National Unified Operational Prediction Capability

Building a NUOPC Model

ESMF 8.5.0 beta snapshot

Content Standards Committee (CSC) Members

December 19, 2022

NUOPC
CMA/CSC Committee
http://www.weather.gov/nuopc

Contents

1 Overview **2**
- 1.1 Document Roadmap . . . 2
- 1.2 Additional NUOPC Resources . . . 2

2 The Big Idea **4**
- 2.1 Specializing Generic Components . . . 4
- 2.2 NUOPC Model Cap . . . 4
- 2.3 How Much of My Code Do I Need to Change? . . . 4
- 2.4 How Do I Know it Works? . . . 5

3 Writing and Testing a NUOPC Cap for your Model **6**
- 3.1 Install ESMF and NUOPC on the Target Machine . . . 6
- 3.2 Prepare Your Model Code . . . 7
- 3.3 Choose a Configuration of Your Model for Development . . . 7
- 3.4 Integrate a Cap Template into Your Codebase . . . 8
- 3.5 Modify Your Build to Generate a NUOPC Makefile Fragment . . . 8
- 3.6 Initialize Your Model from the Cap . . . 10
- 3.7 Call Your Model's Run Subroutine from the Cap . . . 10
- 3.8 Run the Cap with a NUOPC Driver . . . 10
- 3.9 Split Up the Initialization Phases . . . 11
- 3.10 Test and Validate Your Cap . . . 11
- 3.11 Example NUOPC Model cap . . . 11

4 An Example Cap **14**
- 4.1 Module Imports . . . 14
- 4.2 SetServices . . . 14
- 4.3 Checking Return Codes . . . 16
- 4.4 Initialize Phase - Advertise Fields . . . 16
- 4.5 Initialize Phase - Realize Fields . . . 17
- 4.6 Model Advance Specialization . . . 19

1 Overview

The National Unified Operational Prediction Capability (NUOPC) is a strategic initiative to fundamentally advance the nation's computational weather prediction systems and improve forecast models used by National Weather Service, Air Force and Navy meteorologists, mission planners, and decision makers. The NUOPC Layer is a software layer built on top of the Earth System Modeling Framework (ESMF). ESMF is a high-performance modeling framework that provides data structures, interfaces, and operations suited for building coupled models from a set of components. **NUOPC refines the capabilities of ESMF by providing a more precise definition of what it means for a model to be a component and how components should interact and share data in a coupled system.** The NUOPC Layer software is designed to work with typical high-performance models in the Earth sciences domain, most of which are written in Fortran and are based on a distributed memory model of parallelism (MPI).

The NUOPC Layer implements a set of *generic component*s that serve as building blocks that can be assembled together in different ways to build up a coupled modeling application. In some cases, a generic component can be used as is, and in other cases the generic component must be *specialized* (customized) for a particular model or application. Additionally, the NUOPC Layer defines a set of technical rules for how components should behave and interact with each other. These technical rules form the backbone of component interoperability. NUOPC defines this effective interoperability as the ability of a model component to execute without code changes in a driver that provides the fields that it requires, and to return with informative messages if its input requirements are not met. A component that follows the NUOPC Layer technical rules is considered to be NUOPC Layer compliant.

For brevity, throughout this document we will often use the term "NUOPC" to refer to the "NUOPC Layer software" that is the current technical implementation of the NUOPC specification. Also, the term "NUOPC component" is shorthand for a component that is NUOPC Layer compliant and can be used in NUOPC-based systems.

1.1 Document Roadmap

This document is a starting point for model developers and technical managers who are new to the NUOPC Layer software and need to understand the steps involved in making an existing model codebase NUOPC Layer compliant.

The document is divided into the following sections:

- Section 2 describes important parts of the NUOPC design that are critical for anyone planning to write code using the NUOPC API.
- Section 3 describes the development steps involved in making your model code NUOPC Layer compliant.
- Section 4 presents the code of a basic example cap, describing each part in detail.

1.2 Additional NUOPC Resources

This document is not exhaustive, but should help you navigate the process of creating a NUOPC component from your model. As such this document is a companion to other NUOPC resources available:

- The NUOPC website is the main source of information on NUOPC, including instructions for acquiring and using the NUOPC Layer software.
- The NUOPC Reference Manual is the primary technical reference for the NUOPC API and includes a detailed description of the NUOPC generic components.

