

National Unified Operational Prediction Capability

NUOPC Layer Reference

ESMF 8.1.0 beta snapshot

Content Standards Committee (CSC) Members

June 25, 2020

Contents

1	Description	5
2	Design and Implementation Notes	5
2.1	Generic Components	5
2.2	Field Dictionary	8
2.2.1	Field Dictionary file	8
2.2.2	Preloaded Field Dictionary	9
2.3	Metadata	9
2.3.1	Driver Component Metadata	10
2.3.2	Model Component Metadata	11
2.3.3	Mediator Component Metadata	14
2.3.4	Connector Component Metadata	16
2.3.5	State Metadata	18
2.3.6	Field Metadata	18
2.4	Initialization	20
2.4.1	Phase Maps and Component Labels	20
2.4.2	Initialize Phase Definitions	20
2.4.3	Field Pairing	25
2.4.4	Namespaces	25
2.4.5	Connection Options	26
2.4.6	Data-Dependencies during Initialize	28
2.4.7	Transfer of Grid/Mesh/LocStream Objects between Components	29
2.4.8	Field and Grid/Mesh/LocStream Reference Sharing	30
2.4.9	Field Mirroring	31
2.5	Timekeeping	32
2.6	Component Hierarchies	33
3	API	35
3.1	Generic Component: NUOPC_Driver	35
3.1.1	NUOPC_DriverAddComp	37
3.1.2	NUOPC_DriverAddComp	38
3.1.3	NUOPC_DriverAddComp	39
3.1.4	NUOPC_DriverAddRunElement	40
3.1.5	NUOPC_DriverAddRunElement	40
3.1.6	NUOPC_DriverAddRunElement	41
3.1.7	NUOPC_DriverEgestRunSequence	41
3.1.8	NUOPC_DriverGet	42
3.1.9	NUOPC_DriverGetComp	42
3.1.10	NUOPC_DriverGetComp	43
3.1.11	NUOPC_DriverGetComp	43
3.1.12	NUOPC_DriverGetComp	44
3.1.13	NUOPC_DriverIngestRunSequence	44
3.1.14	NUOPC_DriverNewRunSequence	47
3.1.15	NUOPC_DriverPrint	47
3.1.16	NUOPC_DriverSetRunSequence	47
3.2	Generic Component: NUOPC_ModelBase	48
3.3	Generic Component: NUOPC_Model	49
3.3.1	Initialize Phase Specialization - label_SetClock	52
3.3.2	Initialize Phase Specialization - label_DataInitialize	52

3.3.3	Run Phase Specialization - label_SetRunClock	52
3.3.4	Run Phase Specialization - label_CheckImport	53
3.3.5	Run Phase Specialization - label_Advance	53
3.3.6	Run Phase Specialization - label_TimestampExport	53
3.3.7	Finalize Phase Specialization - label_Finalize	53
3.3.8	NUOPC_ModelGet	53
3.4	Generic Component: NUOPC_Mediator	54
3.4.1	NUOPC_MediatorGet	55
3.5	Generic Component: NUOPC_Connector	56
3.5.1	NUOPC_ConnectorGet	58
3.5.2	NUOPC_ConnectorSet	59
3.6	General Generic Component Methods	61
3.6.1	NUOPC_CompAreServicesSet	61
3.6.2	NUOPC_CompAreServicesSet	61
3.6.3	NUOPC_CompAttributeAdd	62
3.6.4	NUOPC_CompAttributeAdd	62
3.6.5	NUOPC_CompAttributeEgest	62
3.6.6	NUOPC_CompAttributeEgest	63
3.6.7	NUOPC_CompAttributeGet	63
3.6.8	NUOPC_CompAttributeGet	64
3.6.9	NUOPC_CompAttributeGet	64
3.6.10	NUOPC_CompAttributeGet	65
3.6.11	NUOPC_CompAttributeGet	65
3.6.12	NUOPC_CompAttributeGet	66
3.6.13	NUOPC_CompAttributeIngest	66
3.6.14	NUOPC_CompAttributeIngest	67
3.6.15	NUOPC_CompAttributeReset	68
3.6.16	NUOPC_CompAttributeReset	69
3.6.17	NUOPC_CompAttributeSet	69
3.6.18	NUOPC_CompAttributeSet	70
3.6.19	NUOPC_CompAttributeSet	70
3.6.20	NUOPC_CompAttributeSet	70
3.6.21	NUOPC_CompAttributeSet	71
3.6.22	NUOPC_CompAttributeSet	71
3.6.23	NUOPC_CompCheckSetClock	72
3.6.24	NUOPC_CompDerive	72
3.6.25	NUOPC_CompDerive	73
3.6.26	NUOPC_CompFilterPhaseMap	73
3.6.27	NUOPC_CompFilterPhaseMap	74
3.6.28	NUOPC_CompGet	74
3.6.29	NUOPC_CompGet	75
3.6.30	NUOPC_CompSearchPhaseMap	75
3.6.31	NUOPC_CompSearchPhaseMap	76
3.6.32	NUOPC_CompSearchRevPhaseMap	76
3.6.33	NUOPC_CompSearchRevPhaseMap	77
3.6.34	NUOPC_CompSetClock	77
3.6.35	NUOPC_CompSetEntryPoint	78
3.6.36	NUOPC_CompSetEntryPoint	78
3.6.37	NUOPC_CompSetInternalEntryPoint	79
3.6.38	NUOPC_CompSetServices	80
3.6.39	NUOPC_CompSpecialize	80

3.6.40	NUOPC_CompSpecialize	81
3.7	Field Dictionary Methods	81
3.7.1	NUOPC_FieldDictionaryAddEntry	81
3.7.2	NUOPC_FieldDictionaryEgest	82
3.7.3	NUOPC_FieldDictionaryGetEntry	82
3.7.4	NUOPC_FieldDictionaryHasEntry	83
3.7.5	NUOPC_FieldDictionaryMatchSyno	83
3.7.6	NUOPC_FieldDictionarySetSyno	84
3.7.7	NUOPC_FieldDictionarySetup	84
3.7.8	NUOPC_FieldDictionarySetup	84
3.8	Free Format Methods	85
3.8.1	NUOPC_FreeFormatAdd	85
3.8.2	NUOPC_FreeFormatCreate	85
3.8.3	NUOPC_FreeFormatCreate	86
3.8.4	NUOPC_FreeFormatDestroy	86
3.8.5	NUOPC_FreeFormatGet	87
3.8.6	NUOPC_FreeFormatGetLine	87
3.8.7	NUOPC_FreeFormatLog	88
3.8.8	NUOPC_FreeFormatPrint	88
3.9	Utility Routines	88
3.9.1	NUOPC_AddNamespace	88
3.9.2	NUOPC_AddNestedState	89
3.9.3	NUOPC_Advertise	90
3.9.4	NUOPC_Advertise	91
3.9.5	NUOPC_AdjustClock	92
3.9.6	NUOPC_CheckSetClock	93
3.9.7	NUOPC_GetAttribute	93
3.9.8	NUOPC_GetAttribute	94
3.9.9	NUOPC_GetAttribute	95
3.9.10	NUOPC_GetStateMemberLists	96
3.9.11	NUOPC_GetTimestamp	97
3.9.12	NUOPC_IsAtTime	98
3.9.13	NUOPC_IsAtTime	98
3.9.14	NUOPC_IsConnected	99
3.9.15	NUOPC_IsConnected	100
3.9.16	NUOPC_IsUpdated	100
3.9.17	NUOPC_IsUpdated	101
3.9.18	NUOPC_NoOp	102
3.9.19	NUOPC_Realize	102
3.9.20	NUOPC_Realize	103
3.9.21	NUOPC_Realize	104
3.9.22	NUOPC_Realize	105
3.9.23	NUOPC_Realize	106
3.9.24	NUOPC_SetAttribute	107
3.9.25	NUOPC_SetAttribute	108
3.9.26	NUOPC_SetTimestamp	108
3.9.27	NUOPC_SetTimestamp	109
3.9.28	NUOPC_SetTimestamp	110
3.10	Auxiliary Routines	110
3.10.1	NUOPC_Write	110
3.10.2	NUOPC_Write	111

3.10.3	NUOPC_Write	112
3.10.4	NUOPC_Write	113
4	Standardized Component Dependencies	115
4.1	Fortran components that are statically built into the executable	116
4.2	Fortran components that are provided as shared libraries	119
4.3	Components that are loaded during run-time as shared objects	120
4.4	Components that depend on components	121
4.5	Components written in C/C++	123
5	NUOPC Layer Compliance	126
5.1	The Compliance Checker	126
5.2	The Component Explorer	128
6	Appendix A: Run Sequence	131

1 Description

The NUOPC Layer is an add-on to the standard ESMF library. It consists of generic code of two different kinds: *utility routines* and *generic components*. The NUOPC Layer further implements a dictionary for standard field metadata.

The utility routines are subroutines and functions that package frequently used calling sequences of ESMF methods into single calls. Unlike the pure ESMF API, which is very class centric, the utility routines of the NUOPC Layer often implement tasks that involve several ESMF classes.

The generic components are provided in form of Fortran modules that implement GridComp and CplComp specific methods. Generic components are useful when implementing NUOPC compliant driver, model, mediator, or connector components. The provided generic components form a hierarchy that allows the developer to pick and choose the appropriate level of specification for a certain application. Depending on how specific the chosen level, generic components require more or less specialization to result in fully implemented components.

2 Design and Implementation Notes

The NUOPC Layer is implemented in Fortran on top of the public ESMF Fortran API.

The NUOPC utility routines form a very straightforward Fortran API, accessible through the NUOPC Fortran module. The interfaces only use native Fortran types and public ESMF derived types. In order to access the utility API of the NUOPC Layer, user code must include the following two `use` lines:

```
use ESMF
use NUOPC
```

2.1 Generic Components

The NUOPC generic components are implemented as a *collection* of Fortran modules. Each module implements a single, well specified set of standard ESMF_GridComp or ESMF_CplComp methods. The nomenclature of the generic component modules starts with the NUOPC_ prefix and continues with the kind: `Driver`, `Model`, `Mediator`, or `Connector`. This is optionally followed by a string of additional descriptive terms. The four kinds of generic components implemented by the NUOPC Layer are:

- `NUOPC_Driver` - A generic driver component. It implements a child component harness, made of State and Component objects, that follows the NUOPC Common Model Architecture. It is specialized by plugging `Model`, `Mediator`, and `Connector` components into the harness. `Driver` components can be plugged into the harness to construct component hierarchies. The generic `Driver` initializes its child components according to a standard Initialization Phase Definition, and drives their `Run()` methods according a customizable run sequence.
- `NUOPC_Model` - A generic model component that wraps a model code so it is suitable to be plugged into a generic `Driver` component.
- `NUOPC_Mediator` - A generic mediator component that wraps custom coupling code (flux calculations, averaging, etc.) so it is suitable to be plugged into a generic `Driver` component.
- `NUOPC_Connector` - A generic component that implements Field matching based on metadata and executes simple transforms (Regrid and Redist). It can be plugged into a generic `Driver` component.

The user code accesses the desired generic component(s) by including a `use` line for each one. Each generic component defines a small set of public names that are made available to the user code through the `use` statement. At a minimum the `SetServices` method is made public. Some of the generic components define additional public routines and labels as part of their user interface. It is recommended to rename the entries of an imported generic component module in the local scope as part of the `use` association. This prevents name clashes.

```
use NUOPC_<GenericComp>, only: &
  <GenericComp>_SS      => SetServices, &
  <GenericComp>_labelA  => labelA
```

A generic component is used by user code to implement a specialized version of the generic component. The user component derives from the generic component code by implementing its own public `SetServices` routine that calls into the generic `SetServices` routine before doing anything else. It is through this mechanism that the deriving component *inherits* functionality that is implemented in the generic component. The example shows how a specific model component is implemented to derive from the generic `NUOPC_Model`:

```
use NUOPC_Model, only: &
  model_SS => SetServices

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

  ! derive this specific "model" component from generic NUOPC_Model
  call NUOPC_CompDerive(model, model_SS, rc=rc)

  ! specializing code for "model" to follow

end subroutine
```

There are three mechanisms through which user code specializes generic components. The first two methods specialize through call-back mechanisms into user implemented routines. The third method specializes by setting the values of parameters implemented by the generic component.

