**Interoperable Physics Driver for NGGPS**

I am confused about the entries on left column. Modules appears twice. First appearance, has underneath module\_nuopc\_physics. I think we do not want all variables to appear in left column, do we?

**Introduction**

The development of the Interoperable Physics Driver (IPD) is being funded by the Next Generation Global Prediction System (NGGPS) program as a means to facilitate the research, development, and transition to operations of innovations in atmospheric physical parameterizations. A prototype of this driver has been developed for the Global Forecast System (GFS) operational physics and works 'out of the box' coupled to the Global Spectral Model (GSM). This documentation covers the use of the IPD within GSM, as well existing infrastructure to develop a standalone version that runs a single physics timestep when ingesting data from the GSM model.

The development of the IPD is in conjunction with the development of the Common Community Physics Package (CCPP). The CCPP is intended to be a collection of parameterizations for use in Numerical Weather Prediction (NWP). These parameterizations are necessary to simulate the effects of processes that are either subgrid in scale (e.g., eddy structures in the planetary boundary layer), or are too complicated to be represented explicitly (e.g., microphysics and radiation).

Common categories of parameterizations include radiation, surface layer, planetary boundary layer and vertical mixing, deep and shallow cumulus, and microphysics. However, other categorizations are possible, and the CCPP will be designed to accommodate schemes that span multiple categories, such as the Simplified Higher Order Closure parameterization (SHOC). The parameterizations can be grouped together into "physics suites" (defined here: [**Definition of a Physics Suite**](index.html#mainpage-suite)), which are sets of parameterizations known to work well together. Indeed, accurately representing the feedbacks and interactions between the physical processes represented by the parameterizations is essential.

The CCPP will be designed to be *model–agnostic* in the sense that parameterizations contained in the package receive inputs from the dynamic core through the IPD. A pre/post physics layer converts variables between those used in the dynamic core and those required by the Driver, and performs any necessary de- and re-staggering.

A single driver will be used to invoke any set of parameterizations or suite within the CCPP. Each suite is accompanied by a pre/post parameterization interface, which converts variables between those provided by the driver and those required by the parameterization, in case they differ. Through this mechanism, the CCPP and driver provide physical tendencies back to the dynamic core, which is in turn responsible for updating the state variables. The driver and CCPP can also provide variables for diagnostic output, or for use in other Earth System models.

The IPD is still in a prototype phase at NCEP and, as such, is implemented in a way specific to the GFS physics. The figure below is an overview diagram of how the IPD is called in the GFS system.

**Driver Implementation**

**Requirements for the IPD**

The IPD is expected to interact with any set of physics and any dynamic core, thus several requirements are needed to meet those requirements. Because of its purpose as a community tool to promote research with operational models and foster transition of research to operations, it is imperative that requirements also be placed on the physics parameterizations.

These requirements are stated explicitly here:

[Interoperable Physics Driver and Common Community Physics Package (CCPP): Goals and Requirements](https://docs.google.com/document/d/1O6ii0r0jX83pD9Sa4FxnAC5d2lAU55yg_zvHaVykeLw/edit#heading=h.qmhn7sgvdtt4)

**Definition of a Physics Suite**

It is important that the IPD able to support a **physics suite** as an identifiably distinguishable entity from an arbitrary group of physical parameterizations. The distinction between **physical parameterization** and **physics suite** is made as follows.

A **physical parameterization** is a code that represents one or more physical processes that force or close model dynamics. It is defined by the code implementation of the mathematical functions comprising the scheme, and not by a particular set of parameters or coefficients that could be set externally.

A **physics suite** is a set of non-redundant atmospheric physical parameterizations that have been designed or modified to work together to meet the forcing and closure requirements of a dynamical core used for a weather or climate prediction application. A set of physical parameterizations chosen to be identified as a suite results from the needs and judgments of a particular user, developer, or group of either.

In some cases, a suite may be identified as a benchmark or reference set of physical parameterizations, against which variations can be tested. Since a suite can be configured in different ways for different applications by modifying its tunable parameters, an accompanying set of tunable parameters should be specified when defining a reference implementation or configuration of a physics suite.

In the context of NGGPS, a Physics Review Committee will be established to determine which physical parameterizations should be accepted onto the Common Community Physics Package, and which physics suites should be identified as such.

