPP metadata argument tables).

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

3.4 Parallel programming rules

Most often shared memory (OpenMP) and MPI 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 in `intent(in)` argument in the argument list (Listings 3.1 and 3.2).
- MPI communication is allowed in the `_init` and `_finalize` phase for the purpose of computing, reading or writing scheme-specific data that is independent of the host model's data decomposition. An example is the initial read of a lookup table of aerosol properties by one or more MPI processes, 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 (Listings 3.1 and 3.2).
 - The use of MPI is restricted to global communications: `barrier`, `broadcast`, `gather`, `scatter`, `reduce`.
 - The use of `MPI_COMM_WORLD` is not allowed.
 - The use of point-to-point communication 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.

```
#ifdef MPI
  use mpi
#endif
#ifdef OPENMP
  use omp_lib
#endif
...
#ifdef MPI
  call MPI_BARRIER(mpicomm, ierr)
#endif

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

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

SCIENTIFIC DOCUMENTATION RULES

Technically, scientific documentation is not needed for a parameterization to work with the CCPP. However, scientific and technical documentation is important for code maintenance and for fostering understanding among stakeholders and 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 platform for generating human-readable output due to its built-in functionality with Fortran, its high level of configurability, and its ability to parse in-line 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 in-line comments. It covers what kind of information should be in the documentation, how to mark up the in-line comments so that doxygen will parse it correctly, where to put various comments within the code, and how to configure and run doxygen to generate HTML (or other format) output. Part of this documentation, namely subroutine argument tables, have functional significance as part of the CCPP infrastructure. These tables 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 v2.0 parameterizations is a good way of getting started in writing documentation for a new scheme. The CCPP Scientific Documentation can be converted to html format (see https://dtcenter.org/gmtb/users/ccpp/docs/sci_doc_v2/).

4.1 Doxygen comments and commands

Firstly, be aware that regular Fortran comments using “!” are not parsed by Doxygen. The Doxygen in-line comment block begins with “!>”, and subsequent lines begin with “!!”. All Doxygen commands start with a backslash (“\”) or an at-sign (“@”). Example in the first line of each Fortran file, brief one sentence overview of the file purpose using Doxygen command “\file”:

```
!> \file gwdps.f
!! This file is the parameterization of orographic gravity wave
!! drag and mountain blocking.
```

The parameter definition begins with “!<”, where the sign “<” just tells Doxygen that documentation follows after the member. 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
```

If you simply want Doxygen to use the original comments, you must modify the comment line to use “!>” at the first line and “!!” at the subsequent lines.

4.2 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

4.2.1 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 gwdeps.f
!! This file is the parameterization of orographic gravity wave
!! drag and mountain blocking.
```

The brief description for each file is displayed next to the source filename on the Doxygen-generated “File List” page (see Figure ??).

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]
 gwdeps.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

4.2.2 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 GMTB 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
...
*/
```

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 `code_overview.txt`:

```
The FV3GFS physics suite uses the parameterizations in the following order,
as defined in \c suite_SCM_GFS_2017_updated.xml and \c suite_SCM_GFS_2018_updated.
↪xml:
+ @ref GFS_RRTMG
+ @ref GFS_SFCLYR
+ @ref GFS_NSST
+ @ref GFS_NOAH
+ @ref GFS_SFCSICE
+ @ref GFS_HEDMF
+ @ref GFS_GWDPS
+ @ref GFS_RAYLEIGH
+ @ref GFS_OZPHYS
+ @ref GFS_H2OPHYS
+ @ref GFS_SAMFdeep
+ @ref GFS_GWDC
+ @ref GFS_SAMFshal
+ GFS Microphysics (MP) scheme option:
  + @ref GFDL_cloud
    + @ref gfdlmp
    + @ref fast_sat_adj
  + @ref GFS_ZHAOC
    + @ref condense
    + @ref precip
+ @ref GFS_CALPRECIPTYPE
```

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.