1. The specializing user code sets entry points for standard component methods not implemented by the generic component by calling `NUOPC_CompSetEntryPoint()`. Methods (and phases) that need to be implemented are clearly documented in the generic component description. The user code may further overwrite standard methods already implemented by the generic component code. However, this should rarely be necessary, and may indicate that there is a better fitting generic component available. Finally, some generic components come with generic routines that are suitable candidates for the standard component methods, yet require that the specializing code registers them as appropriate. Setting entry points for standard component methods should be done in the `SetServices()` routine right after `NUOPC_CompDerive()` is called.
2. Some generic components require that specific methods are attached to the component by calling `NUOPC_CompSpecialize()`. If a generic component uses specialization through attachable methods, the specific method labels (i.e. the names by which these methods are registered) and the purpose of the method are clearly documented. In some cases attachable methods are optional. This is clearly documented. Further, some generic components attach a default method to a label, which then is used for all phases. This default can be overwritten with a phase specific attachable method. Attaching methods to the component should be done in the `SetServices` routine right after setting entry points for the standard component methods.

3. Some generic components offer methods that allow parameter specialization. The options of this specialization type are documented as part of the `Set ()` methods.

Components that inherit from a generic component may choose to only specialize certain aspects, leaving other aspects unspecified. This allows a hierarchy of generic components to be implemented with a high degree of code re-use. The variable level of specialization supports the very differing user needs. Figure 1 depicts the inheritance structure of the standard generic components implemented by the NUOPC Layer. There are two trees, one is rooted in `ESMF_GridComp`, while the other is rooted in `ESMF_CplComp`.

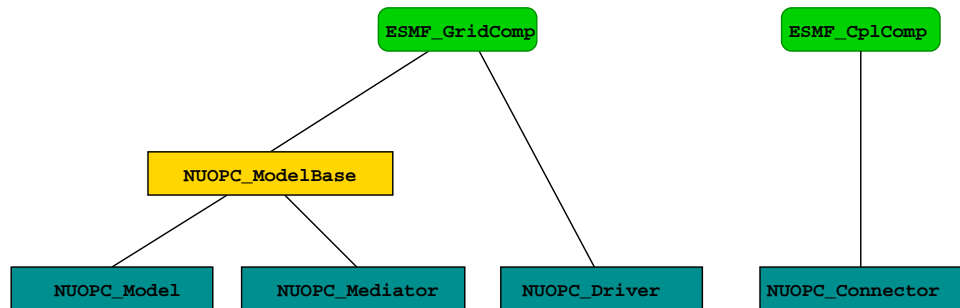


Figure 1: The NUOPC Generic Component inheritance structure. The tree on the left is rooted in `ESMF_GridComp`, while the tree on the right is rooted in `ESMF_CplComp`. The ESMF data types are shown in green. The four main NUOPC Generic Component kinds are shown in dark blue boxes. The yellow box shows a partial specialization in the inheritance tree.

2.2 Field Dictionary

The NUOPC Layer uses standard metadata on Fields to guide the decision making process that is implemented in generic code. The generic `NUOPC_Connector` component, for instance, uses the `StandardName` Attribute to construct a list of matching Fields between the import and export States. The NUOPC Field Dictionary provides a software implementation of a controlled vocabulary for the `StandardName` Field Attribute. It also associates each registered `StandardName` with `CanonicalUnits`. Currently the NUOPC Layer uses the `CanonicalUnits` entry to verify that Fields are provided in their canonical units. In the future, this entry may help support automatic unit conversion among exchanged fields.

The NUOPC Field Dictionary is set up by loading its content from a YAML 1.2 file. See section 2.2.1 for details.

Users can extend the dictionary by adding entries (field definitions or synonyms) to the YAML file, or by calling the `NUOPC_FieldDictionaryAddEntry()` interface.

2.2.1 Field Dictionary file

In a given NUOPC application, the NUOPC Field Dictionary can be set up by calling the `NUOPC_FieldDictionarySetup()` method to read in a properly-formatted YAML file. This feature is intended to improve the interoperability of codes that use the NUOPC Layer, as it allows a broader scientific community to contribute to the growth and upkeep of a common NUOPC Field Dictionary file shared among different Earth System Models. At this time, an initial version of the NUOPC Field Dictionary file is available through the dedicated GitHub repository: <https://github.com/ESCOMP/NUOPCFieldDictionary>, hosted within the Earth System Community Modeling Portal (ESCOMP).

A NUOPC Field Dictionary YAML file is codified as a YAML map (an unordered association of unique keys to values) with only one key: `field_dictionary`. The value associated with this key is itself a YAML map that should include the mandatory key `entries` (pointing to the complete set of dictionary entries), and may include the optional keys: `version_number`, `last_modified`, `institution`, `contact`, `source`, and `description`. These optional keys are intended to hold information about the file itself and are currently ignored by the NUOPC Layer.

Entries in the NUOPC Field dictionary are organized as YAML lists of maps. List items under the `entries` keyword must be indented and preceded with a hyphen (-).

A dictionary entry fully defines a Field if it includes both the `standard_name` and `canonical_units` keys and their associated values. This entry may also include a brief narrative describing the Field, stored as the value of the optional key `description`.

Synonyms can be defined by adding separate entries that include both the `alias` key, associated with either a single synonym (YAML scalar, e.g. `alias: <name>`) or a comma-separated list of synonyms within square brackets (YAML flow sequence, e.g. `alias: [<name1>, <name2>, ...]`), and the `standard_name` key associated with the original Field name to be substituted. The original Field name must be fully defined in the dictionary file. While adding one `alias` keyword to a Field definition dictionary entry is allowed and will be parsed by the NUOPC Layer, it is recommended that all synonyms be included as separate entries.

A NUOPC Field dictionary sample file is included below.

```
field_dictionary:
  version_number: 0.0.1
  last_modified: 2018-03-14T11:01:19Z
  institution: National ESPC, CSC & MCL Working Groups
  contact: esmf_support@ucar.edu
```

```

source:      https://github.com/ESCOMP/NUOPCFieldDictionary
description: Community-based dictionary for shared coupling fields

entries:
- standard_name: air_pressure
  canonical_units: Pa
  description: Air pressure
- standard_name: air_temperature
  canonical_units: K
  description:
    Bulk temperature of the air,
    not the surface (skin) temperature
- alias: p
  standard_name: air_pressure
- alias: [ t, temp ]
  standard_name: air_temperature

```

2.2.2 Preloaded Field Dictionary

A version of the NUOPC Field Dictionary is preloaded by the NUOPC Layer at start-up, and, at this time, consists of the entries show in the table below. The value of the `StandardName` Attribute in each of these entries complies with the Climate and Forecast (CF) conventions guidelines.

StandardName	CanonicalUnits
air_pressure_at_sea_level	Pa
magnitude_of_surface_downward_stress	Pa
precipitation_flux	kg m-2 s-1
sea_surface_height_above_sea_level	m
sea_surface_salinity	1e-3
sea_surface_temperature	K
surface_downward_eastward_stress	Pa
surface_downward_heat_flux_in_air	W m-2
surface_downward_northward_stress	Pa
surface_downward_water_flux	kg m-2 s-1
surface_eastward_sea_water_velocity	m s-1
surface_net_downward_longwave_flux	W m-2
surface_net_downward_shortwave_flux	W m-2
surface_northward_sea_water_velocity	m s-1

2.3 Metadata

The NUOPC Layer makes extensive use of the ESMF Attribute class to implement metadata on Components, States, and Fields. ESMF Attribute Packages (or AttPacks for short) are used to build an Attribute hierarchy for each object.

In some cases the lowest level NUOPC AttPack contains a nested AttPack defined by ESMF. For all objects, the highest level of the NUOPC AttPack hierarchy is implemented with `convention="NUOPC"`, `purpose="Instance"`. The public NUOPC Layer API allows a user to add Attributes to the highest AttPack hierarchy level.

2.3.1 Driver Component Metadata

The Driver component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
 - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Note that some of the Attribute names in the following table are longer than the table column width. In these cases the Attribute name had to be broken into multiple lines. When that happens, a hyphen shows up to indicate the line break. The hyphen is *not* part of the Attribute name!

Attribute name	Definition	Controlled vocabulary
Kind	String value indicating component kind.	Driver
Verbosity	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. bit 0: Intro/Extro of methods with indentation. bit 1: Intro/Extro with memory info. bit 2: Intro/Extro with garbage collection info. bit 3: Intro/Extro with local VM info. bit 8: Log Initialize phase with >>>, <<<, and currTime. bit 9: Log Run phase with >>>, <<<, and currTime. bit 10: Log Finalize phase with >>>, <<<, and currTime. bit 11: Log info about data dependency during initialize resolution. bit 12: Log run sequence execution. bit 13: Log Component creation. bit 14: Log State creation.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, "high": more verbosity, "max": all lower 16 bits
Profiling	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control. bit 0: Run sequence timings.	0, 1, 2, ... "max" = set all lower 16 bits
CompLabel	String value holding the label under which the component was added to its parent driver.	<i>no restriction</i>
InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
RunPhaseMap	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.

FinalizePhaseMap	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
Internal-InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
NestingGeneration	Integer value enumerating nesting level.	0, 1, 2, ...
Nestling	Integer value enumerating siblings within the same generation.	0, 1, 2, ...
Initialize-DataComplete	String value indicating whether all initialize data dependencies have been satisfied.	false, true
Initialize-DataProgress	String value indicating whether progress is being made resolving initialize data dependencies.	false, true
HierarchyProtocol	String value specifying the hierarchy protocol.	"PushUpAllExportsAndUnsatisfiedImports" - activates field mirroring of all exports and unsatisfied imports. By default use reference sharing for the mirrored fields and geom objects. This is the default behavior without having HierarchyProtocol set. "ConnectProvidedFields"- no field mirroring, only connect to externally provided fields in the import and exportStates. "Explorer" - like the default, but do not use reference sharing. <i>All other values currently disable the hierarchy protocol.</i>

2.3.2 Model Component Metadata

The Model component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
 - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Note that some of the Attribute names in the following table are longer than the table column width. In these cases the Attribute name had to be broken into multiple lines. When that happens, a hyphen shows up to indicate the line break. The hyphen is *not* part of the Attribute name!

Attribute name	Definition	Controlled vocabulary
----------------	------------	-----------------------

Kind	String value indicating component kind.	Model
Verbosity	<p>String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control.</p> <p>bit 0: Intro/Extro of methods with indentation. bit 1: Intro/Extro with memory info. bit 2: Intro/Extro with garbage collection info. bit 3: Intro/Extro with local VM info. bit 8: Log Initialize phase with >>>, <<<, and currTime. bit 9: Log Run phase with >>>, <<<, and currTime. bit 10: Log Finalize phase with >>>, <<<, and currTime. bit 11: Log info about data dependency during initialize resolution. bit 12: Log run sequence execution. bit 13: Log Component creation. bit 14: Log State creation.</p>	<p>0, 1, 2, ... "off" = 0 (default), "low": some verbosity, "high": more verbosity, "max": all lower 16 bits</p>
Profiling	<p>String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control.</p>	<p>0, 1, 2, ... "max" = set all lower 16 bits</p>

Diagnostic	<p>String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control diagnostic of the generic component implementation. Higher bits are available for user level diagnostic control.</p> <p>bit 0: Dump fields of the importState on entering the Initialize method.</p> <p>bit 1: Dump fields of the exportState on entering the Initialize method.</p> <p>bit 2: Dump fields of the importState on exiting the Initialize method.</p> <p>bit 3: Dump fields of the exportState on exiting the Initialize method.</p> <p>bit 4: Dump fields of the importState on entering the Run method.</p> <p>bit 5: Dump fields of the exportState on entering the Run method.</p> <p>bit 6: Dump fields of the importState on exiting the Run method.</p> <p>bit 7: Dump fields of the exportState on exiting the Run method.</p> <p>bit 8: Dump fields of the importState on entering the Finalize method.</p> <p>bit 9: Dump fields of the exportState on entering the Finalize method.</p> <p>bit 10: Dump fields of the importState on exiting the Finalize method.</p> <p>bit 11: Dump fields of the exportState on exiting the Finalize method.</p>	<p>0, 1, 2, ...</p> <p>"max" = set all lower 16 bits</p>
CompLabel	String value holding the label under which the component was added to its parent driver.	<i>no restriction</i>
InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
RunPhaseMap	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
FinalizePhaseMap	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
Internal-InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
NestingGeneration	Integer value enumerating nesting level.	0, 1, 2, ...
Nestling	Integer value enumerating siblings within the same generation.	0, 1, 2, ...

Initialize-DataComplete	String value indicating whether all initialize data dependencies have been satisfied.	false, true
Initialize-DataProgress	String value indicating whether progress is being made resolving initialize data dependencies.	false, true
HierarchyProtocol	String value specifying the hierarchy protocol.	"PushUpAllExportsAndUnsatisfiedImports" for field mirroring and connecting, "ConnectProvidedFields" to only connect provided fields (no mirroring), <i>All other values currently disable the hierarchy protocol.</i>

2.3.3 Mediator Component Metadata

The Mediator component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
 - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Note that some of the Attribute names in the following table are longer than the table column width. In these cases the Attribute name had to be broken into multiple lines. When that happens, a hyphen shows up to indicate the line break. The hyphen is *not* part of the Attribute name!