- The NUOPC Prototype Codes page and Git repository include a set of prototype applications that use the NUOPC Layer software. These applications are architectural skeletons that represent typical configurations of NUOPC components and provide numerous examples of using the NUOPC API.
- Cupid is a plugin for the Eclipse Integrated Development Environment that automatically generates NUOPC Layer compliant code and checks existing source code for compliance.
- A BAMS article entitled The Earth System Prediction Suite: Toward a Coordinated U.S. Modeling Capability describes NUOPC and how NUOPC Layer compliant components are being used in coupled modeling systems across U.S. agencies.

2 The Big Idea

This section should help you understand key aspects of the NUOPC Layer design that are critical for writing the code to make your model NUOPC Layer compliant. The NUOPC Layer includes four kinds of generic components, each with a different purpose in a coupled application. One kind of generic component is the *NUOPC Model*, a component that wraps a model code (such as an atmosphere, ocean, or ice model) such that it exposes the set of interfaces defined by the NUOPC specification. You will work primarily with the NUOPC Model generic component in order to make your model NUOPC Layer compliant.

This documentation focuses primarily on the NUOPC **Model** Component. However, you should be aware that there are four kinds of generic components implemented in the NUOPC Layer:

Model Wraps a model code, such as an atmosphere, ocean, or ice model

Connector Handles standard data transformations (e.g., redistribution or regridding) between two components in a single direction

Mediator Contains custom coupling code (e.g., flux calculations, averaging) between Models; unlike the Connector, a Mediator can handle data from multiple Models with data flowing in multiple ways

Driver Coordinates execution of Models, Mediators, and Connectors

2.1 Specializing Generic Components

A key design idea behind NUOPC is that a lot of code (and therefore behavior) is provided for you. This code is provided via the four generic components included with the NUOPC library, plus some additional utility routines. The *NUOPC Model* generic component implements most of the initialization and run behavior for you, but you have to supply some key parts of the implementation that are specific to your model. **The process of supplying your custom code that completes the generic NUOPC Model component is called specialization.** In other words, you are specializing the generic component to work for your particular model. Any parts of the code that you do not specialize are *inherited* from the generic component.

Those familiar with object-oriented programming will recognize the ideas of specialization and inheritance. Since the NUOPC Layer is written in Fortran 90, which has limited support for object-oriented programming, your specialization code is provided in Fortran subroutines which are registered with NUOPC using function pointers. NUOPC makes callbacks into your code when required to execute the specialization code.

2.2 NUOPC Model Cap

A *NUOPC Model cap* is a Fortran module that contains your code that specializes the generic NUOPC Model component for your particular model. The NUOPC Model cap serves as the interface to your model when it's used in NUOPC-based systems. **The term "cap" is used because it is a small software layer that sits on top of your model, making calls into it.** Typically, your model code will not make calls back into the cap. Sometimes we say just "cap" or "NUOPC cap" because it's quicker than saying "NUOPC Model cap."

2.3 How Much of My Code Do I Need to Change?

The amount of code that your need to change depends on how your model is structured and the degree to which it is already an independent component. The NUOPC cap itself does not usually require changes to your model's internals. Instead, the cap primarily acts as a separate software layer, and your model otherwise operates in its usual way.

However, as detailed in the section 3.2, if your model is currently embedded as a subsystem in a larger application and cannot be built independently, you must first take steps to modularize the code and remove dependencies to other models before beginning the NUOPC implementation.

The creation of a NUOPC cap does not mean that your model must always be run as a NUOPC component. Existing models can retain their native modes of operation, and running your model in NUOPC mode becomes a configuration option.

The NUOPC cap becomes a new locus of control for your model when your model is run in NUOPC mode. In other words, it will make calls into your model code to initialize your model and step it forward in time. One result of this is that the very top level main program of your model may not be used at all when your model is run in NUOPC mode. This is because all models participating in a coupled NUOPC application will be controlled by a separate generic component: the NUOPC Driver.

Putting control into a separate driver enables synchronization of all models participating in a coupled application, allows NUOPC to control when each model component runs (and for how long), and allows NUOPC to intercept and inject variables produced and required by your model at key parts during execution. Once you have a working NUOPC cap (you only need to implement it once), you have an interoperable component that can be used in systems with other NUOPC components.

2.4 How Do I Know it Works?

Validating your NUOPC cap is extremely important. The idea is to ensure that your model's current behavior is reproduced exactly as before, but this time with control flowing from the cap. This is why we encourage you to generate some baseline output by running your model in its "normal" way before doing any implementation. You will validate your cap by ensuring that when it controls your model, the same output is reproduced. In most cases the output matches bit-for-bit so a simple file-based comparison will be sufficient.