An **ensemble physics suite** is a collection of physics suites as defined above, and may be implemented as part of multi-physics ensemble forecast system.

**The IPD Prototype**

Much of the code utilizes modern Fortran standards up to F2003 and should be compatible with current Fortran compiler implementations. Model data is encapsulated into several Derived Data Types (DDT) with Type Bound Procedures. Most of the model arguments are pointers into the actual arrays that are allocated and are by default managed externally to the driver. The DDTs serve as containers of the passed arguments and several DDTs exist to provide structure and organization to the data. One goal and constraint of this development was to minimize changes to existing code.

The GFS currently calls eleven physics schemes as a part of its physics suite. In doing so, it passes many atmospheric variables between the dynamic core and the physics modules using an initialization procedure. The list of variable had become an unruly with over a hundred variables. Through the use of the DDTs, the list was reduced to a more succinct set of required variables (on the order of 10) to be used by all of the physics modules in their interaction with the atmospheric model.

The DDTs are defined based on the following template:

type model\_data\_in

private

real :: vara

real :: varb

end type

type model\_data

private

type (model\_data\_in) :: data\_in

type (model\_data\_out) :: data\_out

type (model\_data\_inout) :: data\_inout

contains

procedure :: setin => set\_model\_in

procedure :: setout => set\_model\_out

procedure :: setinout => set\_model\_inout

end type

The currently defined DDTs are as follows:

* [**nuopc\_physics::tbd\_ddt**](structnuopc__physics_1_1tbd__ddt.html) : arguments that still need to be categorized.
* [**nuopc\_physics::state\_fields\_in**](structnuopc__physics_1_1state__fields__in.html) : input states for physics
* [**nuopc\_physics::state\_fields\_out**](structnuopc__physics_1_1state__fields__out.html) : output states for physics
* [**nuopc\_physics::sfc\_properties**](structnuopc__physics_1_1sfc__properties.html) : surface properties
* [**nuopc\_physics::diagnostics**](structnuopc__physics_1_1diagnostics.html) : diagnostic fluxes and other fields
* [**nuopc\_physics::interface\_fields**](structnuopc__physics_1_1interface__fields.html) : fields used for coupling to land/ocean
* [**nuopc\_physics::cloud\_properties**](structnuopc__physics_1_1cloud__properties.html) : cloud related fields
* [**nuopc\_physics::radiation\_tendencies**](structnuopc__physics_1_1radiation__tendencies.html) : radiation fields
* [**nuopc\_physics::dynamic\_parameters**](structnuopc__physics_1_1dynamic__parameters.html) : model parameters that change (used to be part of model\_parameters but these change frequently)
* [**nuopc\_physics::model\_parameters**](structnuopc__physics_1_1model__parameters.html) : paramters that are set once in the initialize phase

**Physics Driver calling sequence**

**Initialize Phase**

In the GSM, module gfs\_physics\_initialize\_mod calls:

* nuopc\_phys\_init (module [**nuopc\_physics**](namespacenuopc__physics.html)) to initialize parameters used in the radiation and other physics parameterizations.
  + Populate idat, used by radupdate, with values from idate (NCEP's absolute date and time of initial conditions).
  + Populate the model\_parameters container (mdl) with the input arguments.
  + Call gfuncphys (module funcphys) to compute all physics function tables.
  + Call rad\_initialize (module rad\_initialize) to initialize fixed control variables for radiation processes.
  + Call set\_soilveg (module set\_soilveg) to initialize soil parameters.

**Run phase**

The current run implementation of GSM code divides the physics calls into two stages; the first call to invoke the radiation physics (gloopr), and a second call to invoke the remaining physics. The GSM makes calls to gloopr and gloopb in turn invoke the physics driver through use of containers and methods provided by [**nuopc\_physics**](namespacenuopc__physics.html).

* gloopr
  + Populate the DDT containers with the data to be sent to the radiation physics call of the physics driver.
    - dyn\_parm%setrad: set the dynamic\_parameters
    - state\_fldin%setrad: set the state\_fields\_in
    - sfc\_prop%setrad: set the sfc\_properties
    - diags%setrad: set the diagnostics
    - cld\_prop%setrad: set the cloud\_properties,
    - rad\_tend%set: set the radiation\_tendencies
    - intrfc\_fld % setrad: set the interface\_fields
  + Invoke the method nuopc\_rad\_run from module [**nuopc\_physics**](namespacenuopc__physics.html) to advance the radiation physics a single step. The method calls grrad with pointers to the containers specifying the call's argument list.