Attribute name	Definition	Controlled vocabulary
Kind	String value indicating component kind.	Mediator
Verbosity	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. bit 0: Intro/Extro of methods with indentation. bit 1: Intro/Extro with memory info. bit 2: Intro/Extro with garbage collection info. bit 3: Intro/Extro with local VM info. bit 8: Log Initialize phase with >>>, <<<, and currTime. bit 9: Log Run phase with >>>, <<<, and currTime. bit 10: Log Finalize phase with >>>, <<<, and currTime. bit 11: Log info about data dependency during initialize resolution. bit 12: Log run sequence execution. bit 13: Log Component creation. bit 14: Log State creation.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, "high": more verbosity, "max": all lower 16 bits

Profiling	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control.	0, 1, 2, ... "max" = set all lower 16 bits
Diagnostic	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control diagnostic of the generic component implementation. Higher bits are available for user level diagnostic control. bit 0: Dump fields of the importState on entering the Initialize method. bit 1: Dump fields of the exportState on entering the Initialize method. bit 2: Dump fields of the importState on exiting the Initialize method. bit 3: Dump fields of the exportState on exiting the Initialize method. bit 4: Dump fields of the importState on entering the Run method. bit 5: Dump fields of the exportState on entering the Run method. bit 6: Dump fields of the importState on exiting the Run method. bit 7: Dump fields of the exportState on exiting the Run method. bit 8: Dump fields of the importState on entering the Finalize method. bit 9: Dump fields of the exportState on entering the Finalize method. bit 10: Dump fields of the importState on exiting the Finalize method. bit 11: Dump fields of the exportState on exiting the Finalize method.	0, 1, 2, ... "max" = set all lower 16 bits
CompLabel	String value holding the label under which the component was added to its parent driver.	<i>no restriction</i>
InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
RunPhaseMap	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
FinalizePhaseMap	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
Internal-InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10

NestingGeneration	Integer value enumerating nesting level.	0, 1, 2, ...
Nestling	Integer value enumerating siblings within the same generation.	0, 1, 2, ...
Initialize-DataComplete	String value indicating whether all initialize data dependencies have been satisfied.	false, true
Initialize-DataProgress	String value indicating whether progress is being made resolving initialize data dependencies.	false, true
HierarchyProtocol	String value specifying the hierarchy protocol.	"PushUpAllExportsAndUnsatisfiedImports" for field mirroring and connecting, "ConnectProvidedFields" to only connect provided fields (no mirroring), <i>All other values currently disable the hierarchy protocol.</i>

2.3.4 Connector Component Metadata

The Connector Component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: General
- Includes:
 - ESG General (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Attribute name	Definition	Controlled vocabulary
Kind	String value indicating component kind.	Connector
Verbosity	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. bit 0: Intro/Extro of methods with indentation. bit 1: Intro/Extro with memory info. bit 2: Intro/Extro with garbage collection info. bit 3: Intro/Extro with local VM info. bit 8: Log FieldTransferPolicy. bit 9: Log bond level info. bit 10: Log CplList construction. bit 11: Log GeomObject Transfer. bit 12: Log looping over all elements in CplList for RouteHandle computation, FieldSharing, and Timestamp propagation. bit 13: Log Run phase with >>>, <<<, and currTime. bit 14: Log info about RouteHandle execution. bit 15: Log info about RouteHandle release.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, "high": more verbosity, "max": all lower 16 bits

Profiling	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control. bit 0: Run method execution timings. bit 1: Reconcile timings.	0, 1, 2, ... "max" = set all lower 16 bits
Diagnostic	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits are reserved to control diagnostic of the generic component implementation. Higher bits are available for user level diagnostic control. bit 0: Dump fields of the importState on entering the Initialize method. bit 1: Dump fields of the exportState on entering the Initialize method. bit 2: Dump fields of the importState on exiting the Initialize method. bit 3: Dump fields of the exportState on exiting the Initialize method. bit 4: Dump fields of the importState on entering the Run method. bit 5: Dump fields of the exportState on entering the Run method. bit 6: Dump fields of the importState on exiting the Run method. bit 7: Dump fields of the exportState on exiting the Run method. bit 8: Dump fields of the importState on entering the Finalize method. bit 9: Dump fields of the exportState on entering the Finalize method. bit 10: Dump fields of the importState on exiting the Finalize method. bit 11: Dump fields of the exportState on exiting the Finalize method.	0, 1, 2, ... "max" = set all lower 16 bits
CompLabel	String value holding the label under which the component was added to its parent driver.	<i>no restriction</i>
InitializePhaseMap	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
RunPhaseMap	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
FinalizePhaseMap	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	<i>label-string</i> =Z, where <i>label-string</i> can be chosen freely, and Z = actual ESMF phase number.
CplList	List of StandardNames of the connected Fields. Each StandardName entry may be followed by a colon separated list of connection options. The details are discussed in section 2.4.5	<i>Standard names</i> as per field dictionary, followed by <i>connection options</i> defined in section 2.4.5.
CplSetList	List of coupling sets. Each coupling set is identified by a string value.	<i>no restriction</i>
ConnectionOptions	Connection options to be applied to all the fields in the CplList by default.	<i>Connection options</i> defined in section 2.4.5.

2.3.5 State Metadata

The State metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
 - *no other packages at the moment*
- Description: Namespace implementation for import and export.

Attribute name	Definition	Controlled vocabulary
Namespace	String value holding the namespace of all the objects contained in the State.	<i>no restriction</i>
FieldTransferPolicy	String value indicating to Connector to transfer/mirror Fields.	transferNone, transferAll

2.3.6 Field Metadata

The Field metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
 - ESG General
- Description: Basic Field description with connection and time stamp metadata.

Attribute name	Definition	Controlled vocabulary
Connected	Connected status.	false, true
TimeStamp	Ten integer values representing ESMF Time object.	N/A
ProducerConnection	String value indicating whether the Field has been connected with a producer.	open, targeted, connected
ConsumerConnection	String value indicating whether the Field has been connected with a consumer.	open, targeted, connected
Updated	String value indicating updated status during initialization.	false, true
ProducerTransferOff	String value indicating a producer component's ability to transfer information about the advertised Field, including its GeomObject.	will provide, can provide, cannot provide

ProducerTransferAction	String value indicating the action a producer component is supposed to take with respect to transferring Field information, including its GeomObject.	provide, accept
ConsumerTransferOffer	String value indicating a consumer component's ability to transfer information about the advertised Field, including its GeomObject.	will provide, can provide, cannot provide
ConsumerTransferAction	String value indicating the action a consumer component is supposed to take with respect to transferring Field information, including its GeomObject.	provide, accept
SharePolicyField	String value indicating a component's policy with respect to sharing the Field data allocation.	share, not share
ShareStatusField	String value indicating the status with respect to sharing the underlying Field data allocation that was negotiated.	shared, not shared
SharePolicyGeomObject	String value indicating a component's policy with respect to sharing the Grid or Mesh on which the advertised Field object is defined.	share, not share
ShareStatusGeomObject	String value indicating the status with respect to sharing the underlying GeomObject that was negotiated.	shared, not shared
UngriddedLBound	Integer value list. If present equals the ungriddedLBound of the provider field.during a GeomObject transfer.	<i>no restriction</i>
UngriddedUBound	Integer value list. If present equals the ungriddedUBound of the provider field.during a GeomObject transfer.	<i>no restriction</i>
GridToFieldMap	Integer value list. If present equals the gridToFieldMap of the provider field.during a GeomObject transfer.	<i>no restriction</i>
ArbDimCount	Integer value. If present equals the arbDimCount of the provider field.during a GeomObject transfer.	<i>no restriction</i>
MinIndex	Integer value list. If present equals the minIndex (of tile 1) of the provider field.during a GeomObject transfer.	<i>no restriction</i>
MaxIndex	Integer value list. If present equals the maxIndex (of tile 1) of the provider field.during a GeomObject transfer.	<i>no restriction</i>
TypeKind	Integer value. If present equals the integer representation of typekind of the provider field.during a GeomObject transfer.	<i>implementation dependent range</i>
GeomLoc	Integer value. If present equals the integer representation of staggerloc (for Grid) or meshloc (for Mesh) of the provider field.during a GeomObject transfer.	<i>implementation dependent range</i>

2.4 Initialization

2.4.1 Phase Maps and Component Labels

ESMF introduces the concept of standard component methods: `Initialize`, `Run`, and `Finalize`. ESMF further recognizes the need for being able to split each of the standard methods into multiple phases. On the ESMF level, phases are implemented by a simple integer `phase` index. The NUOPC layer adds an abstraction layer that allows phases to be referenced by label.

For complex scenarios, e.g. multiple versions of multi-stage initialize sequences, the use of an integer based phase index quickly becomes confusing. The NUOPC Layer addresses this issue by introducing three component level attributes: `InitializePhaseMap`, `RunPhaseMap`, and `FinalizePhaseMap`. These attributes map logical NUOPC phase labels to integer ESMF phase indices. A NUOPC compliant component fully documents its available phases through the phase maps.

Currently the NUOPC Layer leverages the `InitializePhaseMap` during the initialization loop that is implemented by the generic `NUOPC_Driver`. It looks for phase map entries according to the initialize phase definition outlined in section 2.4.2. The `RunPhaseMap` is used when setting up run sequences in the Driver. The `NUOPC_DriverAddRunElement()` takes the `phaseLabel` argument, and uses the `RunPhaseMap` attribute internally to translate the label into the corresponding ESMF phase index. The `FinalizePhaseMap` is currently not used within the NUOPC Layer.

Within NUOPC, components under a driver are also referenced by label. Every component is associated with a label when it is added to a driver through the `NUOPC_DriverAddComp()` call. Multiple instances of the same component can be added to a driver, provided each instance is given a unique label. Connectors between components are identified by providing the label of the source component and destination component.

2.4.2 Initialize Phase Definitions

The interaction between NUOPC compliant components during the initialization process is regulated by the **Initialize Phase Definition** or **IPD**. The IPDs are versioned, with a higher version number indicating backward compatibility with all previous versions.

There are two perspectives of looking at the IPD. From the driver perspective the IPD regulates the sequence in which it must call the different phases of the `Initialize()` routines of its child components. To this end the generic `NUOPC_Driver` component implements support for IPDs up to a version specified in the API documentation.

The other angle of looking at the IPD is from the driver's child components. From this perspective the IPD assigns specific meaning to each initialize phase. The child components of a driver can be divided into two groups with respect to the meaning the IPD assigns to each initialize phase. In one group are the model, mediator, and driver components, and in the other group are the connector components. Child components publish their available initialize phases through the `InitializePhaseMap` attribute.

The driver also calls into its own internal initialize methods. This allows the driver to participate in the initialization of its children in a structured fashion. The internal initialization phases of a driver are published via the `InternalInitializePhaseMap` attribute.

The following tables document the meaning of each initialization phase of the available IPD versions for the child components and for the driver component itself. The phases are listed in the same sequence in which the driver calls them.

IPDv00 label	Component	Meaning
IPDv00p1	driver-internal	<i>unspecified by NUOPC</i>
IPDv00p1	models, mediators, drivers	Advertise their import and export Fields.
IPDv00p1	connectors	Construct their CplList Attribute.
IPDv00p2	driver-internal	<i>unspecified by NUOPC</i>
IPDv00p2	models, mediators, drivers	Realize their import and export Fields.
IPDv00p2a	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Reconcile the import and export States.
IPDv00p2b	connectors	Precompute the RouteHandle.
IPDv00p3	driver-internal	<i>unspecified by NUOPC</i>
IPDv00p3	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv00p4	driver-internal	<i>unspecified by NUOPC</i>
IPDv00p4	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.

IPDv01 label	Component	Meaning
IPDv01p1	driver-internal	<i>unspecified by NUOPC</i>
IPDv01p1	models, mediators, drivers	Advertise their import and export Fields.
IPDv01p1	connectors	Construct their CplList Attribute.
IPDv01p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv01p2	models, mediators, drivers	<i>unspecified/unused by NUOPC</i>
IPDv01p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute.
IPDv01p3	driver-internal	<i>unspecified by NUOPC</i>
IPDv01p3	models, mediators, drivers	Realize their "connected" import and export Fields.
IPDv01p3a	connectors	Reconcile the import and export States.
IPDv01p3b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv01p4	driver-internal	<i>unspecified by NUOPC</i>
IPDv01p4	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv01p5	driver-internal	<i>unspecified by NUOPC</i>
IPDv01p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.