We also provide tools to help you check whether your cap is NUOPC-compliant. **NUOPC Compliance can be evaluated using a combination of two tools, the Component Explorer and the Compliance Checker, included in the ESMF/NUOPC software distribution.** More information is provided in sections 3.8 and 3.10.

3 Writing and Testing a NUOPC Cap for your Model

While there is no one right way to write the *NUOPC Model cap* code, the following recommended steps represent an incremental approach to developing the cap.

1. Install ESMF and NUOPC on the Target Machine (3.1)
2. Prepare Your Model Code (3.2)
3. Choose a Configuration of Your Model for Development (3.3)
4. Integrate a Cap Template into Your Codebase (3.4)
5. Modify Your Build to Generate a NUOPC Makefile Fragment (3.5)
6. Initialize Your Model from the Cap (3.6)
7. Call Your Model's Run Subroutine from the Cap (3.7)
8. Run the Cap with a NUOPC Driver (3.8)
9. Split Up the Initialization Phases (3.9)
10. Test and Validate Your Cap (3.10)

3.1 Install ESMF and NUOPC on the Target Machine

First, you need to ensure the prerequisite software is available on the target system.

The primary prerequisite software is the NUOPC library, which is included with the ESMF distribution, and your model, including any of its dependencies.

Acquire the latest ESMF release from GitHub:

```
$ git clone https://github.com/esmf-org/esmf.git --branch main --depth 1
```

Compile and install ESMF. Full installation details can be found in the ESMF User Guide. An example of the basic procedure for one particular system is outlined below.

```
# set environment variables for build
# the actual settings depend on your platform
# and the compilation options you select
$ export ESMF_DIR=/path/to/esmf
$ export ESMF_COMPILER=gfortran
$ export ESMF_COMM=openmpi
$ export ESMF_PIO=internal
$ export ESMF_NETCDF=split
$ export ESMF_NETCDF_INCLUDE=/usr/include
$ export ESMF_NETCDF_LIBS="-lnetcdff -lnetcdf"
$ export ESMF_NETCDF_LIBPATH=/usr/lib
$ export ESMF_INSTALL_PREFIX=/path/to/install
```

```
# build
$ cd /path/to/esmf
$ gmake
$ gmake check
$ gmake install
```

3.2 Prepare Your Model Code

Before starting a NUOPC cap implementation, your model must already be modularized such that it can be built by itself and does not contain hard dependencies to other model components. For example, if the model targeted for NUOPC compliance is a subsystem embedded in a larger application, the model will first need to be extracted such that it can be built by itself as a library.

The model also needs to be roughly divided into several execution methods: initialize, run, and finalize. Each of these methods may contain several phases. The run method should allow the model to execute a single timestep, or accept a parameter defining the number of timesteps or a "run until" time.

Your NUOPC cap code will be cleanest if your model exposes data structures for input and output variables with clear, well-documented naming conventions. This will simplify the process of hooking up fields in the NUOPC cap to your model's data structures. The NUOPC Field Dictionary uses the Climate and Forecast conventions for defining field standard names, but can support field name aliases.

Finally, the model should not use the global MPI_COMM_WORLD communicator explicitly, but should accept a communicator at some point during startup. A global search and replace can be used to replace all uses of MPI_COMM_WORLD with a different communicator defined as a global variable in your model.

3.3 Choose a Configuration of Your Model for Development

When implementing the NUOPC interfaces for your model, you want to get into an efficient edit-compile-debug cycle. This will require running a configuration of your model that can be used to test the NUOPC code you will write.

You should choose a configuration of your model that is simple and stable. Many models have regression test configurations that can be run quickly and have small output files. These configurations are typically low resolution, have short execution times, and sometimes have idealized initial conditions. Some models can also be configured with some of the physics options turned off to reduce the total amount of computation. More scientifically interesting or higher resolution configurations can be used after ensuring that the NUOPC cap is working for the basic case.

Compile your model on the target system and generate baseline output for the selected configuration. This will typically be a small set of history or restart files. We'll use these files later to ensure that your model is reproducing the expected output when executed through the NUOPC cap. In most cases, when your model is executed through its NUOPC cap, the output should be bit-for-bit identical with non-NUOPC runs. (The one caveat to this is that when your model is used in a coupled system, roundoff error may occur due to slight differences introduced when grid interpolation is used between models.)

If your model is already using ESMF, **you will need to update your build to link against ESMF version 7 or later.** Instructions for checking out this version of ESMF appear in section 3.1.

