CCPP Training
College Park, MD, March 12-13, 2019

CCPP prebuild

Dom Heinzeller
Global Model Test Bed

Developmental Testbed Center

Outline of Talk

- CCPP prebuild versus actual build
- Usage
- CCPP prebuild in the dynamic build
- CCPP prebuild in the static build
- How to modify CCPP prebuild config (for physics developers)
- Extra credit
- Wrap up

CCPP prebuild versus actual build

CCPP prebuild …

- is a set of Python scripts in `ccpp/framework/scripts/`
- requires a host-model dependent configuration file
- runs before ccpp-framework and ccpp-physics are compiled
- is called by NEMS builder before CCPP component is built
- establishes the link between the variables provided by the host model and the variables required by the physics schemes
- creates files required by the build system, auto-generates code that is used in the host model, generates caps for running the physics (different for dynamic/static build)

CCPP prebuild script: usage

ccpp_prebuild.py is called from the NEMSfv3gfs top-level directory

- dynamic build

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

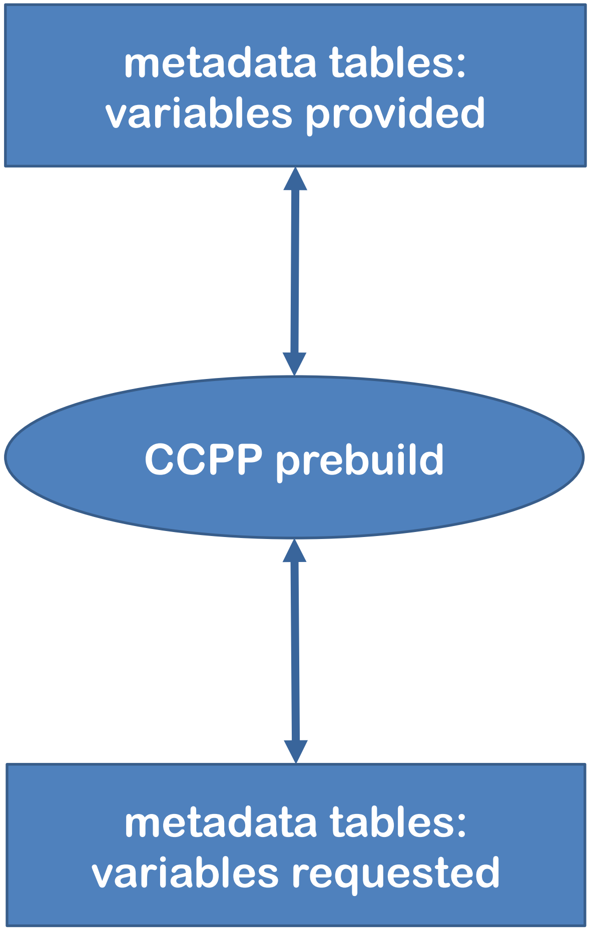
- static build, requires a suite definition file

```
./ccpp/framework/scripts/ccpp_prebuild.py [--debug] \
       --config=ccpp/config/ccpp_prebuild_config.py \
       --static \
       --suite=ccpp/suites/suite_FV3_CPT.xml
```

CCPP prebuild in the dynamic build

ccpp_prebuild.py

- requires metadata tables on both sides
- checks requested vs provided variables by standard_name
- checks units, rank, type (more to come)
- creates Fortran code that adds pointers to the host model variables and stores them in the ccpp-data structure (ccpp_fields_*.inc)
- creates caps for physics schemes
- populates makefiles with schemes and caps