IPDv02 label	Component	Meaning
IPDv02p1	driver-internal	<i>unspecified by NUOPC</i>
IPDv02p1	models, mediators, drivers	Advertise their import and export Fields.
IPDv02p1	connectors	Construct their CplList Attribute.
IPDv02p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv02p2	models, mediators, drivers	<i>unspecified/unused by NUOPC</i>
IPDv02p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute.
IPDv02p3	driver-internal	<i>unspecified by NUOPC</i>
IPDv02p3	models, mediators, drivers	Realize their "connected" import and export Fields.
IPDv02p3a	connectors	Reconcile the import and export States.
IPDv02p3b	connectors	Precompute the RouteHandle according to the CplList Attribute.

IPDv02p4	driver-internal	<i>unspecified by NUOPC</i>
IPDv02p4	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv02p5	driver-internal	<i>unspecified by NUOPC</i>
IPDv02p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.
<i>A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:</i>		
Run ()	connectors	Loop over all Connectors that connect to the Component that is currently indexed by the outer loop.
IPDv02p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
<i>Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.</i>		

IPDv03 label	Component	Meaning
IPDv03p1	driver-internal	<i>unspecified by NUOPC</i>
IPDv03p1	models, mediators, drivers	Advertise their import and export Fields and set the TransferOfferGeomObject Attribute.
IPDv03p1	connectors	Construct their CplList Attribute.
IPDv03p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv03p2	models, mediators, drivers	<i>unspecified/unused by NUOPC</i>
IPDv03p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Set the TransferActionGeomObject Attribute.
IPDv03p3	driver-internal	<i>unspecified by NUOPC</i>
IPDv03p3	models, mediators, drivers	Realize their "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv03p3	connectors	Transfer the Grid/Mesh/LocStream objects (only DistGrid) for Field pairs that have a provider and an acceptor side.
IPDv03p4	driver-internal	<i>unspecified by NUOPC</i>
IPDv03p4	models, mediators, drivers	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv03p4	connectors	Transfer the full Grid/Mesh/LocStream objects (with coordinates) for Field pairs that have a provider and an acceptor side.
IPDv03p5	driver-internal	<i>unspecified by NUOPC</i>
IPDv03p5	models, mediators, drivers	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv03p5a	connectors	Reconcile the import and export States.
IPDv03p5b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv03p6	driver-internal	<i>unspecified by NUOPC</i>
IPDv03p6	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv03p7	driver-internal	<i>unspecified by NUOPC</i>
IPDv03p7	models, mediators, drivers	Handle Field data initialization. Timestamp the export Fields.
<i>A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:</i>		
Run ()	connectors	Loop over all Connectors that connect to the Component that is currently indexed by the outer loop.

IPDv03p7	models, mediators, drivers	Handle Field data initialization. Time stamp the export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
<i>Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.</i>		

IPDv04 label	Component	Meaning
IPDv04p1	driver-internal	<i>unspecified by NUOPC</i>
IPDv04p1	models, mediators, drivers	Advertise their import and export Fields and set the TransferOfferGeomObject Attribute.
IPDv04p1a	connectors	Consider all connection possibilities for their CplList Attribute.
IPDv04p1b	connectors	Unambiguous construction of their CplList Attribute.
IPDv04p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv04p2	models, mediators, drivers	<i>unspecified/unused by NUOPC</i>
IPDv04p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Set the TransferActionGeomObject Attribute.
IPDv04p3	driver-internal	<i>unspecified by NUOPC</i>
IPDv04p3	models, mediators, drivers	Realize their "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv04p3	connectors	Transfer the Grid/Mesh/LocStream objects (only DistGrid) for Field pairs that have a provider and an acceptor side.
IPDv04p4	driver-internal	<i>unspecified by NUOPC</i>
IPDv04p4	models, mediators, drivers	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv04p4	connectors	Transfer the full Grid/Mesh/LocStream objects (with coordinates) for Field pairs that have a provider and an acceptor side.
IPDv04p5	driver-internal	<i>unspecified by NUOPC</i>
IPDv04p5	models, mediators, drivers	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv04p5a	connectors	Reconcile the import and export States.
IPDv04p5b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv04p6	driver-internal	<i>unspecified by NUOPC</i>
IPDv04p6	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv04p7	driver-internal	<i>unspecified by NUOPC</i>
IPDv04p7	models, mediators, drivers	Handle Field data initialization. Timestamp the export Fields.
<i>A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:</i>		
Run ()	connectors	Loop over all Connectors that connect to the Component that is currently indexed by the outer loop.
IPDv04p7	models, mediators, drivers	Handle Field data initialization. Time stamp the export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
<i>Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.</i>		

IPDv05 label	Component	Meaning
IPDv05p1	driver-internal	Advertise import and export Fields and set the TransferOfferGeomObject Attribute. Optionally set FieldTransferPolicy Attribute on States.
IPDv05p1	models, mediators, drivers	Advertise their import and export Fields and set the TransferOfferGeomObject Attribute. Optionally set FieldTransferPolicy Attribute on States.
IPDv05p1	connectors	Consider FieldTransferPolicy Attribute on import and export States. Advertise Fields to be transferred.
IPDv05p2	driver-internal	Optionally modify import and export States before connectors construct CplList Attribute.
IPDv05p2	models, mediators, drivers	Optionally modify import and export States before connectors construct CplList Attribute.
IPDv05p2a	connectors	Consider all connection possibilities for their CplList Attribute.
IPDv05p2b	connectors	Unambiguous construction of their CplList Attribute.
IPDv05p3	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv05p3	models, mediators, drivers	<i>unspecified/unused by NUOPC</i>
IPDv05p3	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Set the TransferActionGeomObject Attribute.
IPDv05p4	driver-internal	Realize "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv05p4	models, mediators, drivers	Realize their "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv05p4	connectors	Transfer the Grid/Mesh/LocStream objects (only DistGrid) for Field pairs that have a provider and an acceptor side.
IPDv05p5	driver-internal	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv05p5	models, mediators, drivers	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv05p5	connectors	Transfer the full Grid/Mesh/LocStream objects (with coordinates) for Field pairs that have a provider and an acceptor side.
IPDv05p6	driver-internal	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv05p6	models, mediators, drivers	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv05p6a	connectors	Reconcile the import and export States.
IPDv05p6b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv05p7	driver-internal	<i>unspecified by NUOPC</i>
IPDv05p7	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv05p8	driver-internal	<i>unspecified by NUOPC</i>
IPDv05p8	models, mediators, drivers	Handle Field data initialization. Timestamp the export Fields.
<i>A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:</i>		
Run ()	connectors	Loop over all Connectors that connect to the Component that is currently indexed by the outer loop.

IPDv05p8	models, mediators, drivers	Handle Field data initialization. Time stamp the export Fields and set the Updated and InitializedDataComplete Attributes accordingly.
<i>Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.</i>		

2.4.3 Field Pairing

The NUOPC Model and Mediator components are required to advertise their import and export Fields with a standard set of Field metadata. This set includes the `StandardName` attribute. The NUOPC Layer implements a strategy of pairing advertised Fields that is based primarily on the `StandardName` of the Fields, and in more complex situations further utilizes the `Namespace` attribute on States.

Field pairing is accomplished as part of the initialization procedure and is a collective effort of the Driver and its child components: Models, Mediator, Connectors. The exact handshakes between these components is outlined as part of the Initialize Phase Definition in section 2.4.2.

The Connectors are the most active players when it comes to Field pairing. The end result of the process is where each Connector has a list of Fields that it connects between its importState and its exportState. Each connector keeps this list in its component level metadata as `CplList` attribute.

During the first stage of Field pairing, each Connector matches all of the Fields in its importState to all of the Fields in its exportState by looking at their `StandardName` attribute. For every match a *bondLevel* is calculated and stored in the Field on the export side, i.e. on the consumer side of the connection, in the Field's `ConsumerConnection` attribute. The largest found bondLevel is kept for each Field on the export side.

The bondLevel is a measure of how strong the pairing is considering the namespace rules explained in section 2.4.4. Without the use of namespaces the bondLevel for all Field pairs that match by their `StandardName` is equal to 1.

After the first stage, there may be ambiguous Field pairs present. Ambiguous Field pairs are those that map different producer Fields (i.e. Fields in the importState of a Connector) to the *same* consumer Field (i.e. a Field in the exportState of a Connector). While the NUOPC Layer support having multiple consumer Fields connected to a single producer Field, it does not support the opposite condition. The second stage of Field pairing is responsible for disambiguating Field pairs with the same consumer Field.

Field pair disambiguation is based on the *bondLevel* that was calculated and stored on the consumer side Field for each pair during the first stage. The disambiguation rule simply selects the connection with the highest bondLevel and discards all lesser connection to the same consumer side Field. However, if the highest bondLevel is not unique, i.e. there are multiple pairs with the same bondLevel, disambiguation is not possible and an error is returned to the Driver by the Connector that finds the ambiguity first.

Assuming that the disambiguation step was successful, each Connector holds a valid `CplList` attribute with entries that correspond to the Field pairs that it is responsible for. At this stage the Driver can still overwrite this attribute and implement custom pairs if that is desired.

2.4.4 Namespaces

Namespaces are used to control and fine-tune the disambiguation of Field pairs during the initialization. The general procedure of Field pairing and disambiguation is outlined in section 2.4.3, here the use of namespaces is described.

The NUOPC Layer implements namespaces through the `Namespace` attribute on `ESMF_State` objects. The value

of this attribute is a simple character string. The NUOPC Layer automatically creates the import and export States of every Model and Mediator component that is added to a Driver. The `Namespace` attribute of these States is automatically set to the `compLabel` string that was provided during `NUOPC_DriverAdd()`. Doing this places every Field that is advertised through these States inside the component's unique namespace.

A secondary namespace can be added to a State using the `NUOPC_StateNamespaceAdd()` method. This creates a new State that is nested inside of an existing State, and sets the `Namespace` attribute of the new State. Fields that are advertised inside of such a nested State are in a namespace with two parts: `NS1:NS2`. Here `NS1` is the preset namespace of the import or export State (equal to the `compLabel`), and `NS2` is a freely chosen namespace string.

During Field pairing the namespace on each side of the connection is considered in the two part format `NS1:NS2`. The first part is equal to the `compLabel` of the corresponding component, and `NS2` is either the namespace of a nested State, or empty if the Field is not inside a nested State. Using this format, the calculation of the *bondLevel* during Field pairing is governed by the following rules:

- Namespace matching is done in a cross wise fashion, meaning `NS1` from one side is compared to `NS2` of the other side, and vice versa.
- The *bondLevel* is incremented by one counter for each cross-wise match between namespaces. (Considering that the *bondLevel* starts out as 1 for any Field pair with matching standard names, the maximum *bondLevel* that can be reached is 3.)
- Finding one side of the cross-wise comparison being an empty string is neither counted as a match nor a mis-match. The *bondLevel* remains unchanged.
- A Field pair for which a mis-match in either of the two cross-wise namespace comparisons is detected is discarded from the possible pairs. It is not further considered.

In practice then, a component that targets a specific other component with its advertised Fields would add a secondary namespace to its import or export State, and set that namespace to the `compLabel` of the targeted component. This increases the *bondLevel* for each pair from 1 to 2. An even higher *bondLevel* of 3 is achieved when both sides target each other by specifying the other component's `compLabel` through a secondary namespace.

In conclusion, namespaces can affect the *bondLevel* calculation for each pair, but they do not affect how pairs are constructed and disambiguated. In particular, the requirement for unambiguous Field pairs for each consumer Field remains unchanged, and it is an error condition if the highest *bondLevel* for a consumer Field does not correspond to a unique Field pair.

2.4.5 Connection Options

Once the field pairing discussed in the previous sections is completed, each Connector component holds an attribute by the name of `CplList`. The `CplList` is a list type attribute with as many entries as there are fields for which the Connector component is responsible for connecting. The first part of each of these entries is always the `StandardName` of the associated field. See section 2.2 for a discussion of the NUOPC field dictionary and standard names.

After the `StandardName` part, each `CplList` entry may optionally contain a string of *connection options*. Each Driver component has the chance as part of the internal `IPDv04p2` phase (see 2.4.2) to modify the `CplList` attribute of all the Connectors that it drives.

The individual connection options are colon separated, leading to the following format for each `CplList` entry:

```
StandardName[:option1[:option2[: ...]]]
```

The format of the options is:

OptionName=value1[=spec1][,value2[=spec2][, ...]]

OptionName and the value strings are case insensitive. There are single and multi-valued options as indicated in the table below. For single valued options only value1 is relevant. If the same option is listed multiple times, only the first occurrence will be used. If an option has a default value, it is indicated in the table. If a value requires additional specification via =spec then the specifications are listed in the table.