3.4 Integrate a Cap Template into Your Codebase

An important question is where you will put your NUOPC cap source code. The NUOPC cap code added to a model is minimal and is typically contained either in a single source file or a small set of files. **We recommend including the NUOPC cap code in the same code repository with the rest of your model code as this helps to ensure the cap evolves with your code and simplifies the process of acquiring a NUOPC-compliant version of your model.** The exact right place to put the cap code is your decision and largely depends on your model's directory structure.

Including the cap code in your model's codebase does *not* imply that your model must always be run in NUOPC mode. Instead, when the cap is complete, the NUOPC mode can be viewed as a configuration option of your model.

You need not start from scratch. Instead start with a NUOPC cap template. To acquire a cap template you can:

- use the cap template below,
- copy code from one of the NUOPC Prototype Applications or
- use the Cupid plugin for Eclipse to generate code. Cupid automatically generates NUOPC compliant code fragments for specialization points and presents NUOPC API reference documentation based on where you are in your NUOPC cap code. Installation instructions are provided on the Cupid website, and for additional support please email the ESMF support list.

Put the initial cap code into your model source tree. Then, modify your Makefile or build scripts so that the cap is compiled with the rest of your model code. Unless your model is already using ESMF, you'll need to add ESMF compile and linking flags in order to build the cap. **When ESMF is installed, a Makefile fragment named esmf.mk is generated that contains variables that can be appended to your compile and link flags.** The ESMF User Guide explains how to use these variables in your Makefile.

3.5 Modify Your Build to Generate a NUOPC Makefile Fragment

The goal of adding a NUOPC cap to your model is so that it can be used with other NUOPC-compliant models in a coupled system. From a technical standpoint, there are several ways that your model code can be included into a final coupled system binary. Two common options are to link to your model statically and to link it in dynamically from a shared library.

In either case, to simplify the process of compiling and linking against your model, **your model's build process should produce a Makefile fragment file that defines the following six variables:**

ESMF_DEP_FRONT The name of the Fortran module to be used in a USE statement, or (if it ends in ".h") the name of the header file to be used in an #include statement, or (if it ends in ".so") the name of the shared object to be loaded at run-time.

ESMF_DEP_INCPATH The include path to find module or header files during compilation. Must be specified as absolute path.

ESMF_DEP_CMPL_OBJS Object files that need to be considered as compile dependencies. Must be specified with absolute path.

ESMF_DEP_LINK_OBJS Object files that need to be considered as link dependencies. Must be specified with absolute path.

ESMF_DEP_SHRD_PATH The path or list of paths to find shared libraries during link-time (and during run-time unless overridden by LD_LIBRARY_PATH). Must be specified as absolute paths.

ESMF_DEP_SHRD_LIBS Shared libraries that need to be specified during link-time, and must be available during run-time. Must be specified with absolute path.

An example makefile fragment useful for statically linking against your model looks like this:

```
#file: abc.mk

ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH    = <absolute path to associated ABC module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/abc.o <absolute path>/xyz.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

The variables in the makefile fragment expose a set of dependencies that the higher-level build system can use to compile and link against your model. An easy way to generate the makefile fragment is to modify your model's Makefile to include a new target:

```
.PRECIOUS: %.o

%.mk: %.o
   @echo "# ESMF self-describing build dependency makefile fragment" > $@
   @echo >> $@
   @echo "ESMF_DEP_FRONT     = ABC"                          >> $@
   @echo "ESMF_DEP_INCPATH   = `pwd`"                        >> $@
   @echo "ESMF_DEP_CMPL_OBJS = `pwd`/"$<                     >> $@
   @echo "ESMF_DEP_LINK_OBJS = "$(addprefix `pwd`/, $(OBJS)) >> $@
   @echo "ESMF_DEP_SHRD_PATH = "                             >> $@
   @echo "ESMF_DEP_SHRD_LIBS = "                             >> $@

abc.mk: $(OBJS)
```

The Standardized Component Dependencies section of the NUOPC Reference Manual contains more details on setting up NUOPC makefile fragments.

Finally, if your build procedure typically produces an executable, it is recommended that you add a Makefile target (or similar build option) that produces a library instead of an executable. When used in a NUOPC system, your model's main program will not be used–instead, a `NUOPC_Driver` will be linked to your cap and it will be the locus of control (i.e., the main program).

Makefile Target Conventions

If your model is built using Make, a common convention is to add two special targets that build your model and also compile the NUOPC code you will write.