**gloopr calls to IPD**

* gloopb
  + Populate the DDT containers with the data to be sent to the non-radiation physics call of the physics driver.
    - dyn\_parm%setphys: set the dynamic\_parameters
    - state\_fldin%setphys: set the state\_fields\_in
    - diags%setphys: set the diagnostics
    - intrfc\_fld%setphys: set the interface\_fields
    - rad\_tend%set: set the radiation\_tendencies
    - sfc\_propt%setphys: set the sfc\_properties
    - cld\_prop%setphys: set the cloud\_properties
    - tbddata%set: set the tbd\_ddt
  + Invoke the method nuopc\_phys\_run from module [**nuopc\_physics**](namespacenuopc__physics.html) to advance the non-radiation physics a single step. The method calls gbphys with pointers to the containers specifying the call's argument list.

**gloopb calls to IPD**

**Getting the code**

This page explains how to checkout the code and what is included.

**Checking out the code**

The source code is available by combining two branches of the EMC repository, one for NEMS and one for GFS. The code can be checked out by typing the following lines:

* svn checkout <https://svnemc.ncep.noaa.gov/projects/nems/branches/PhysDrvI>
* cd PhysDrvI/src/atmos
* svn checkout <https://svnemc.ncep.noaa.gov/projects/gsm/branches/PhysDrvI> gsm

**Directory Structure**

Once you have obtained the code, you will have the following directory structure within the PhysDrvI directory (only relevant directories and files are listed here).

* src/atmos/phys/
  + [**nuopc\_physics.F90**](nuopc__physics_8_f90.html) ... physics driver, DDTs, wrapper subroutines
  + [**grrad.f**](grrad_8f.html) ... radiation subroutine
  + [**gbphys.f**](gbphys_8f.html) ... non-radiation physics subroutine
* src/atmos/gsm/phys
  + gfs\_physics\_iniialize\_mod.f ... calls [**nuopc\_phys\_init()**](group__module__nuopc__physics.html#ga67cf9d4433fe27cc3437c800d70196ca)
  + gfs\_physics\_run\_mod.f ... calls do\_physics\_one\_step, passes mdl\_param from init
  + do\_physics\_onestep.f ... cals gloopr and gloopb, passes mdl\_param
  + [**gloopr.f**](gloopr_8f.html) ... fills the DDT containers and calls nuopc\_rad\_run
  + [**gloopb.f**](gloopb_8f.html) ... fills the DDT containers and calls nuopc\_phys\_run
  + gfs\_physics\_internal\_state\_mod.f ... defines the gfs physics internal state

**Using the standalone functionality**

This page explains how one might use the existing driver capabilities to run in a test mode that does not require running the dynamic core.

**Why a standalone function?**

In a testing environment, it can be useful to extract physics input data from the dynamic core output, run a standalone driver, and then verify that the results produced using the driver are identical to the results produced by the full model.

In addition to outline how one might use a standalone feature, understanding the steps required to run the driver in a simplified framework may demystify the functionality of the driver in the full system.

**Note**

Setting up the standalone wrapper is an exercise left to the user.

**General algorithm for building a stand alone wrapper**

A Fortran wrapper to call the standalone capability would need the following steps:

1. The following are required external modules:
2. use nuopc\_physics,
3. only: state\_fields\_in, state\_fields\_out, sfc\_properties,
4. diagnostics,
5. interface\_fields, cloud\_properties, radiation\_tendencies,
6. model\_parameters,phys\_init\_readin,nuopc\_phys\_init,
7. rad\_run\_readin,nuopc\_rad\_update,nuopc\_rad\_run,rad\_run\_saveout,
8. phys\_run\_readin, nuopc\_phys\_run,phys\_run\_saveout,
9. tbd\_ddt, use\_nuopc, dynamic\_parameters
10. use module\_radsw\_parameters, only : topfsw\_type, sfcfsw\_type
11. use module\_radlw\_parameters, only : topflw\_type, sfcflw\_type
12. use funcphys
13. use namelist\_soilveg
14. use physcons, only : dxmax, dxmin, dxinv
15. Declare all variables (see [**gloopb.f**](gloopb_8f.html) and [**gloopr.f**](gloopr_8f.html))
16. Read the physics initialization field from a file (nuopc\_physics::nuopc\_phys\_init\_readin())
17. Call gfuncphys to initialize module funcphys
18. Initialize physics tables, radiation variables, and soil-related parameters (nuopc\_phys\_init)
19. Call the setrad procedures to assign the targets to pointers for the following DDTs:
    * dyn\_parm
    * state\_fldin
    * sfc\_prop
    * diags
    * cld\_prop
    * rad\_tend
    * intrfc\_fld
20. Read in those necessary inputs to run nuops\_rad (rad\_run\_readin)
21. Get external dataset to run nuopc\_rad (nuopc\_rad\_update)
22. Run the radiation (nuopc\_rad\_run, a wrapper that calls grrad)
23. Save the radiation step output to radrun\_saveout.dat (rad\_run\_saveout)
24. Call the setphys procedures to assign targets to pointers for the following DDTs:
    * dyn\_parm
    * state\_fldin
    * state\_fldout
    * diags
    * intrfc\_fld
    * rad\_tend
    * sfc\_prop
    * cld\_prop
    * tbddata
25. Read in the necessary inputs to run physics (phys\_run\_readin)
26. Run the physics (nuopc\_phys\_run)
27. Save the physics output to physrun\_saveout.dat (phys\_run\_saveout)

Before running such a wrapper, solar, aerosol, and co2 data must be linked to the run directory. Solar constant data must be linked to fort.102. These files include several files:

* aerosol.dat
* co2historicaldata\_[YYYY].txt
* co2historicaldata\_glob.txt
* co2monthlycyc.txt
* solarconstant\_noaa\_a0.txt

Where YYYY is a four-digit year. These files are available ????

**Build and run**

Basic requirements to build and run the standalone wrapper outlined above:

1. Compile
   * The make.sh loads the configuration file and load required modules. (Zeus)

An example make.sh file:

#!/bin/sh

set -ax

curdir=`pwd`

# Copy machine specific config to generic name, used by physics makefile

# NEMS\_loc is the installation location of the NEMS code. NOAA's Zues is used in this example

cd ${NEMS\_loc}/src

configure 6\_gsm\_zeus

cd $curdir

# Load the modules needed for this config

source ${NEMS\_loc}/src/conf/modules.nems

# Build src/atmos/share, src/atmos/phys, and standalone test driver

gmake

* + A makefile should build the src/atmos/share/libshare.a and src/atmos/phys/libphys.a

An example makefile:

# include ../src/conf/configure.nems

#

# MAKEFILE = makefile

#

# Zeus config: This is hardcoded for clarity using the config for zeus. See ../src/conf for other configurations/systems

F90 = ifort -g -align array32byte -convert big\_endian -assume byterecl

FREE = -free

TRAPS = -g -traceback

FFLAGS = $(TRAPS) -fp-model strict

OPTS\_GFS = -O0

FFLAGS\_GFS = $(OPTS\_GFS) $(FFLAGS) $(FREE)

R8 = -r8

OPTS = $(FFLAGS\_GFS) $(R8)

FFLAG90 = $(OPTS) $(FREE)

TARGET = standalone\_wrapper.x

INCS = -I../src/atmos/share \

-I../src/atmos/phys

MAIN = standalone\_wrapper.o

LIBS = ../src/atmos/share/libshare.a \

../src/atmos/phys/libphys.a

LIBDIR=/contrib/nceplibs/nwprod/lib/

W3\_LIB = -L${LIBDIR} -lw3nco\_d -lw3emc\_d

$(TARGET): libs $(MAIN)