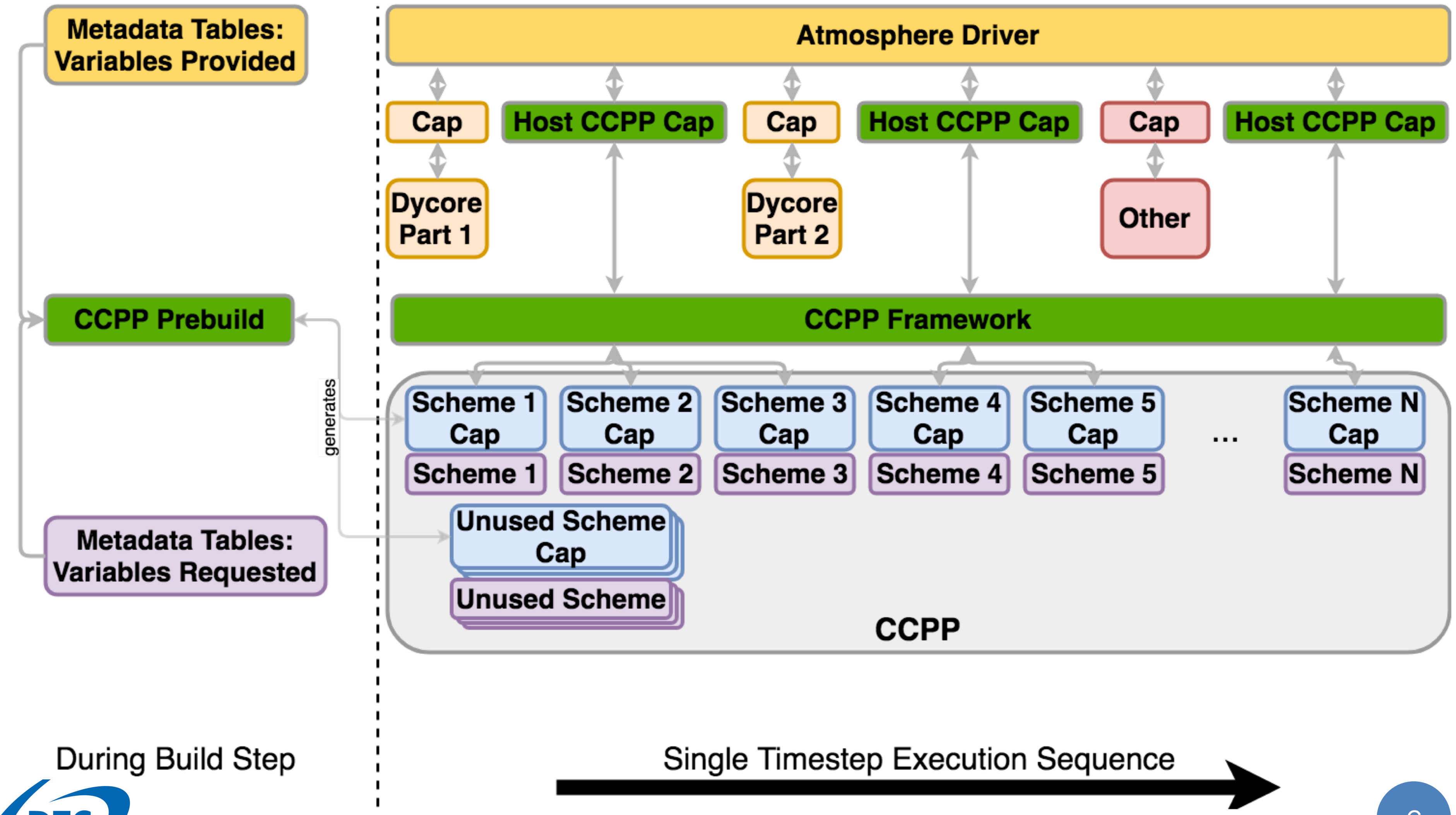


metadata tables: variables provided

CCPP prebuild

metadata tables: variables requested

CCPP prebuild in the dynamic build



Files generated for dynamic build

```
ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_FV3.tex
  # variables provided by host model and required by physics,
  # run ‘make’ in this dir to get PDF (also creates Dev. Guide)
ccpp/physics/CCPP_CAPS.{cmake,mk}
  # makefile snippets that contain all caps to be compiled
ccpp/physics/CCPP_SCHEMES.{cmake,mk}
  # makefile snippets that contain all schemes to be compiled
ccpp/physics/CCPP_VARIABLES_FV3.html
  # variables provided by host model
ccpp/physics/*_cap.F90
  # one cap per physics scheme
```