```
# this target builds your model and your NUOPC cap
$ make nuopc
# this target installs your NUOPC-compliant model to a particular directory
$ make nuopcinstall DESTDIR=/path/to/install
```

3.6 Initialize Your Model from the Cap

The cap template you placed in your source tree is not yet connected to your model. You now need to add a call into your model's existing initialization subroutine(s).

NUOPC defines a precise initialization sequence–i.e., a series of steps that all NUOPC components are expected to take when starting up. A user component cap interacts with the NUOPC initialization sequence through specific specialization points. Specifically this means using the `NUOPC_CompSpecialize()` method during the component's SetServices method for each of the required specializations, and providing the necessary implementation. The `NUOPC_CompSpecialize()` method takes a `specLabel` argument to indicate the targeted specialization. All available specialization labels for model components are listed in the NUOPC reference manual under the NUOPC_Model API section.

Instead of tackling the full NUOPC initialization sequence at this point in developing your cap, we recommend that you start by adding calls in your cap's first initialization phase to your model's existing initialization subroutine(s). A good place to do this is within the Advertise Fields initialization phase. This is the phase where each component "advertises" the fields it requires and can potentially provide.

You will need to add `use` statements at the top if your cap to import the relevant initialization subroutines from your model into the NUOPC cap module. The example code in section 3.11 shows where to add the call to your model's initialization subroutine(s).

In the next section you will add another call into your model code before attempting to execute your NUOPC cap.

3.7 Call Your Model's Run Subroutine from the Cap

The Advance specialization point provided by the NUOPC Model generic component is the place where you will call your model's timestep subroutine. You should add this call now. Refer to the example code in section 3.11 below to see where to add this call.

This call should only move the model forward a single timestep, not the full run length. If the subroutine requires a parameter such as the timestep length or the time to stop, then these parameters can be retrieved from the cap's `ESMF_Clock` object.

If your model does not have a subroutine that takes a single timestep, you will need to create one now.

3.8 Run the Cap with a NUOPC Driver

Now you should test the basic cap you have implemented. First, build your model along with the cap code using your model's build script or Makefile. If you followed the procedure in section 3.5, your build process should have produced a NUOPC Makefile fragment file in addition to the compiled object files (or library).

One option for testing the cap is to run it using the NUOPC Component Explorer, a specialized `NUOPC_Driver` designed to execute any `NUOPC_Model`. Complete instructions for acquiring the Component Explorer and linking it to your NUOPC cap are available.

The instructions above also describe how to turn on the NUOPC Compliance Checker while running the Component Explorer. The Compliance Checker produces additional output in the ESMF log files that is useful for debugging. It also produces WARNINGS in the logs if a compliance issue is identified. When running with the basic cap, you should not necessarily expect to have all compliance issues resolved.

3.9 Split Up the Initialization Phases

Once the basic cap described above can be executed using the Component Explorer, you should modify the cap to implement the required initialization sequence as described in the NUOPC reference manual. This includes advertising fields with standard names and realizing fields by creating `ESMF_Field` objects to wrap your model variables. As part of this process, you will need to describe your model's grid structure using the ESMF geometric classes, e.g., `ESMF_Grid` and `ESMF_Mesh`.

3.10 Test and Validate Your Cap

After splitting up the initilization phases, rebuild your model and execute it again using the Component Explorer with the Compliance Checker turned on. Ideally, you should see no compliance WARNINGS in the generated log files.

To validate that the NUOPC cap is faithfully reproducing your model's behavior when run in non-NUOPC mode, you should compare your model's output when run with the NUOPC cap against a baseline run. This is the best test to ensure that the cap is working correctly. If the NUOPC cap reproduces your baseline run, you are ready to integrate your NUOPC Model cap into a coupled system with other NUOPC components.

3.11 Example NUOPC Model cap

The following code is a starting point for creating a basic NUOPC Model cap.

```
module MYMODEL

  !-------------------------------------------------------------------------
  ! Basic NUOPC Model cap
  !-------------------------------------------------------------------------

  use ESMF
  use NUOPC
  use NUOPC_Model, &
    modelSS    => SetServices

  ! add use statements for your model's initialization
  ! and run subroutines

  implicit none

  private

  public :: SetServices

  !-------------------------------------------------------------------------
  contains
  !-------------------------------------------------------------------------

  subroutine SetServices(model, rc)
    type(ESMF_GridComp)  :: model
    integer, intent(out) :: rc

    rc = ESMF_SUCCESS
```

```
  ! derive from NUOPC_Model
  call NUOPC_CompDerive(model, modelSS, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

  ! specialize model
  call NUOPC_CompSpecialize(model, specLabel=label_Advertise, &
    specRoutine=Advertise, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out
  call NUOPC_CompSpecialize(model, specLabel=label_RealizeProvided, &
    specRoutine=Realize, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out
  call NUOPC_CompSpecialize(model, specLabel=label_Advance, &
    specRoutine=Advance, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

end subroutine

!-------------------------------------------------------------------------------

subroutine Advertise(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  rc = ESMF_SUCCESS

  ! Eventually, you will advertise your model's import and
  ! export fields in this phase.  For now, however, call
  ! your model's initialization routine(s).