4.2.3 Physics Scheme Pages

Each major scheme in CCPP should have its own scheme page containing an overview of the parameterization. This page is not tied to the Fortran code directly; instead, it is created with a separate text file that starts with the command “`\page`”. Each page has a label (e.g., “`GFS_ZHAOC`” in the following example) and a user-visible title (“GFS Zhao-Carr Microphysics 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_ZHAOC GFS Zhao-Carr Microphysics Scheme

\section des_zhao Description
This is the GFS scheme for grid-scale condensation and precipitation which is
based on Zhao and Carr (1997) \cite zhao_and_carr_1997 and Sundqvist et al.
(1989) \cite sundqvist_et_al_1989 .
...
Figure 1 shows a schematic illustration of this scheme.
...
\image html GFS_zhaocarr_schematic.png "Figure 1: Schematic illustration of the_
↪precipitation scheme" width=10cm
...

\section intro_zhao Intraphysics Communication
+ For grid-scale condensation and evaporation of cloud process
(\ref arg_table_zhaocarr_gscond_run)
+ For precipitation (snow or rain) production
(\ref arg_table_zhaocarr_precpd_run)

\section Gen_zhao General Algorithm
+ \ref general_gscond
+ \ref general_precpd

*/

```

The HTML result is [here](#). The physics scheme page will often describe the following:

a. **Description section (“\section”), which usually includes:**

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

To insert images into Doxygen documentation, you’ll need to have your images ready in a graphical format (i.e., .png) depending on which Doxygen output you’re willing to generate. For example, for LaTeX output the images must be provided in Encapsulated PostScript (.eps); For HTML output the images could be provided in the Portable Network Graphic (.png) format. Example of including the same image in multiple formats for HTML and LaTeX outputs:

```

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

```

In the CCPP, images are put under `./docs/img/` directory.

b. **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.

c. **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. Here is an example of GFS Zhao-Carr microphysics scheme 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.

4.2.4 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.
!! ...
!!\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
- **An argument table that includes entries for each subroutine argument**
 - The argument table content should be immediately preceded by the following line:

```
!!\section arg_table_SUBROUTINE_NAME
```

This line is also functional documentation used during the CCPP prebuild step. The first line of the table should contain the following “header” names

- a. **local_name**: contains the local subroutine variable name
- b. **standard_name**: CF-compliant standard name
- c. **long_name**: a short description
- d. **units**: format follows “unit exponent”, i.e. m2 s-2 for m2/s2
- e. **rank**: 0 for scalar, 1 for 1-D array, 2 for 2-D array, etc.
- f. **type**: integer, real, logical, etc.
- g. **kind**: the specified floating point precision kind (at present, to be extended to different integer kinds in the future)
- h. **intent**: in, out, inout
- i. **optional**: T/F

The argument table should be immediately followed by a blank doxygen line “!!”, which is needed to denote the end of an argument table. Here is an example :

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

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

- 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 argument table 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] delt    real, physics time step
!! ...
!! \section general_mfpbl mfpbl General Algorithm
!! -# Determine an updraft parcel's entrainment rate, buoyancy, and vertical_
↳velocity.
!! -# Recalculate the PBL height ...
!! -# Calculate the mass flux profile and updraft properties.
!! \section detailed_mfpbl mfpbl Detailed Algorithm
!> @{
    subroutine mfpbl(im,ix,km,ntrac,delt,cnvflg,                &
        &  z1,zm,thvx,q1,t1,u1,v1,hpbl,kpbl,                &
        &  sflx,ustar,wstar,xmf,tcko,qcko,ucko,vcko)
        ...
    end subroutine mfpbl
!> @}
```

4.2.5 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
```

4.2.6 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.

4.3 Doxygen Configuration

4.3.1 Configuration file

The CCPP contains a Doxygen configuration file `./ccpp/physics/physics/docs/ccpplatex_dox`, 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 html documentation relative `OUTPUT_DIRECTORY`:

```
HTML_OUTPUT = html
HTML_FILE_EXTENSION = .html
```

where `HTML_FILE_EXTENSION` tells what the extension of the html files should be.

- 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:

4.3.2 Diagrams

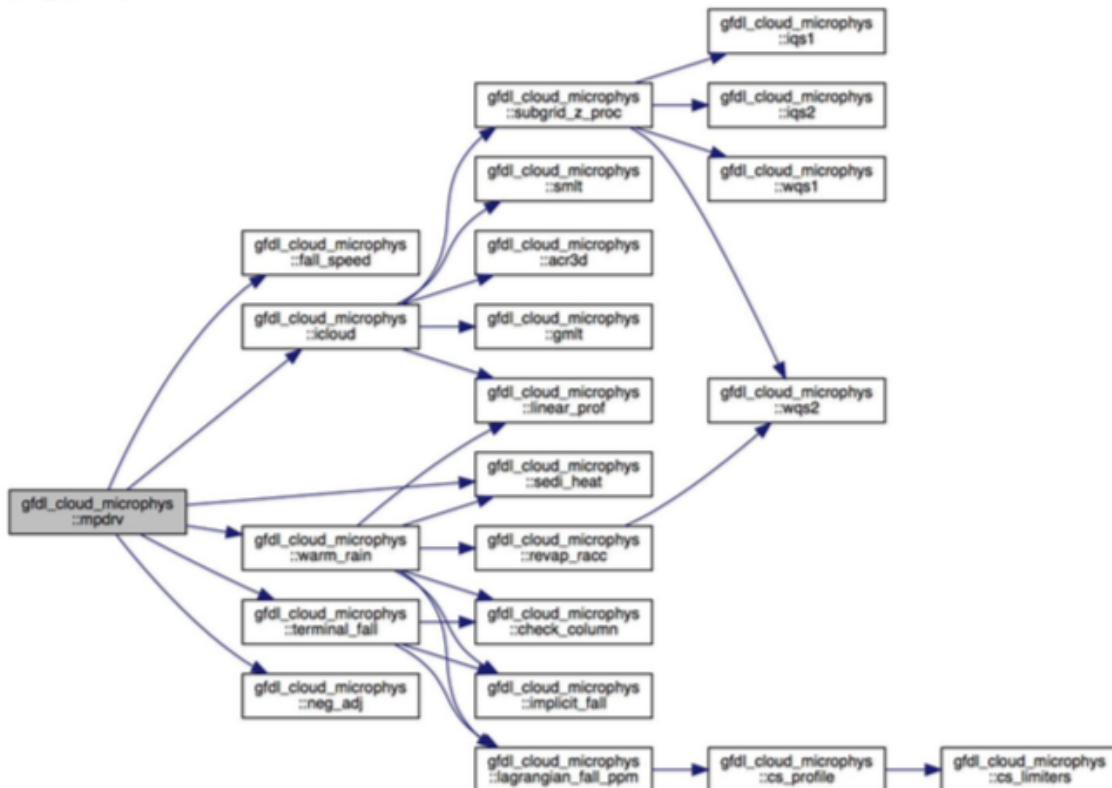
On its own, Doxygen is capable of creating simple text-based class diagrams. With the help of the additional software GraphViz, Doxygen can generate additional graphics-based diagrams, optionally in UML style. To enable GraphViz support, the configure file parameter “HAVE_DOT” must be set to “YES”.

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:

Here is the call graph for this function:



4.4 Using Doxygen

In order to generate Doxygen-based documentation, you need to follow four steps:

1. Have the Doxygen executable installed on your computer. This is already done on theia machine. Add the following line into `.cshrc` file under your home directory:

```
alias doxygen/scratch4/BMC/gmtb/doxygen-1.8.10/bin/doxygen
```

Source your `.cshrc` file.

2. Document your code, including Doxygen main page, scheme pages and in-line comments within source code as described above.
3. Prepare a Bibliography file in BibTeX format for paper referred in the physics suite.
4. 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.
5. Run the Doxygen command from the command line with the doxygen configuration file given as an argument:

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

Running this command may output compiler-like warning or errors that need to be fixed in order to produce proper output. Output is generated depending on the type specified (HTML, LaTeX, etc.) in the configuration file and is put in a location specified in the configuration file. The generated HTML documentation can be viewed by pointing a HTML browser to the `index.html` file in the `./docs/doc/html/` directory.

For precise instructions on creating the scientific documentation, contact the GMTB helpdesk at gmtb-help@ucar.edu.

CONFIGURING AND BUILDING OPTIONS (WEIWEI)

BUILDING AND RUNNING HOST MODELS (WEIWEI)

ADDING A NEW SCHEME (WEIWEI)

GLOSSARY

CCPP the Common Community Physics Package designed to facilitate the implementation of physics innovations in state-of-the-art atmospheric models

CCPP framework the driver that connects the physics schemes with a host model

CCPP physics the pool of CCPP-compliant physics schemes

dynamic CCPP build CCPP framework and physics libraries are dynamically linked to the executable

entry/exit point subroutine a subroutine that is exposed to the CCPP framework and serves as a public interface for the underlying code; for example, scheme A uses several source files and subroutines to parameterize a physical process, but its interaction with a host model (exchange of arguments) is entirely through one subroutine, which is defined as the entry/exit point subroutine

group Set of physics schemes within a suite definition file (SDF) that are called together, for example ‘radiation’

host cap manually-generated interface (that includes some auto-generated code) between the host model/application and the CCPP framework and physics

host model/application an atmospheric driver or other model that allocates memory, provides metadata for the variables passed into and out of the physics, and controls time-stepping; the framework that the physics schemes are connected with

hybrid CCPP enables use of non-CCPP physics and CCPP-compliant physics in the same run

interstitial scheme a scheme used to calculate the missing variables based on existing variables in the model and run before/after the scheme and cannot be part of the scheme itself. At present, adding interstitial schemes should be done in cooperation with the GMTB Help Desk

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 auto-generated interface between an individual physics scheme and the CCPP framework

set Collection of physics schemes that do not share memory (e.g. fast and slow physics)

standalone CCPP non-Hybrid CCPP. Only CCPP-compliant parameterizations can be used. Physics scheme selection and order is determined by an external suite definition file (SDF)

standard_name variable names based on CF conventions (<http://cfconventions.org>) that are uniquely identified by CCPP physics schemes and provided by a host model

static CCPP build CCPP framework and physics libraries are statically linked to the executable

subcycling executing a physics scheme in a loop with a shorter timestep than the rest of physics (or the dynamics)

suite a collection of physics schemes and interstitial schemes that is known to work well together

suite definition file (SDF) an external file containing information about a physics suite's construction; its contents describe the schemes that are called, what order they are called, whether they are subcycled, and whether they are grouped into units to be called together with intervening non-physics code

.xsd file extension XML schema definition

BIBLIOGRAPHY

- [HBCF18] D. Heinzeller, L. Bernardet, L. Carson, and G. Firl. Common community physics package (ccpp) developers' guide v2.0. 2018. URL: <https://dtcenter.org/gmtb/users/ccpp/docs/CCPP-DevGuide-v2.pdf>.

Symbols

.xsd file extension, [30](#)

C

CCPP, [29](#)

CCPP framework, [29](#)

CCPP physics, [29](#)

D

dynamic CCPP build, [29](#)

E

entry/exit point subroutine, [29](#)

G

group, [29](#)

H

host cap, [29](#)

host model/application, [29](#)

hybrid CCPP, [29](#)

I

interstitial scheme, [29](#)

P

parameterization, [29](#)

physics cap, [29](#)

S

set, [29](#)

standalone CCPP, [29](#)

standard_name, [29](#)

static CCPP build, [29](#)

subcycling, [29](#)

suite, [29](#)

suite definition file (*SDF*), [30](#)