Files generated for dynamic build

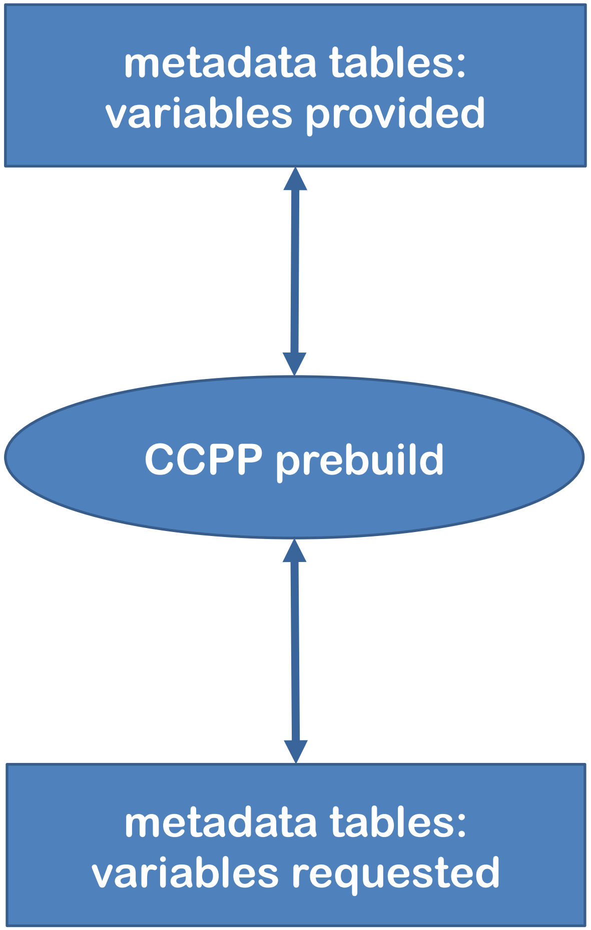
```
FV3/atmos_cubed_sphere/driver/fvGFS/ccpp_modules_fast_physics.inc
FV3/atmos_cubed_sphere/driver/fvGFS/ccpp_modules_slow_physics.inc
FV3/atmos_cubed_sphere/driver/fvGFS/ccpp_fields_fast_physics.inc
FV3/atmos_cubed_sphere/driver/fvGFS/ccpp_fields_slow_physics.inc

FV3/ipd/ccpp_modules_fast_physics.inc
FV3/ipd/ccpp_modules_slow_physics.inc
FV3/ipd/ccpp_fields_fast_physics.inc
FV3/ipd/ccpp_fields_slow_physics.inc

  # auto-generated code to include in host model caps (called
  # TARGET FILES) via CPP (preprocessor) directives:
  #   FV3/ipd/IPD_CCPP_driver.F90 for slow physics
  #   FV3/atmos_cubed_sphere/driver/fvGFS/atmosphere.F90
  #                               for fast physics
  # *.inc files 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
```