  ! call my_model_init()

end subroutine

!-------------------------------------------------------------------------------

subroutine Realize(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  rc = ESMF_SUCCESS

  ! Eventually, you will realize your model's fields here,
  ! but leave empty for now.
```

```
  end subroutine

  !-----------------------------------------------------------------------------

  subroutine Advance(model, rc)
    type(ESMF_GridComp)  :: model
    integer, intent(out) :: rc

    ! local variables
    type(ESMF_Clock)              :: clock
    type(ESMF_State)              :: importState, exportState

    rc = ESMF_SUCCESS

    ! query the Component for its clock, importState and exportState
    call NUOPC_ModelGet(model, modelClock=clock, importState=importState, &
      exportState=exportState, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    ! HERE THE MODEL ADVANCES: currTime -> currTime + timeStep

    ! Because of the way that the internal Clock was set by default,
    ! its timeStep is equal to the parent timeStep. As a consequence the
    ! currTime + timeStep is equal to the stopTime of the internal Clock
    ! for this call of the Advance() routine.

    call ESMF_ClockPrint(clock, options="currTime", &
      preString="------>Advancing MODEL from: ", rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    call ESMF_ClockPrint(clock, options="stopTime", &
      preString="--------------------------------> to: ", rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    ! Call your model's timestep routine here

    ! call my_model_update()

  end subroutine

end module
```

4 An Example Cap

In this section we'll look at code for an example NUOPC Model cap. The example shows the basic structure of a NUOPC Model cap for a fictitious atmosphere model called ATM. It is slightly simpler than a "real" cap, but has enough detail to show the basic coding structures. Each section of the example cap code will be broken down and described separately.

Finding More NUOPC Code Examples
In addition to the example code in this section, the NUOPC Prototypes Subversion repository contains many small example applications that are helpful for understanding the architecture of NUOPC applications and showing example uses of the NUOPC API. These example applications can be compiled and executed on your system.

A good starting point is the SingleModelProto application, which includes a single Model with a Driver and the AtmOcnProto application which includes two Models, a Connector, and a Driver.

4.1 Module Imports

The required NUOPC subroutines in the cap are grouped into a Fortran module, here called ATM. All NUOPC Model caps will import the `ESMF`, `NUOPC`, and `NUOPC_Model` modules. Typically, other `use` statements will appear as well to import subroutines and variables from your model code. The module exposes only a single public entry point (subroutine) called `SetServices`. This should be true for all NUOPC Model caps.

```
module ATM

  !-------------------------------------------------------------------------
  ! Basic NUOPC Model cap for ATM component (a fictitious atmosphere model).
  !-------------------------------------------------------------------------

  use ESMF
  use NUOPC
  use NUOPC_Model, &
    modelSS    => SetServices

  implicit none

  private

  public :: SetServices

  !-------------------------------------------------------------------------
  contains
  !-------------------------------------------------------------------------
```

4.2 SetServices

Every NUOPC Component must include a `SetServices` subroutine similar to the one shown below. All NUOPC `SetServices` routines have the same parameter list and should do several things:

- indicate the generic component being specialized,
- register any specialization points.

In the example code, the call to `NUOPC_CompDerive()` indicates that this component derives from (and specializes) the generic `NUOPC_Model` component. In other words, this is a `NUOPC_Model` component customized for a specific model.

The calls to `NUOPC_CompSpecialize()` register subroutines that are implemented in the cap. The `specLabel` argument specifies NUOPC-defined specialization labels. NUOPC defines explicitly what happens during each phase of the initialization and these labels uniquely define any specialization that might be supplied by the user. For example, `label_Advertise` is responsible for advertising field in the import- and exportState of the component. The `NUOPC_CompSpecialize()` also takes the `specRoutine` argument to indicate what routine provides the actual specialization. This subroutine appears later on in the cap and the name of the registered subroutine is entirely up to you.

The same specialization approach is used to specialize the generic Run method. Here `label_Advance` is specialized by subroutine `Advance`. The Advance specialization point is called by NUOPC whenever it needs your model to take a single timestep forward. Basically, this means you'll need to add a call inside the specialization subroutine to your model's timestepping subroutine.