OptionName	Definition	Type	Values
dstMaskValues	List of integer values that defines the mask values.	multi	List of integers.
dumpWeights	Enable or disable dumping of the interpolation weights into a file.	single	true, false(default)
extrapDistExponent	The exponent to raise the distance to when calculating weights for the nearest_idavg extrapolation method.	single	real(default 2.0)
extrapMethod	Fill in points not mapped by the re-grid method.	single	none(default), nearest_idavg, nearest_stod, creep
extrapNumLevels	The number of levels to output for the extrapolation methods that fill levels. When a method is used that requires this, then an error will be returned, if it is not specified.	single	integer
extrapNumSrcPnts	The number of source points to use for the extrapolation methods that use more than one source point.	single	integer(default 8)
ignoreDegenerate	Ignore degenerate cells when checking the input Grids or Meshes for errors.	single	true, false(default)
pipelineDepth	Maximum number of outstanding non-blocking communication calls during the parallel interpolation. Only relevant for cases where the automatic tuning procedure fails to find a setting that works well on a given hardware.	single	integer
poleMethod	Extrapolation method around the pole(s).	single	none(default), allavg, npntavg=" <i>integer indicating number of points</i> "

remapMethod	Redistribution or interpolation to compute the regridding weights.	single	redist, bilinear(default), patch, nearest_stod, nearest_dtos, conserve
srcMaskValues	List of integer values that defines the mask values.	multi	List of integers.
srcTermProcessing	Number of terms in each partial sum of the interpolation to process on the source side. This setting impacts the bit-for-bit reproducibility of the parallel interpolation results between runs. The strictest bit-for-bit setting is achieved by setting the value to 1.	single	integer
termOrder	Order of the terms in each partial sum of the interpolation. This setting impacts the bit-for-bit reproducibility of the parallel interpolation results between runs. The strictest bit-for-bit setting is achieved by setting the value to <code>srcseq</code> .	single	free(default), srcseq, srcpet
unmappedAction	The action to take when unmapped destination elements are encountered.	single	ignore(default), error
zeroRegion	The region of destination elements set to zero before adding the result of the sparse matrix multiplication. The available options support total, selective, or no zeroing of destination elements.	single	total(default), select, empty

2.4.6 Data-Dependencies during Initialize

For multi-model applications it is not uncommon that during start-up one or more components depends on data from one or more other components. These types of data-dependencies during initialize can become very complex very quickly. Finding the "correct" sequence to initialize all components for a complex dependency graph is not trivial. The NUOPC Layer deals with this issue by repeatedly looping over all components that indicate that their initialization has data dependencies on other components. The loop is finally exited when either all components have indicated completion of their initialization, or a dead-lock situation is being detected by the NUOPC Layer.

The data-dependency resolution loop is implemented as part of Initialize Phase Definition version 2 (IPDv02) as defined in section 2.4.2. Participating components communicate their current status to the NUOPC Layer via Field and Component metadata. Participants are those components that contain an IPDv02p5 assignment in their InitializePhaseMap Attribute according to sections 2.3.1, 2.3.2, 2.3.3, and 2.3.4.

Every time a component's IPDv02p5 initialization phase is called it is responsible for setting the InitializeDataComplete and InitializeDataProgress Attributes according to its current status before

returning. For convenience, the NUOPC Layer provides a generic implementation of an IPDv02p5 phase initialize method for Models and Mediators (available as ESMF Initialize phase 5). This generic implementation takes care of setting the `InitializeDataProgress` Attribute automatically. It does so by inspecting the `Updated Field` Attribute (see section 2.3.6) on all the Fields in the component's `exportState`. The generic IPDv02p5 implementation must be specialized by attaching a method for specialization point `label_DataInitialize`. This specialization method is responsible for checking the Fields in the `importState` and for initializing any internal data structures and Fields in the `exportState`. Fields that are fully initialized in the `exportState` must be indicated by setting their `Updated` Attribute to "true". Once the component is fully initialized it must further set its `InitializeDataComplete` Attribute to "true" before returning.

During the execution of the data-dependency resolution loop the NUOPC Layer calls all of the Connectors to a Model/Mediator component before calling the component's IPDv02p5 method. Doing so ensures that all the currently available Fields are passed to the component before it tries to access them during IPDv02p5. Once a component has set its `InitializeDataComplete` Attribute to "true" it, and the Connectors to it, will no longer be called during the remainder of the resolution loop.

When *all* of the components with an IPDv02p5 initialization phase have set their `InitializeDataComplete` Attribute to "true", the NUOPC Layer successfully exits the data-dependency resolution loop. The loop is also interrupted before all `InitializeDataComplete` Attributes are set to "true" if a full cycle completes without any indicated progress. The NUOPC Layer flags this situation as a potential dead-lock and returns with error.

2.4.7 Transfer of Grid/Mesh/LocStream Objects between Components

There are modeling scenarios where the need arises to transfer physical grid information from one component to another. One common situation is that of modeling systems that utilize Mediator components to implement the interactions between Model components. In these cases the Mediator often carries out computations on a Model's native grid and performs regridding to the grid of other Model components. It is both cumbersome and error prone to recreate the Model grid in the Mediator. The Initialize Phase Definition version 3 (IPDv03) and above, defined in section 2.4.2, support the transfer of `ESMF_Grid`, and `ESMF_Mesh`, and `ESMF_LocStream` objects between Model and/or Mediator components during initialization.

The NUOPC Layer transfer protocol for `GeomObjects` (i.e. ESMF Grids, Meshes, or LocStreams) is based on two Field attributes: `TransferOfferGeomObject` and `TransferActionGeomObject`. The `TransferOfferGeomObject` attribute is used by the Model and/or Mediator components to indicate for each Field their intent for the associated `GeomObject`. The predefined values of this attribute are: "will provide", "can provide", and "cannot provide". The `TransferOfferGeomObject` attribute must be set during IPDv03p1.

The generic Connector uses the intents from both sides and constructs a response according to the table below. The response is provided by the Connector during IPDv03p2 by setting the value of the `TransferActionGeomObject` attribute to either "provide" or "accept" on each Field. Fields indicating `TransferActionGeomObject` equal to "provide" must be realized on a Grid, Mesh, or LocStream object in the Model/Mediator initialize method for phase IPDv03p3.

Fields that hold "accept" for the value of the `TransferActionGeomObject` attribute require two additional negotiation steps. By IPDv03p4 the Model/Mediator component can access the transferred Grid/Mesh/LocStream on the Fields that have the "accept" value. However, only the `DistGrid`, i.e. the decomposition and distribution information of the Grid/Mesh/LocStream is available at this stage, not the full physical grid information such as the coordinates. At this stage the Model/Mediator may modify this information by replacing the `DistGrid` object in the Grid/Mesh/LocStream. The `DistGrid` that is set on the Grid/Mesh/LocStream objects when leaving the Model/Mediator phase IPDv03p4 will consequently be used by the generic Connector to fully transfer the Grid/Mesh/LocStream object. The fully transferred objects are available on the Fields with "accept" during Model/Mediator phase IPDv03p5, where they can be used to realize the respective Field objects. Realizing typi-

cally just requires the `ESMF_FieldEmptyComplete()` call to be made. At this point all Field objects are fully realized and the initialization process can proceed as usual.

The following table shows how the generic Connector sets the `TransferActionGeomObject` attribute on the Fields according to the incoming value of `TransferOfferGeomObject`.

<code>TransferOfferGeomObject</code> Incoming side A	<code>TransferOfferGeomObject</code> Incoming side B	Outgoing setting by generic Connector
"will provide"	"will provide"	A:TransferActionGeomObject="provide" B:TransferActionGeomObject="provide"
"will provide"	"can provide"	A:TransferActionGeomObject="provide" B:TransferActionGeomObject="accept"
"will provide"	"cannot provide"	A:TransferActionGeomObject="provide" B:TransferActionGeomObject="accept"
"can provide"	"will provide"	A:TransferActionGeomObject="accept" B:TransferActionGeomObject="provide"
"can provide"	"can provide"	if (A is import side) then A:TransferActionGeomObject="provide" B:TransferActionGeomObject="accept" if (B is import side) then A:TransferActionGeomObject="accept" B:TransferActionGeomObject="provide"
"can provide"	"cannot provide"	A:TransferActionGeomObject="provide" B:TransferActionGeomObject="accept"
"cannot provide"	"will provide"	A:TransferActionGeomObject="accept" B:TransferActionGeomObject="provide"
"cannot provide"	"can provide"	A:TransferActionGeomObject="accept" B:TransferActionGeomObject="provide"
"cannot provide"	"cannot provide"	Flagged as error!

2.4.8 Field and Grid/Mesh/LocStream Reference Sharing

For coupling scenarios with a very high coupling frequency, or for situations where large data volumes are exchanged (e.g. 3D volumetric fields), it can be necessary for fields and geom objects (Grid, Mesh, and LocStreams) to share their data via references. Reference sharing greatly reduces the coupling cost compared to local or remote copies.

In the current implementation, in order for NUOPC components to be coupled via reference sharing, they must only have data defined (i.e. have DEs) on PETs that are part of both components. Further, the distribution of data across the PETs must be identical for both components. If these conditions are met, and both sides of the connection indicate that they are willing to participate in reference sharing, the NUOPC Connector will handle technical details. The Connector will provide fields to the components that reference the exact same data allocations in memory. Notice however that once reference sharing is active, the NUOPC Layer cannot protect against components violating the data access conventions. Specifically fields in the `importState` are not to be modified by the component. Reference sharing requires a higher level of "trust" between the components. NUOPC therefore requires that both sides of a connection agree to reference sharing.

A component uses the `SharePolicyField` and `SharePolicyGeomObject` attributes on each field to indicate whether it is willing to reference share the data of a field, and/or the geom object on which the field is built. A setting of `share` indicates a component's willingness to share, while `not share` indicates the opposite. The share policy attributes are automatically set when a field is advertised via the `NUOPC_Advertise()` method. By default this method sets both share policies to `not share`.

When a Connector negotiates the connections between two components, it first considers the transfer offer attributes (i.e. `TransferOfferGeomObject`) on both sides for each field to determine the `TransferActionGeomObject` attribute for both side. The details of this protocol are outline in section 2.4.7. There are two cases to consider for each field that are relevant for reference sharing:

The simple case is where the Connector determines that for a specific field both sides must provide the field and geom object. This is indicated by `TransferActionGeomObject` being set to `provide` on both sides. For this case the `ShareStatusField` and `ShareStatusGeomObject` attributes are automatically set to `not shared` for all the fields, preventing any reference sharing.

The more interesting case is where one side of the connection receives the `TransferActionGeomObject` on a field set to `provide`, while the other side receives `accept`. In this case, the next step is for the Connector to take the `SharePolicyField` and `SharePolicyGeomObject` attributes on both sides into consideration. For each of the two attributes separately, if one side indicates `not share`, both sides will receive the associated `ShareStatus` set to `not shared`. However, if both sides of the connection indicate a `SharePolicy` of `share`, the Connector must further inspect the `petLists` to see if reference sharing is possible for the specific field. Under the current implementation a field is sharable with another component if all the PETs on which the field holds DEs are also in the other component's `petList`. If this condition is not met for the specific field, then the associated `ShareStatus` is set to `not shared`. Otherwise the `ShareStatus` is set to `shared`

During later phases of the Initialization protocol the Connector performs different operations, depending on how the `TransferActionGeomObject`, `ShareStatusField`, and `ShareStatusGeomObject` attributes were set as per the above protocol:

- For a field that has `ShareStatusGeomObject` equal to `share`, the geom object provided by the provider component will be made available to the acceptor component.
- For a field that has `ShareStatusField` equal to `share`, the Connector realizes the field for the acceptor component using the data allocation reference provided by the field of the provider component.

2.4.9 Field Mirroring

In some cases it is helpful for a NUOPC component to automatically mirror or match the set of fields advertised in another component. One purpose of this is to automatically resolve the import data dependencies of a component, by setting up a component that exactly provides all of the needed fields. This is currently used in the NUOPC Component Explorer: when driving a child NUOPC Model with required import fields, the Component Explorer uses the field mirroring capability to advertise in the driver-self export State the exact set of fields advertised in the child NUOPC Model. This ensures that the entire Initialize Phase Sequence will complete (because all dependencies are satisfied) and all phases can be exercised by the Component Explorer.

The field mirror capability is also useful with NUOPC Mediators since these components often exactly reflect, in separate States, the sets of fields of each of the connected components. The field mirroring capability, therefore, can be used to ensure that a Mediator is always capable of accepting fields from connected components, and removes the need to specify field lists in multiple places, i.e., both within a set of Model components connected to a Mediator and within the Mediator itself.

The field mirror capability is supported in the Initialize Phase Definition version 5 (IPDv05) and higher, defined in section 2.4.2. During IPDv05p1, driver-self, models, mediators, and drivers advertise fields in their import and export States. At this point, these components can also optionally set the `FieldTransferPolicy` Attribute in their import and export States. The default value "TransferNone" indicates that no fields should be mirrored. The other option, "TransferAll", indicates that fields should be mirrored in the State of a connected component.