Developmental Testbed Center

CCPP prebuild in the static build

ccpp_prebuild.py

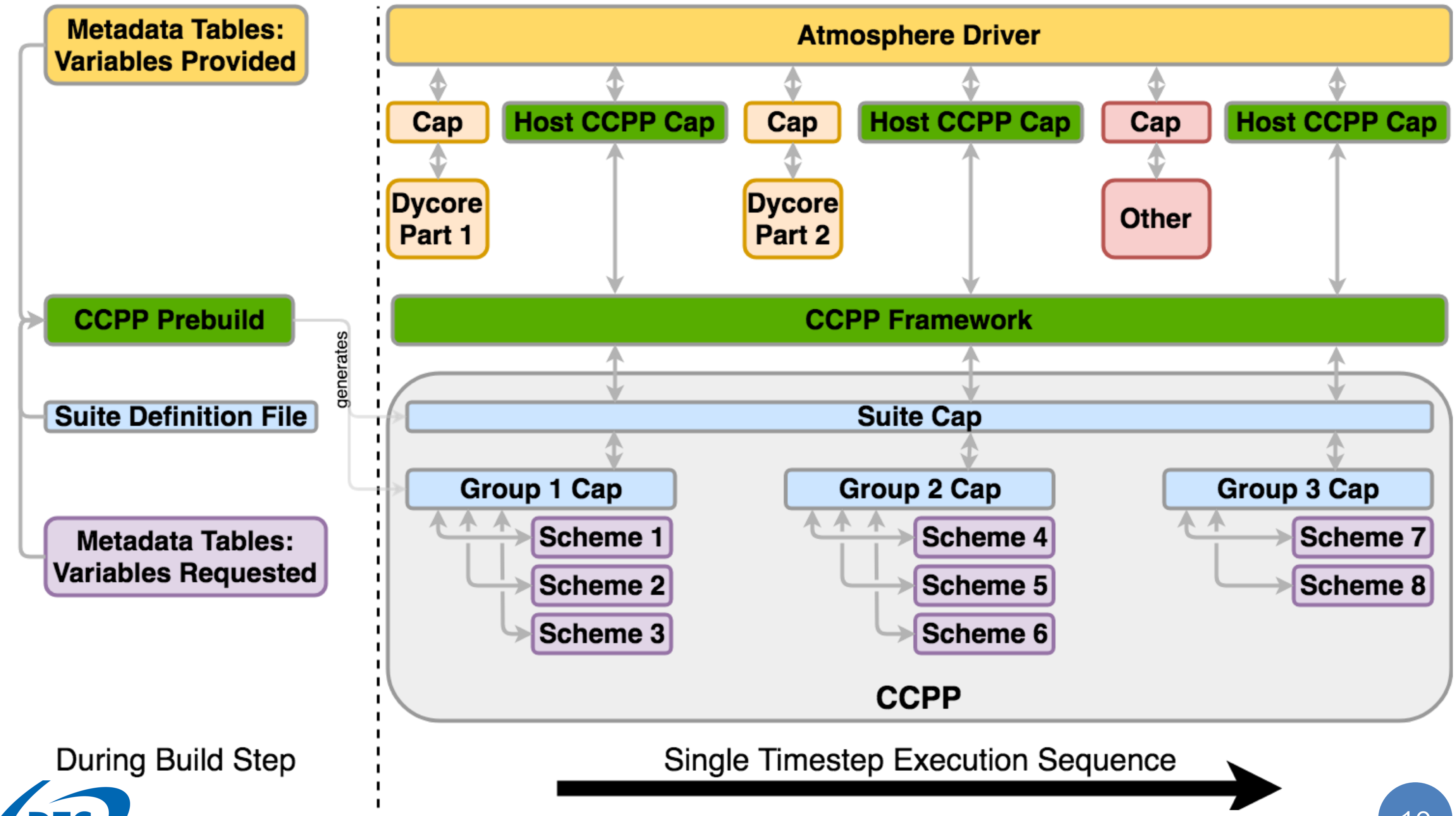
- requires metadata tables on both sides
- requires a suite definition file
- checks requested vs provided variables by standard_name
- checks units, rank, type (more to come)
- filters unused schemes and variables
- creates Fortran code (static API) that replaces the dynamic API (ccpp-framework)
- creates caps for physics groups and suite
- populates makefiles with schemes and caps