```
subroutine SetServices(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  rc = ESMF_SUCCESS

  ! the NUOPC model component will register the generic methods
  call NUOPC_CompDerive(model, modelSS, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

  ! specialize model
  call NUOPC_CompSpecialize(model, specLabel=label_Advertise, &
    specRoutine=Advertise, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out
  call NUOPC_CompSpecialize(model, specLabel=label_RealizeProvided, &
    specRoutine=Realize, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out
  call NUOPC_CompSpecialize(model, specLabel=label_Advance, &
    specRoutine=Advance, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

end subroutine
```

4.3 Checking Return Codes

Essentially all ESMF and NUOPC methods have an optional integer return code parameter. You are highly encouraged to call `ESMF_LogFoundError` after every ESMF/NUOPC call in order to check the return code and record any errors in the log files that ESMF generates during the run. Including the `line` and `file` parameters will help to located where errors occur in the code. These parameter values are typically filled in by the pre-processor.

4.4 Initialize Phase - Advertise Fields

In this section we see the implementation of the `Advertise` subroutine, which is registered for the `label_Advertise` specialization. The full list of specialization labels is described in the NUOPC Reference Manual.

For now you should notice a few things:

- All specialization subroutines are standard ESMF attachable methods with the same parameter list:
  - `model` - a reference to the component itself (`ESMF_GridComp`)
  - `rc` - an `integer` return code
- If the subroutine succeeds, it should return `ESMF_SUCCESS` in the return code. Otherwise it should return an error code. The list of pre-defined ESMF error codes is provided in the ESMF Reference Manual.

The purpose of this phase is for your model to **advertise its import and export fields**. This means that your model announces which model variables it is capable of exporting (e.g., an atmosphere might export air pressure at sea level) and which model variables it requires (e.g., an atmosphere might require sea surface temperature as a boundary condition). The reason there is an explicit **advertise** phase is because NUOPC dynamically matches fields among all the models participating in a coupled simulation during runtime. So, we need to collect the list of possible input and output fields from all the models during their initialization.

As shown in the code below, to advertise a field you call `NUOPC_Advertise` with the following parameters:

- either the `importState` or `exportState` object
- the standard name of the field, based on the CF conventions
- an optional name for the field, which can be used to retrieve it later from its `ESMF_State`; if ommited the standard name will be used as the field name
- a return code

The example code below advertises one import field with the standard name `"sea_surface_temperature"`, and two export fields with standard names `"air_pressure_at_sea_level"` and `"surface_net_downward_shortwave_flux"`.

Advertising a Field does NOT allocate memory
Note that NUOPC does not allocate memory for fields during the advertise phase or when `NUOPC_Advertise` is called. Instead, this is simply a way for models to communicate the standard names of fields. During a later phase, only those fields that are *connected* (e.g., a field exported from one model that is imported by another) need to have memory allocated. Also, since ESMF will accept pointers to pre-allocated memory, it is usually not necessary to change how memory is allocated for your model's variables.

```
!-------------------------------------------------------------------------
```

```
subroutine Advertise(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  ! local variables
  type(ESMF_State)        :: importState, exportState

  rc = ESMF_SUCCESS

  ! query for importState and exportState
  call NUOPC_ModelGet(model, importState=importState, &
    exportState=exportState, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

  ! importable field: sea_surface_temperature
  call NUOPC_Advertise(importState, &
    StandardName="sea_surface_temperature", name="sst", &
    TransferOfferGeomObject="will provide", rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

  ! exportable field: air_pressure_at_sea_level
  call NUOPC_Advertise(exportState, &
    StandardName="air_pressure_at_sea_level", name="pmsl", &
    TransferOfferGeomObject="will provide", rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

  ! exportable field: surface_net_downward_shortwave_flux
  call NUOPC_Advertise(exportState, &
    StandardName="surface_net_downward_shortwave_flux", name="rsns", &
    TransferOfferGeomObject="will provide", rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

end subroutine
```

4.5 Initialize Phase - Realize Fields

The following code fragment shows the `Realize` subroutine, which specializes `label_RealizeProvided`. During this phase, fields that were previously advertised should now be **realized**. Realizing a field means that an `ESMF_Field` object is created and it is added to the appropriate `ESMF_State`, either import or export.

In order to create an ESMF_Field, you'll first need to create one of the ESMF geometric types, ESMF_Grid, ESMF_Mesh, or ESMF_LocStream. For 2D and 3D logically rectangular grids (such as a lat-lon grid), the typical choice is ESMF_Grid. For unstructured grids, use an ESMF_Mesh.

Describing your model's grid (physical discretization) in the ESMF representation is one of the most important parts of creating a NUOPC cap. The ESMF geometric types are described in detail in the ESMF Reference Manual:

- ESMF_Grid - logically rectangular grids
- ESMF_Mesh - unstructured grids
- ESMF_LocStream - a set of observational points

For the sake a simplicity, a 10x100 Cartesian grid is created in the code below and assigned to the variable gridIn.

An ESMF_Field is created by by passing in the field name (should be the same as advertised), the grid, and the data type of the field to ESMF_FieldCreate.

Fields are put into import or export States by calling NUOPC_Realize. The example code realizes three fields in total, one import and two export, and all three share the same grid.