During IPDv05p1, each Connector consider the `FieldTransferPolicy` Attribute on both its import and export

States. If *both* States have a `FieldTransferPolicy` of "TransferAll", then fields are transferred between the States in both directions (i.e., import to export and export to import). The transfer process works as follows: First, the `TransferOfferGoemObject` attribute is reversed between the providing side and accepting side. Intuitively, if a field from the providing component is to be mirrored and it can provide its own geometric object, then the mirrored field on the accepting side should be set to accept a geometric object. Then, the field to be mirrored is advertised in the accepting State using a call to `NUOPC_Advertise()` such that the mirrored field shares the same Standard Name.

At this point the Initialization Sequence continues as usual. Since fields to be mirrored have been advertised with matching Standard Names, the field pairing algorithm will now match them in the usual way thereby establishing a connection between the original and mirrored fields.

2.5 Timekeeping

The NUOPC Layer associates an internal clock with three of its four generic component kinds: `NUOPC_Driver`, `NUOPC_Model`, and `NUOPC_Mediator`. The `NUOPC_Connector` is the only NUOPC component kind that does not have an internal clock object that is managed by NUOPC.

The component internal clocks are implemented as `ESMF_Clock` objects. The interaction between these clock objects between a parent component (driver) and its child components (models, mediators, and drivers) is defined by the NUOPC timekeeping behavior described below.

For a simple run sequence with only a single coupling time-step, the driver clock sets the `startTime`, `stopTime`, and `timeStep` to be the beginning, the end, and the coupling period of the run, respectively. At the beginning of executing the run sequence, the driver clock `currTime` is set to its `startTime`. As the driver component executes the run sequence, it passes its clock to each child component that it executes. At the end of each full sweep through the run sequence the driver `currTime` is incremented by `timeStep` (i.e. the coupling period). This continues until the driver clock `stopTime` has been reached, and the run is complete.

When a child component is being called during the execution of the driver run sequence, it receives the driver/parent clock. This access is read-only, and the child component is only allowed to inspect but not modify the parent clock. The child component is expected to run forward a single coupling period, i.e. one `timeStep` on the parent clock. Specifically this means that the `currTime` on the child clock must match the `currTime` on the parent clock. It then must take a single `timeStep` of the parent clock forward, using its own clock to do so. The child component can implement this forward step by taking multiple smaller advances on its own clock.

The generic NUOPC component implementation provides the following assistance to implement the above described behavior:

- During initialization of a component, its clock is set as a copy of its parent clock. Specifically the settings for `startTime`, `stopTime`, `timeStep`, and `currTime` are propagated. Alarms are not propagated.
- A component can customize aspects of its clock during initialization by using the `label_SetClock` specialization point.
- During run time, the default `label_SetRunClock` specialization checks that the `currTime` matches between child and parent clock. It further checks that the child clock can reach the parent's `currTime+timeStep`, i.e. the next coupling time, by an integral number of its own time steps. If so, the `stopTime` on the child clock is set to the parent's `currTime+timeStep`.
 - It can be useful to customize `label_SetRunClock`, e.g. if the parent uses dynamic coupling periods, or in case of a run sequence with multiple coupling periods. In these cases the component must react to the parent `timeStep` provided during execution of the run sequence. In general the `currTime` match should be implemented, followed by setting the child's `timeStep` according to the information provided

on the parent clock. Finally the `stopTime` on the child clock should be set as to return at the next coupling time determined by the parent clock.

- Once past the `label_SetRunClock` specialization, the component checks the timestamps on the fields in the import state. This is done by calling into the `label_CheckImport` specialization point. The default implementation simply checks that all import fields are at `currTime` of the child clock.
 - In more complex situations, where the interaction between different components happens with different coupling periods, it can be necessary to specialize the `label_CheckImport` of a component. For example, a component might receive fields in its import state that carry different timestamps. Consequently, `label_CheckImport` must implement a more complex relationship between the component's `currTime`, and the timestamps on each import field.
- Finally the component clock is stepped forward from `currTime` to `stopTime`, using the `timeStep` interval set in the child clock. During this loop, the `label_Advance` specialization is called for each time step. The `label_Advance` specialization is responsible for any accumulating and averaging that may be necessary.
 - In practice often the `timeStep` on the child clock is chosen to be identical to that of the parent clock. This way the `label_Advance` specialization is only called once for every coupling period. In this approach the details about potentially smaller model time steps, and associated accumulation and averaging is handled below the NUOPC cap layer of a model.
- After the `stopTime` has been reached on the child clock, the `label_TimestampExport` specialization point is called before the component returns to the parent. The default implementation simply timestamps all the fields in the export state with the `currTime` of the child clock.

2.6 Component Hierarchies

The NUOPC Layer supports component hierarchies. The key feature to support this capability is the ability for a generic `NUOPC_Driver` to add another `NUOPC_Driver` component as a child, and to drive it much like a `NUOPC_Model` component. The standard Initialize Phase Definitions (IPDs), as documented under section 2.4.2, are used to define the interaction upward and downward the hierarchy tree.

In the current implementation a driver interacts upward with its parent driver using `IPDv02`. This allows for data-dependencies during initialize to be resolved throughout the entire component hierarchy. The implementation is based on a sweep algorithm that continues up and down the hierarchy until either all data-dependencies have been resolved, or a dead-lock situation has been detected and flagged.

Along the downward direction, the interaction of a driver with its children currently supports up to `IPDv05`. This allows a driver to mirror its child components' fields, and to transfer or share geom objects and fields up the component hierarchy. All of the interactions of a driver with its child components are handled by explicit `NUOPC_Connector` instances. These instances are automatically added by the driver when needed.

The detailed behavior of a `NUOPC_Driver` component within a component hierarchy depends on the setting of the `HierarchyProtocol` attribute on the driver component itself. Section 2.3.1 lists all of the driver attributes defined by NUOPC. By default the `HierarchyProtocol` attribute is unset. For unset `HierarchyProtocol` or when set to `PushUpAllExportsAndUnsatisfiedImports`, the driver component pushes all the fields from its children exportStates into its own exportState, and all unsatisfied fields in its children importStates into its own importState. This is done using the standard Field Mirroring protocol discussed under 2.4.9. Further the driver sets the `SharePolicyGeomObject`, and `SharePolicyField` to share for all the fields it mirrors. This triggers the reference share protocol as described in section 2.4.8.

When the `HierarchyProtocol` is set to `Explorer`, the driver component still mirrors the fields from its child components' `import-` and `exportStates`, as was done for the default, however, the share policies will not be set. This protocol option is used by the NUOPC `ComponentExplorer` to connect to user provided components.

Finally, for a setting of `HierarchyProtocol` to `ConnectProvidedFields`, the driver does not modify its own `import-` and `exportState`. Instead connections are made only between fields that have been added to the driver states externally. This is useful for the situation where a `NUOPC_Driver` component is called directly via ESMF component method from a level that is outside of NUOPC. In this situation, field and/or geom object sharing must be activated explicitly if desired.

3 API

3.1 Generic Component: NUOPC_Driver

MODULE:

```
module NUOPC_Driver
```

DESCRIPTION:

Component that drives and coordinates initialization of its child components: Model, Mediator, and Connector components. For every Driver time step the same run sequence, i.e. sequence of Model, Mediator, and Connector Run methods is called. The run sequence is fully customizable. The default run sequence implements explicit time stepping.

SUPER:

```
ESMF_GridComp
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

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

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
 - Ensure that the `InitializePhaseMap` and `InternalInitializePhaseMap` attributes are set consistent with the available NUOPC Initialize Phase Definition (IPD) versions (see section 2.4.2 for a precise definition). The default implementation uses IPDv02 for `InitializePhaseMap`, and sets
 - * IPDv02p1 (NUOPC PROVIDED)
 - * IPDv02p3 (NUOPC PROVIDED)
 - * IPDv02p5 (NUOPC PROVIDED).
 - The default implementation uses IPDv05 for `InternalInitializePhaseMap`, and sets
 - * IPDv05p1 (NUOPC PROVIDED)
 - * IPDv05p2 (NUOPC PROVIDED)
 - * IPDv05p3 (NUOPC PROVIDED)
 - * IPDv05p4 (NUOPC PROVIDED)
 - * IPDv05p6 (NUOPC PROVIDED)
 - * IPDv05p8 (NUOPC PROVIDED).
- phase 1: (REQUIRED, NUOPC PROVIDED)
 - A default Initialize entry point for the higher level (e.g. application level) to initialize the Driver with a single call.
 - Internally calls into the `InitializePhaseMap`: IPDv02p1, IPDv02p3, IPDv02p5 phases in sequence.

- `InitializePhaseMap: IPDv02p1 (NUOPC PROVIDED)`
 - Allocate and initialize internal data structures.
 - If the internal clock is not yet set, set the default internal clock to be a copy of the incoming clock, but only if the incoming clock is valid.
 - *Required specialization* to set component services: `label_SetModelServices`.
 - * Call `NUOPC_DriverAddComp()` for all Model, Mediator, and Connector components to be added.
 - * Optionally replace the default clock.
 - Create States for all of the child GridComps.
 - Create Connectors to/from Driver component itself.
 - Set default run sequence.
 - Execute `Initialize phase=0` for all Model, Mediator, and Connector components. This is the method where each component is required to initialize its `InitializePhaseMap` Attribute.
 - *Optional specialization* to analyze and modify the `InitializePhaseMap` Attribute of the child components before the Driver uses it: `label_ModifyInitializePhaseMap`.
 - *Optional specialization* to set run sequence: `label_SetRunSequence`.
 - Drive the initialize sequence for the child components, compatible with up to IPDv05, as documented in section 2.4.2, through IPDv05p3.
- `InitializePhaseMap: IPDv02p3 (NUOPC PROVIDED)`
 - Continue to drive the initialize sequence for the child components, compatible with up to IPDv05, as documented in section 2.4.2, through IPDv05p7.
- `InitializePhaseMap: IPDv02p5 (NUOPC PROVIDED)`
 - Continue to drive the initialize sequence for the child components, compatible with up to IPDv05, as documented in section 2.4.2, through IPDv05p8.
- `InternalInitializePhaseMap: IPDv05p1 (NUOPC PROVIDED)`
 - Request that fields in export and import State of child components are mirrored onto the driver's own import and export States.
 - This includes transferring the associated Grid, Mesh, or LocStream objects.
- `InternalInitializePhaseMap: IPDv05p2 (NUOPC PROVIDED)`
 - Reset the request of field mirroring.
- `InternalInitializePhaseMap: IPDv05p3 (NUOPC PROVIDED)`
 - Add the `REMAPMETHOD=redist` option to all entries in `CplList` attribute on all Connectors to/from the driver itself.
 - *Optional specialization* to modify the `CplList` attribute on all of the Connectors: `label_ModifyCplLists`.
- `InternalInitializePhaseMap: IPDv05p4 (NUOPC PROVIDED)`
 - Check that all connected fields in the driver's own import and export State have a producer connection.
- `InternalInitializePhaseMap: IPDv05p6 (NUOPC PROVIDED)`
 - Complete the allocation of all the fields in the driver's own import and export State.
- `InternalInitializePhaseMap: IPDv05p8 (NUOPC PROVIDED)`

- Set the `InitializeDataComplete` consistent with the data-dependency protocol.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - If the incoming clock is valid, set the internal stop time to one time step interval on the incoming clock.
 - Drive the time stepping loop, from current time to stop time, incrementing by time step.
 - * For each time step iteration the Model and Connector components `Run()` methods are being called according to the run sequence.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - Execute the `Finalize()` methods of all Connector components in order.
 - Execute the `Finalize()` methods of all Model components in order.
 - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.
 - Destroy all Model components and their import and export states.
 - Destroy all Connector components.
 - Internal clean-up.
-

3.1.1 NUOPC_DriverAddComp - Add a GridComp child to a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverAddComp()
recursive subroutine NUOPC_DriverAddGridComp(driver, compLabel, &
  compSetServicesRoutine, compSetVMRoutine, petList, info, comp, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                :: driver
character(len=*), intent(in)        :: compLabel
interface
  recursive subroutine compSetServicesRoutine(gridcomp, rc)
    use ESMF
    implicit none
    type (ESMF_GridComp) :: gridcomp ! must not be optional
    integer, intent(out)  :: rc       ! must not be optional
  end subroutine
end interface
interface
  recursive subroutine compSetVMRoutine(gridcomp, rc)
    use ESMF
    implicit none
    type (ESMF_GridComp) :: gridcomp ! must not be optional
    integer, intent(out)  :: rc       ! must not be optional
  end subroutine
end interface
```

```

        end subroutine
    end interface
    optional                                :: compSetVMRoutine
    integer,                                intent(in),  optional :: petList(:)
    type(ESMF_Info),                        intent(in),  optional :: info
    type(ESMF_GridComp),                    intent(out), optional :: comp
    integer,                                intent(out), optional :: rc

```

DESCRIPTION:

Create and add a GridComp (i.e. Model, Mediator, or Driver) as a child component to a Driver. The component is created on the provided `petList`, or by default across all of the Driver PETs.