$(F90) -o $(TARGET) $(MAIN) $(LIBS) $(W3\_LIB)

libs:

cd ../src/atmos/share && gmake -f makefile.phys

cd ../src/atmos/phys && gmake -f makefile.phys

$(MAIN): %.o: %.f90

$(F90) $(FFLAG90) $(INCS) -c standalone\_wrapper.f90

clean:

cd ../src/atmos/share && gmake clean

cd ../src/atmos/phys && gmake clean

rm -f \*.o \*.x

1. Run
   * Input files
     + init\_savein.dat: initial setting for some model parameters and physics options
     + radrun\_savein.dat: initialization for radiation physics
     + physrun\_savein.dat: initialization for other physics
   * Input constants files are located in the data directory.
     + solarconstant\_noaa\_a0.txt: solar constant with cycle approx (link to unit 102)
     + aerosol.dat: climatological aerosol global distribution
     + co2monthlycyc.txt and co2historicaldata\_yyyy.txt: co2 data
   * Output files
     + radrun\_saveout.dat: output from radiation run
     + physrun\_saveout.dat: output from physics run
   * Run
     + Setup the run directory by linking in required data and constants files
     + Run the fortran wrapper for a single time-step iteration

An example run script:

#!/bin/csh

#clear the old outputs

/bin/rm \*saveout.dat

# copy the solar aerosol and co2 data

cp data/\* .

#link solar constant file

ln -s solarconstant\_noaa\_a0.txt fort.102

#run

standalone\_wrapper.x

#delete solar aerosol and co2 data

/bin/rm solarconstant\_noaa\_a0.txt aerosol.dat co2\* fort.102

**Note**

All three savein files in this version were obtained by running GSM with the IPD configured to use lon=65, lan=8, and kdt=1 Running with these values provided a set of eight non-contiguous grid columns that included land and sea points. The saved inputs and outputs were generated using -O0 (no optimization). Results using -O3 are not the same as -O0.

[**Todo:**](todo.html#_todo000001)

Figure out where data is and fix the docs

Figure out if GMTB is providing canned data for a standalone situation. Fix docs.

**Note**

Output from the GSM run are included for comparison. radrun\_saveout.dat.gsm physrun\_saveout.dat.gsm

**Compare output**

To perform a bit-wise comparison of the output from the driver run in standalone mode and the full model:

cmp radrun\_saveout.dat radrun\_saveout.dat.gsm > cmp\_radrunout

cmp physrun\_saveout.dat physrun\_saveout.dat.gsm > cmp\_phyrunout

**Canned standalone input**

Input files for use with the standalone capability are available on Theia, here: ????

To generate your own input data from a model run of choice, run the full model with the IPD enabled.

**Modules**

Here is a list of all modules:

|  |  |
| --- | --- |
| **[module\_nuopc\_physics](group__module__nuopc__physics.html" \t "_self)** | IPD, or the interface between the dynamic core and the physics parameterizations |
| **[gbphys](group__gbphys.html" \t "_self)** | Driver that invokes GFS physics |
| **[module\_radiation\_driver](group__module__radiation__driver.html" \t "_self)** | Radiation driver module, which prepares atmospheric profiles and invokes main radiation calculations |

**gbphys**

Driver that invokes GFS AM physics. [More...](#details)

|  |  |
| --- | --- |
| Functions/Subroutines | |
| subroutine | [**gbphys**](group__gbphys.html#ga34cd2db09b580c23df51c96d5905a805) |
|  | Parameter descriptions include intent, name, description, and size. [More...](#ga34cd2db09b580c23df51c96d5905a805) |
|  | |

Detailed Description

Driver that invokes GFS AM physics.

Function/Subroutine Documentation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **subroutine gbphys** | **(** |  | **)** |  |

Parameter descriptions include intent, name, description, and size.

gbphys is the driver subroutine to invoke GFS physics (except radiation but radiative heating is applied here) at physics time steps

**Parameters**

|  |  |  |
| --- | --- | --- |
| [in] | **ix,im** | - integer, horiz dimension and num of used pts 1 |
| Detailed Description  Sets commonly used control variables for radiation. See Detailed Description for sections.  Sets commonly used control variables for radiation.  Section 1 contains control variables defined in the form of parameter. They are pre-determined choices and not adjustable during model's run-time.   * 1.1 SW Radiation control flags * 1.2 LW Radiation control flags * 1.3 LW Aerosol properties   Section 2 contains control variables defined as module variables. They are more flexible to be changed during run-time by either through input namelist, or through model environment condition. They are pre-assigned here as the default values.   * 2.1 Control flags for module radiation\_astronomy * 2.2 Control flags for module radiation\_aerosols * 2.3 Control flags for module radiation\_gases * 2.4 Control flags for module radiation\_clouds * 2.5 Control flags for module radiation\_surface * 2.6 General purpose control flags |  |  |