```
subroutine Realize(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  ! local variables
  type(ESMF_State)        :: importState, exportState
  type(ESMF_Field)        :: field
  type(ESMF_Grid)         :: gridIn
  type(ESMF_Grid)         :: gridOut

  rc = ESMF_SUCCESS

  ! query for importState and exportState
  call NUOPC_ModelGet(model, importState=importState, &
    exportState=exportState, rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out

  ! create a Grid object for Fields
  gridIn = ESMF_GridCreateNoPeriDimUfrm(maxIndex=(/10, 100/), &
    minCornerCoord=(/10._ESMF_KIND_R8, 20._ESMF_KIND_R8/), &
    maxCornerCoord=(/100._ESMF_KIND_R8, 200._ESMF_KIND_R8/), &
    coordSys=ESMF_COORDSYS_CART, staggerLocList=(/ESMF_STAGGERLOC_CENTER/), &
    rc=rc)
  if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
    line=__LINE__, &
    file=__FILE__)) &
    return  ! bail out
  gridOut = gridIn ! for now out same as in

  ! importable field: sea_surface_temperature
  field = ESMF_FieldCreate(name="sst", grid=gridIn, &
    typekind=ESMF_TYPEKIND_R8, rc=rc)
```

```
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out
    call NUOPC_Realize(importState, field=field, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    ! exportable field: air_pressure_at_sea_level
    field = ESMF_FieldCreate(name="pmsl", grid=gridOut, &
      typekind=ESMF_TYPEKIND_R8, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out
    call NUOPC_Realize(exportState, field=field, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    ! exportable field: surface_net_downward_shortwave_flux
    field = ESMF_FieldCreate(name="rsns", grid=gridOut, &
      typekind=ESMF_TYPEKIND_R8, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out
    call NUOPC_Realize(exportState, field=field, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

  end subroutine
```

4.6 Model Advance Specialization

As described in the section 4.2, the subroutine `Advance` (shown below) has been registered to the *specialization point* with the label `model_label_Advance` in the `SetServices` subroutine. This specialization point subroutine is called within the generic `NUOPC_Model` run phase in order to request that your model take a timestep forward. The code to do this is model dependent, so it does not appear in the subroutine below.

Each NUOPC component maintains its own clock (an `ESMF_Clock` object). The clock is used here to indicate the current model time and the timestep size. When the subroutine finishes, your model should be moved ahead in time from the current time by one timestep. NUOPC will automatically advance the clock for you, so there is no explicit call to do that here.

Since there is no actual model for us to advance in this example, the code below simply prints the current time and stop time (current time + timestep) to standard out.

With respect to specialization point subroutines in general, note that:

- All specialization point subroutines rely on the ESMF attachable methods capability, and therefore all share the same parameter list:
  - a pointer to the component (`ESMF_GridComp`)
  - an `integer` return code
- Because the import/export states and the clock do not come in through the parameter list, they must be accessed via a call to `NUOPC_ModelGet` as shown in the code below.

```
  subroutine Advance(model, rc)
    type(ESMF_GridComp)  :: model
    integer, intent(out) :: rc

    ! local variables
    type(ESMF_Clock)              :: clock
    type(ESMF_State)              :: importState, exportState

    rc = ESMF_SUCCESS

    ! query the Component for its clock, importState and exportState
    call NUOPC_ModelGet(model, modelClock=clock, importState=importState, &
      exportState=exportState, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    ! HERE THE MODEL ADVANCES: currTime -> currTime + timeStep

    ! Because of the way that the internal Clock was set by default,
    ! its timeStep is equal to the parent timeStep. As a consequence the
    ! currTime + timeStep is equal to the stopTime of the internal Clock
    ! for this call of the Advance() routine.

    call ESMF_ClockPrint(clock, options="currTime", &
      preString="------>Advancing ATM from: ", rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    call ESMF_ClockPrint(clock, options="stopTime", &
      preString="--------------------------------> to: ", rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

  end subroutine

end module
```