metadata tables: variables provided

CCPP prebuild

metadata tables: variables requested

CCPP prebuild in the static build

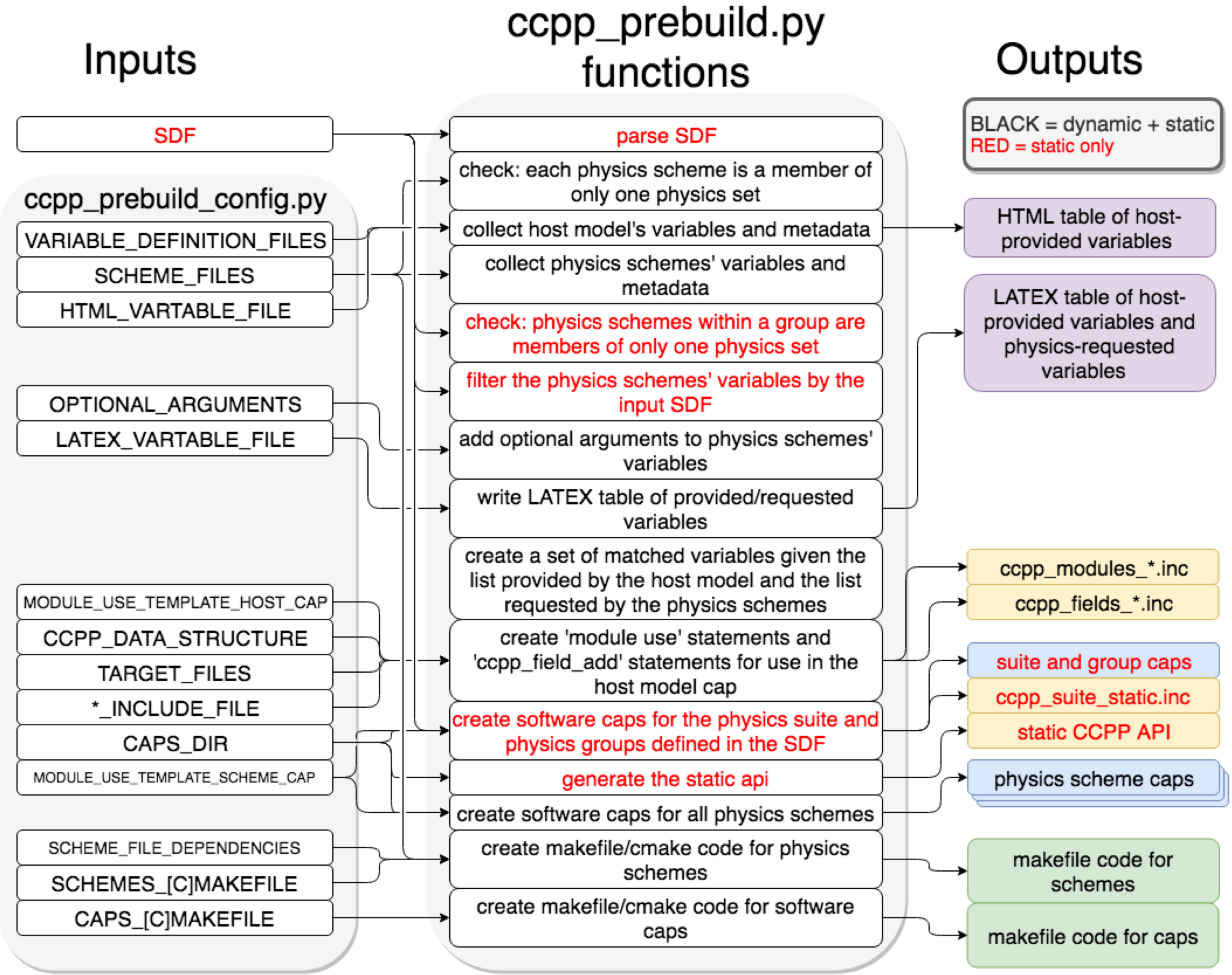


Files generated for static build

```
ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_FV3.tex
  # variables provided by host model and required by physics,
  # run 'make' in this dir to get PDF (also creates Dev. Guide)
ccpp/framework/src/ccpp_suite_static.inc
  # contains name of suite used at compile time, which
  # is compared to name of suite used at run time
ccpp/physics/CCPP_CAPS.{cmake, mk}
  # makefile snippets that contain all caps to be compiled
ccpp/physics/CCPP_SCHEMES.{cmake,mk}
  # makefile snippets that contain all schemes to be compiled
ccpp/physics/CCPP_VARIABLES_FV3.html
  # variables provided by host model
ccpp/physics/ccpp_group_*_cap.F90
  # one cap per physics group
ccpp/physics/ccpp_suite_cap.F90
  # cap for the entire suite
```

Files generated for static build

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

  # auto-generated API for static build that replaces
  # the dynamic API (aka ccpp-framework), the interface
  # is identical between the two APIs
  # TARGET FILES as before:
  #   FV3/ipd/IPD_CCPP_driver.F90 for slow physics
  #   FV3/atmos_cubed_sphere/driver/fvGFS/atmosphere.F90
  #                               for fast physics
```



Inputs

ccpp_prebuild.py functions

Outputs

SDF

ccpp_prebuild_config.py

VARIABLE_DEFINITION_FILES

SCHEME_FILES

HTML_VARTABLE_FILE

OPTIONAL_ARGUMENTS

LATEX_VARTABLE_FILE

MODULE_USE_TEMPLATE_HOST_CAP

CCPP_DATA_STRUCTURE

TARGET_FILES

*_INCLUDE_FILE

CAPS_DIR

MODULE_USE_TEMPLATE_SCHEME_CAP

SCHEME_FILE_DEPENDENCIES

SCHEMES_[C]MAKEFILE

CAPS_[C]MAKEFILE

parse SDF

check: each physics scheme is a member of only one physics set

collect host model's variables and metadata

collect physics schemes' variables and metadata

check: physics schemes within a group are members of only one physics set

filter the physics schemes' variables by the input SDF

add optional arguments to physics schemes' variables

write LATEX table of provided/requested variables

create a set of matched variables given the list provided by the host model and the list requested by the physics schemes

create 'module use' statements and 'ccpp_field_add' statements for use in the host model cap

create software caps for the physics suite and physics groups defined in the SDF

generate the static api

create software caps for all physics schemes

create makefile/cmake code for physics schemes

create makefile/cmake code for software caps

BLACK = dynamic + static
RED = static only

HTML table of host-provided variables

LATEX table of host-provided variables and physics-requested variables

ccpp_modules_*.inc

ccpp_fields_*.inc

suite and group caps

ccpp_suite_static.inc

static CCPP API

physics scheme caps

makefile code for schemes

makefile code for caps

Modifying CCPP prebuild config

CCPP prebuild is complex, but physics developers don’t need to fiddle with all the details (host model developers need to, but only once)!

What to change in `ccpp/config/ccpp_prebuild_config.py` when adding new physics or modifying existing physics:

- add new scheme (CCPP entry point) to `SCHEME_FILES` dictionary, choose correct set of physics (most likely `‘slow physics’`)
- add any dependencies to `SCHEME_FILES_DEPENDENCIES` list
- if optional arguments are used, add name of scheme and subroutine to `OPTIONAL_ARGUMENTS` and choose either `‘all’` or `‘none’` or provide a list of optional arguments to use (standard names)
- use existing entries and in-line documentation as guidance

Wrap up

- CCPP prebuild is the work horse of the CCPP
- needs to run before CCPP framework and physics are built
- is run automatically by NEMS build system
- does different things for dynamic and static builds
- single configuration file in Python format

```
ccpp/config/ccpp_prebuild_config.py
```

- physics developers need to change three options at most

```
SCHEME_FILES
SCHEME_FILES_DEPENDENCIES
OPTIONAL_ARGUMENTS
```

Behind the scenes

(extra credit: cdata in the dynamic and static build)

Extra credit: cdata in dyn/stat build

- cdata is a CCPP internal data type, defined in `ccpp/framework/src/ccpp_types.F90`
- cdata has five internal variables required by the static and dynamic build

```
integer           :: errflg
character(len=512) :: errmsg
  # for error handling: assign error message,
  # set errflg to /=0 and return from scheme
integer           :: loop_cnt
  # supports subcycling capability (default 1)
integer           :: blk_no
  # stores block number (def. 1 if no blocking)
integer           :: thrd_no
  # stores thread number (def. 1 if no threading)
```

Extra credit: cdata in dyn/stat build

- cdata is a CCPP internal data type, defined in
  `ccpp/framework/src/ccpp_types.F90`
- cdata has five internal variables required by the static and dynamic build
- CCPP physics calls are made with a scalar cdata (i.e. element of a cdata array if blocking and/or threading are used)

```
!$OMP parallel do
do nb = 1,nblks
   nt = omp_get_thread_num()+1
   call ccpp_physics_run(cdata_block(nb,nt), ierr=ierr)
   if (ierr/=0) ...
end do
```

➔ that's how CCPP knows about block/thread numbers

Extra credit: cdata in dyn/stat build

- cdata is a CCPP internal data type, defined in
  `ccpp/framework/src/ccpp_types.F90`
- cdata has five internal variables required by the static and dynamic build
- CCPP physics calls are made with a scalar cdata (i.e. element of a cdata array if blocking and/or threading are used) → CCPP knows blk/thrd no
- For the dynamic build, cdata also contains a lookup table that instructs CCPP where (in memory) to find a variable with a given standard_name

```
air_temperature :
   c_loc(GFS_Data(cdata%blk_no)%Statein%tgrs)
water_vapor_specific_humidity_at_lowest_model_layer :
   c_loc(GFS_Data(cdata%blk_no)%Statein%qgrs(:,1,1))
```

this information comes from host model metadata