The specified `compSetServicesRoutine()` is called back immediately after the new child component has been created internally. Very little around the component is set up at that time (e.g. component attributes are not available). The routine should therefore be very light weight, with the sole purpose of setting the entry points of the component – typically by deriving from a generic component followed by the appropriate specializations.

If provided, the `compSetVMRoutine()` is called back before the `compSetServicesRoutine()`. This allows the child component to set aspects of its own VM, such as threading or the PE distribution among PETs.

The `compLabel` must uniquely identify the child component within the context of the Driver component.

If the `comp` argument is specified, it will reference the newly created component on return.

3.1.2 NUOPC_DriverAddComp - Add a GridComp child from shared object to a Driver

INTERFACE:

```

! Private name; call using NUOPC_DriverAddComp()
recursive subroutine NUOPC_DriverAddGridCompSO(driver, compLabel, &
    sharedObj, petList, comp, rc)

```

ARGUMENTS:

```

type(ESMF_GridComp)                                :: driver
character(len=*), intent(in)                        :: compLabel
character(len=*), intent(in), optional              :: sharedObj
integer, intent(in), optional                        :: petList(:)
type(ESMF_GridComp), intent(out), optional           :: comp
integer, intent(out), optional                       :: rc

```

DESCRIPTION:

Create and add a GridComp (i.e. Model, Mediator, or Driver) as a child component to a Driver. The component is created on the provided `petList`, or by default across all of the Driver PETs.

The `SetServices()` routine in the `sharedObj` is called back immediately after the new child component has been created internally. Very little around the component is set up at that time (e.g. component attributes are not available). The routine should therefore be very light weight, with the sole purpose of setting the entry points of the component – typically by deriving from a generic component followed by the appropriate specializations.

The `compLabel` must uniquely identify the child component within the context of the Driver component.

If the `comp` argument is specified, it will reference the newly created component on return.

3.1.3 NUOPC_DriverAddComp - Add a CplComp child to a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverAddComp()
recursive subroutine NUOPC_DriverAddCplComp(driver, srcCompLabel, &
  dstCompLabel, compSetServicesRoutine, compSetVMRoutine, petList, info, &
  comp, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: driver
character(len=*), intent(in)        :: srcCompLabel
character(len=*), intent(in)        :: dstCompLabel
interface
  recursive subroutine compSetServicesRoutine(cplcomp, rc)
    use ESMF
    implicit none
    type(ESMF_CplComp)              :: cplcomp ! must not be optional
    integer, intent(out)             :: rc      ! must not be optional
  end subroutine
end interface
interface
  recursive subroutine compSetVMRoutine(cplcomp, rc)
    use ESMF
    implicit none
    type(ESMF_CplComp)              :: cplcomp ! must not be optional
    integer, intent(out)             :: rc      ! must not be optional
  end subroutine
end interface
optional                             :: compSetVMRoutine
integer, target, intent(in), optional :: petList(:)
type(ESMF_Info), intent(in), optional :: info
type(ESMF_CplComp), intent(out), optional :: comp
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create and add a CplComp (i.e. Connector) as a child component to a Driver. The component is created on the provided `petList`, or by default across the union of PETs of the components indicated by `srcCompLabel` and `dstCompLabel`.

The specified `SetServices()` routine is called back immediately after the new child component has been created internally. Very little around the component is set up at that time (e.g. component attributes are not available). The routine should therefore be very light weight, with the sole purpose of setting the entry points of the component – typically by deriving from a generic component followed by the appropriate specializations.

The `compLabel` must uniquely identify the child component within the context of the Driver component.

If the `comp` argument is specified, it will reference the newly created component on return.

3.1.4 NUOPC_DriverAddRunElement - Add RunElement for Model, Mediator, or Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverAddRunElement()
recursive subroutine NUOPC_DriverAddRunElementMPL(driver, slot, compLabel, &
  phaseLabel, relaxedflag, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: driver
integer,          intent(in)        :: slot
character(len=*),  intent(in)        :: compLabel
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
character(len=*),  intent(in),  optional :: phaseLabel
logical,           intent(in),  optional :: relaxedflag
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Add an element associated with a Model, Mediator, or Driver component to the run sequence of the Driver. The component must have been added to the Driver, and associated with `compLabel` prior to this call.

If `phaseLabel` was not specified, the first entry in the `RunPhaseMap` attribute of the referenced component will be used to determine the run phase of the added element.

By default an error is returned if no component is associated with the specified `compLabel`. This error can be suppressed by setting `relaxedflag=.true.`, and no entry will be added to the run sequence.

The `slot` number identifies the run sequence time slot in case multiple sequences are available. Slots start counting from 1.

3.1.5 NUOPC_DriverAddRunElement - Add RunElement for Connector

INTERFACE:

```
! Private name; call using NUOPC_DriverAddRunElement()
recursive subroutine NUOPC_DriverAddRunElementCPL(driver, slot, srcCompLabel, &
  dstCompLabel, phaseLabel, relaxedflag, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: driver
integer,          intent(in)        :: slot
```

```

character(len=*), intent(in) :: srcCompLabel
character(len=*), intent(in) :: dstCompLabel
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
character(len=*), intent(in), optional :: phaseLabel
logical, intent(in), optional :: relaxedflag
integer, intent(out), optional :: rc

```

DESCRIPTION:

Add an element associated with a Connector component to the run sequence of the Driver. The component must have been added to the Driver, and associated with `srcCompLabel` and `dstCompLabel` prior to this call.

If `phaseLabel` was not specified, the first entry in the `RunPhaseMap` attribute of the referenced component will be used to determine the run phase of the added element.

By default an error is returned if no component is associated with the specified `compLabel`. This error can be suppressed by setting `relaxedflag=.true.`, and no entry will be added to the run sequence.

The `slot` number identifies the run sequence time slot in case multiple sequences are available. Slots start counting from 1.

3.1.6 NUOPC_DriverAddRunElement - Add RunElement that links to another slot

INTERFACE:

```

! Private name; call using NUOPC_DriverAddRunElement()
recursive subroutine NUOPC_DriverAddRunElementL(driver, slot, linkSlot, rc)

```

ARGUMENTS:

```

type(ESMF_GridComp) :: driver
integer, intent(in) :: slot
integer, intent(in) :: linkSlot
integer, intent(out), optional :: rc

```

DESCRIPTION:

Add an element to the run sequence of the Driver that links to the time slot indicated by `linkSlot`.

3.1.7 NUOPC_DriverEgestRunSequence - Egest the run sequence as FreeFormat

INTERFACE:

```

recursive subroutine NUOPC_DriverEgestRunSequence(driver, freeFormat, rc)

```

ARGUMENTS:

```

type (ESMF_GridComp) :: driver
type (NUOPC_FreeFormat), intent (out) :: freeFormat
integer, intent (out), optional :: rc

```

DESCRIPTION:

Egest the run sequence stored in the driver as a FreeFormat object. It is the caller's responsibility to destroy the created freeFormat object.

3.1.8 NUOPC_DriverGet - Get info from a Driver

INTERFACE:

```

! Private name; call using NUOPC_DriverGet()
recursive subroutine NUOPC_DriverGet(driver, slotCount, parentClock, rc)

```

ARGUMENTS:

```

type (ESMF_GridComp) :: driver
integer, intent (out), optional :: slotCount
type (ESMF_Clock), intent (out), optional :: parentClock
integer, intent (out), optional :: rc

```

DESCRIPTION:

Access Driver information.

3.1.9 NUOPC_DriverGetComp - Get a GridComp child from a Driver

INTERFACE:

```

! Private name; call using NUOPC_DriverGetComp()
recursive subroutine NUOPC_DriverGetGridComp(driver, compLabel, comp, petList, &
importState, exportState, relaxedflag, rc)

```

ARGUMENTS:

```

type (ESMF_GridComp) :: driver
character(len=*), intent (in) :: compLabel
type (ESMF_GridComp), intent (out), optional :: comp
integer, pointer, optional :: petList(:)
type (ESMF_State), intent (out), optional :: importState
type (ESMF_State), intent (out), optional :: exportState
logical, intent (in), optional :: relaxedflag
integer, intent (out), optional :: rc

```

DESCRIPTION:

Query the Driver for a GridComp (i.e. Model, Mediator, or Driver) child component that was added under compLabel.

If provided, the petList argument will be associated with the petList that was used to create the referenced component.

By default an error is returned if no component is associated with the specified compLabel. This error can be suppressed by setting relaxedflag=.true., and unassociated arguments will be returned.

3.1.10 NUOPC_DriverGetComp - Get a CplComp child from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
recursive subroutine NUOPC_DriverGetCplComp(driver, srcCompLabel, &
      dstCompLabel, comp, petList, relaxedflag, rc)
```

ARGUMENTS:

type (ESMF_GridComp)		:: driver
character(len=*),	intent(in)	:: srcCompLabel
character(len=*),	intent(in)	:: dstCompLabel
type (ESMF_CplComp),	intent(out), optional	:: comp
integer,	pointer, optional	:: petList(:)
logical,	intent(in), optional	:: relaxedflag
integer,	intent(out), optional	:: rc

DESCRIPTION:

Query the Driver for a CplComp (i.e. Connector) child component that was added under compLabel.

If provided, the petList argument will be associated with the petList that was used to create the referenced component.

By default an error is returned if no component is associated with the specified compLabel. This error can be suppressed by setting relaxedflag=.true., and unassociated arguments will be returned.

3.1.11 NUOPC_DriverGetComp - Get all the GridComp child components from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
recursive subroutine NUOPC_DriverGetAllGridComp(driver, compList, petLists, &
      rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: driver
type(ESMF_GridComp), pointer, optional :: compList(:)
type(ESMF_PtrInt1D), pointer, optional :: petLists(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

Get all the GridComp (i.e. Model, Mediator, or Driver) child components from a Driver. The incoming `compList` and `petLists` arguments must be unassociated. On return it becomes the responsibility of the caller to deallocate the associated `compList` and `petLists` arguments

3.1.12 NUOPC_DriverGetComp - Get all the CplComp child components from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
recursive subroutine NUOPC_DriverGetAllCplComp(driver, compList, petLists, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: driver
type(ESMF_CplComp), pointer         :: compList(:)
type(ESMF_PtrInt1D), pointer, optional :: petLists(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

Get all the CplComp (i.e. Connector) child components from a Driver. The incoming `compList` and `petLists` arguments must be unassociated. On return it becomes the responsibility of the caller to deallocate the associated `compList` and `petLists` arguments

3.1.13 NUOPC_DriverIngestRunSequence - Ingest the run sequence from FreeFormat

INTERFACE:

```
recursive subroutine NUOPC_DriverIngestRunSequence(driver, freeFormat, &
autoAddConnectors, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: driver
type(NUOPC_FreeFormat), intent(in), target :: freeFormat
logical, intent(in), optional :: autoAddConnectors
integer, intent(out), optional :: rc
```

DESCRIPTION:

Ingest the run sequence from a FreeFormat object and replace the run sequence currently held by the driver. Every line in `freeFormat` corresponds to either a component run sequence element, or is part of a time loop definition.

Component run sequence elements define the run method of a single component. The lines are interpreted sequentially, however, components will execute concurrently as long as this is not prevented by data-dependencies or overlapping `petLists`.

Each line specifies the precise run method phase for a single component instance. For model, mediator, and driver components the format is this:

```
compLabel [phaseLabel]
```

Here `compLabel` is the label by which the component instance is known to the driver. It is optionally followed a `phaseLabel` identifying a specific run phase. An example of calling the run phase of the ATM instance that contains the "fast" processes, and is labeled `fast`:

```
ATM fast
```

By default, i.e. without `phaseLabel`, the first registered run method of the component is used.

The format for connector components is different. It looks like this:

```
srcCompLabel -> dstCompLabel [connectionOptions]
```

A connector instance is uniquely known by the two components it connects, i.e. by `srcCompLabel` and `dstCompLabel`. The syntax requires that the token `->` be specified between source and destination. Optionally `connectionOptions` can be supplied using the format discussed under section 2.4.5. The connection options are set as attribute `ConnectionOptions` on the respective connector component.

An example of executing the connector instance that transfers fields from the ATM component to the OCN component, using redistribution for remapping:

```
ATM -> OCN :remapMethod=redist
```

By default `autoAddConnectors` is `.false.`, which means that all components referenced in the `freeFormat` run sequence, including connectors, must already be available as child components of the driver component. An error will be returned if this is not the case. However, when `autoAddConnectors` is set to `.true.`, connector components encountered in the run sequence that are not already present in the driver will be added automatically. The default `NUOPC_Connector` implementation is used for all automatically added connector instances.

Lines that contain a time loop definition have the general format:

```
@{timeStep|*}[:runDuration]
...
@
```

Both `timeStep` and `runDuration` are numbers in units of seconds. Time loops can be nested and concatenated.

A wildcard "*" character can be specified in place of an actual `timeStep` number. In this case the `timeStep` of the associated run clock object is set to be equal to the `timeStep` of the time loop one level up in the loop nesting hierarchy. If a wildcard time step is used for a single outer time loop in the run sequence, then the associated run clock is identical to the driver clock and must be set explicitly by the driver code, or its parent component.

The `runDuration` specification is optional. If omitted, the duration of the associated run clock is set to the `timeStep` of the time loop one level up in the loop nesting hierarchy. This ensures that for a single nested time loop, the loop returns to the parent loop level at the appropriate time.

A simple example of a single time loop with one hour timestep:

```
@3600
...
...
@
```

Each time loop has its own associated clock object. NUOPC manages these clock objects, i.e. their creation and destruction, as well as `startTime`, `endTime`, `timeStep` adjustments during the execution. The outer most time loop of the run sequence is a special case. It uses the driver clock itself. If a single outer most loop is defined in the run sequence provided by `freeFormat`, this loop becomes the driver loop level directly. Therefore, setting the `timeStep` or `runDuration` for the outer most time loop results modifying the driver clock itself. However, for cases with concatenated loops on the upper level of the run sequence in `freeFormat`, a single outer loop is added automatically during ingestion, and the driver clock is used for this loop instead.

A more complex run sequence example, that shows component run sequence elements outside of time loops, a nested time loop, time step wildcards, explicit duration specifications, and concatenated time loops:

```
@100:800
  ATM -> OCN
  OCN -> ATM
  ATM
  OCN
  @*
    OCN -> EXTOCN
    EXTOCN
  @
@
ATM -> EXTATM
EXTATM
@100:1000
  ATM -> OCN
  OCN -> ATM
  ATM
  OCN
@
```

Here the `timeStep` of the first time loop is explicitly chosen at 100s. The `runDuration` is explicitly set to 800s. The first time loop steps the current time forward for 800s, for each iteration executing ATM-OCN coupling, followed by the nested loop that calls the `OCN -> EXTOCN` and `EXTOCN` components. The nested loop uses a wildcard `timeStep` and therefore is identical to the parent loop level `timeStep` of 100s. The nested `runDuration` is not specified and therefore also defaults to the parent time step of 100s. In other words, the nested loop is executed exactly once for every parent loop iteration.

After 800s the first time loop is exited, and followed by explicit calls to ATM -> EXTAMT and EXTATM components. Finally the second time loop is entered for another 1000s runDuration. The timeStep is again explicitly set to 100s. The second time loop only implements ATM-OCN coupling, and no coupling to EXTOCN is implemented. Finally, after 1800s the sequence returns to the driver level loop.

3.1.14 NUOPC_DriverNewRunSequence - Replace the run sequence in a Driver

INTERFACE:

```
recursive subroutine NUOPC_DriverNewRunSequence(driver, slotCount, rc)
```

ARGUMENTS:

```

type (ESMF_GridComp)           :: driver
integer,          intent (in)   :: slotCount
integer,          intent (out), optional :: rc

```

DESCRIPTION:

Replace the current run sequence of the Driver with a new one that has slotCount slots. Each slot uses its own clock for time keeping.

3.1.15 NUOPC_DriverPrint - Print internal Driver information

INTERFACE:

```
recursive subroutine NUOPC_DriverPrint(driver, orderflag, rc)
```

ARGUMENTS:

```

type (ESMF_GridComp)           :: driver
logical,          intent (in), optional :: orderflag
integer,          intent (out), optional :: rc

```

DESCRIPTION:

Print internal Driver information. If orderflag is provided and set to .true., the output is ordered from lowest to highest PET. Setting this flag makes the method collective.

3.1.16 NUOPC_DriverSetRunSequence - Set internals of RunSequence slot

INTERFACE:


```
! Private name; call using NUOPC_DriverSetRunSequence()
recursive subroutine NUOPC_DriverSetRunSequence(driver, slot, clock, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                :: driver
integer,          intent (in)        :: slot
type (ESMF_Clock),  intent (in)       :: clock
integer,          intent (out), optional :: rc
```

DESCRIPTION:

Set the `clock` in the run sequence under `slot` of the Driver.

3.2 Generic Component: NUOPC_ModelBase

MODULE:

```
module NUOPC_ModelBase
```

DESCRIPTION:

Partial specialization of a component with a default *explicit* time dependency. Each time the `Run` method is called the component steps one `timeStep` forward on the passed in parent clock. The component flags incompatibility during `Run` if the current time of the incoming clock does not match the current time of the internal clock.

SUPER:

```
ESMF_GridComp
```

USE DEPENDENCIES:

```
use ESMF
```

SETS SERVICES:

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

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
 - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 for a precise definition). The default implementation sets the following mapping:
 - * IPDv00p1 = 1: (REQUIRED, IMPLEMENTOR PROVIDED)
 - * IPDv00p2 = 2: (REQUIRED, IMPLEMENTOR PROVIDED)
 - * IPDv00p3 = 3: (REQUIRED, IMPLEMENTOR PROVIDED)
 - * IPDv00p4 = 4: (REQUIRED, IMPLEMENTOR PROVIDED)

RUN:

- phase 1: (NUOPC PROVIDED)
 - SPECIALIZATION REQUIRED/PROVIDED: `label_SetRunClock` to check and set the internal Clock against the incoming Clock.
 - * IF (Phase specific specialization present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: By default check that internal Clock and incoming Clock agree on current time and that the time step of the incoming Clock is a multiple of the internal Clock time step. Under these conditions set the internal stop time to one time step interval of the incoming Clock. Otherwise exit with error, flagging an incompatibility.
 - SPECIALIZATION REQUIRED/PROVIDED: `label_CheckImport` to check Fields in the import State.
 - * IF (Phase specific specialization is present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: By default check that all import Fields are at the current time of the internal Clock.
 - Time stepping loop: starting at current time, running to stop time of the internal Clock.
 - * Timestamp the Fields in the export State according to the current time of the internal Clock.
 - * SPECIALIZATION REQUIRED: `label_Advance` to execute model or mediation code.
 - * SPECIALIZATION OPTIONAL: `label_AdvanceClock` to advance the current time of the internal Clock. By default (without specialization) advance the current time of the internal Clock according to the time step of the internal Clock.
 - SPECIALIZATION OPTIONAL: `label_TimestampExport` to timestamp Fields in the export State.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.

3.3 Generic Component: NUOPC_Model

MODULE:

```
module NUOPC_Model
```

DESCRIPTION:

Model component with a default *explicit* time dependency. Each time the `Run` method is called the model integrates one `timeStep` forward on the passed in parent clock. The internal clock is advanced at the end of each `Run` call. The component flags incompatibility during `Run` if the current time of the incoming clock does not match the current time of the internal clock.

SUPER:

```
NUOPC_ModelBase
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```

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

```

INITIALIZE:

- phase 0: Set Initialize Phase Definition Version (REQUIRED, NUOPC PROVIDED)
 - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 for a precise definition). The default implementation sets the following mapping:
 - * IPDv00p1 = 1: (REQUIRED, IMPLEMENTOR PROVIDED)
 - Advertise Fields in import and export States.
 - * IPDv00p2 = 2: (REQUIRED, IMPLEMENTOR PROVIDED)
 - Realize the advertised Fields in import and export States.
 - * IPDv00p3 = 3: (REQUIRED, NUOPC PROVIDED)
 - Check compatibility of the Fields' Connected status.
 - * IPDv00p4 = 4: (REQUIRED, NUOPC PROVIDED)
 - Handle Field data initialization. Time stamp the export Fields.
- IPDv00p1, IPDv01p1, IPDv02p1, IPDv03p1, IPDv04p1, IPDv05p1: Advertise fields in import and export States (REQUIRED, IMPLEMENTOR PROVIDED)
 - Advertise fields in import/export states using one of the two `NUOPC_Advertise` methods (3.9.3, 3.9.4). The methods require Standard Names for each field, and the Standard Names must appear in the NUOPC Field Dictionary or a runtime error is generated. `NUOPC_Advertise` accepts a `TransferOfferGeomObject` argument which may be one of:
 - * “will provide” (default) - The field will provide its own geometric object (i.e., Grid, Mesh, or LocStream)
 - * “can provide” - The field can provide its own geometric object, but only if the connected field in the other component will not provide it
 - * “cannot provide” - The field cannot provide its own geometric object. It must accept a geometric object from a connected field.

See section 2.4.7 for more details about transferring geometric objects between NUOPC components. Memory is not allocated for advertised fields, but attributes are set on the field which can be used in later phases, especially for determining if another component can provide and/or consume the advertised field.
- IPDv00p2, IPDv01p3, IPDv02p3, IPDv03p3, IPDv04p3, IPDv05p4: Realize field *providing* a geometric object (REQUIRED*, IMPLEMENTOR PROVIDED)
 - Realize connected import and export fields that have their `TransferActionGeomObject` attribute set to “provide”, i.e., that will provide their own geometric object (i.e., Grid, Mesh, or LocStream). “provide” is the default value of `TransferActionGeomObject`. Realize means an `ESMF_Field` object is created on a geometric object and memory for the field is allocated or referenced.

The `NUOPC_Realize` methods (??, 3.9.22) are used to realize fields. Only previously advertised fields can be realized and the field's name is used to search the state for the previously advertised field.

*Note: This phase is not required if all fields are *accepting* a geometric object.
- IPDv03p4, IPDv04p4, IPDv05p5: Modify decomposition of accepted geometric object (OPTIONAL, IMPLEMENTOR PROVIDED)
 - Optionally modify the decomposition information of any accepted geometric object by replacing the DistGrid. In the case of the Grid geometric object, this can be accomplished by retrieving the Grid (and its DistGrid) from the Field, creating a new DistGrid with modified decomposition, creating a new Grid on

the new (modified) DistGrid, and then using `ESMF_FieldEmptySet` to replace the existing Grid with the new one.

This phase is useful when accepting a Grid from another component, but when the PET counts differ between components. In this case, a new decomposition needs to be set based on the current processor count.

- IPDv03p5, IPDv04p5, IPDv05p6: Realize fields *accepting* a geometric object (REQUIRED*, IMPLEMENTOR PROVIDED)
 - Realize connected import and export fields that have their `TransferActionGeomObject` attribute set to “accept”, i.e., that will accept a geometric object from a connected field in another component. If the generic `NUOPC_Connector` is used, at this point the full geometric object has already been set in the field and only a call to `ESMF_FieldEmptyComplete` is required to allocate memory for the field.
 - The `NUOPC_Realize` methods (??, 3.9.22) are used to realize fields. Only previously advertised fields can be realized and the field’s name is used to search the state for the previously advertised field.
 - *Note: This phase is not required if all fields are *providing* a geometric object.
- IPDv00p3, IPDv01p4, IPDv02p4, IPDv03p6, IPDv04p6, IPDv05p7: Verify import fields connected and set clock (NUOPC PROVIDED)
 - If the model internal clock is found to be not set, then set the model internal clock as a copy of the incoming clock.
 - *Optional specialization* to set the internal clock and/or alarms: `label_SetClock`.
 - Check compatibility, ensuring all advertised import Fields are connected.
- IPDv00p4, IPDv01p5: Initialize export fields (NUOPC PROVIDED)
 - *Optional specialization* to initialize export Fields: `label_DataInitialize`
 - Time stamp Fields in export State for compatibility checking.
- IPDv02p5, IPDv03p7, IPDv04p7, IPDv05p8: Initialize export fields (NUOPC PROVIDED)
 - *Optional specialization* to initialize export Fields: `label_DataInitialize`
 - Timestamp Fields in export State for compatibility checking.
 - Set Component metadata used to resolve initialize data dependencies.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - SPECIALIZATION REQUIRED/PROVIDED: `label_SetRunClock` to check and set the internal Clock against the incoming Clock.
 - * IF (Phase specific specialization present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: By default check that internal Clock and incoming Clock agree on current time and that the time step of the incoming Clock is a multiple of the internal Clock time step. Under these conditions set the internal stop time to one time step interval of the incoming Clock. Otherwise exit with error, flagging an incompatibility.
 - SPECIALIZATION REQUIRED/PROVIDED: `label_CheckImport` to check Fields in the import State.
 - * IF (Phase specific specialization is present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: By default check that all import Fields are at the current time of the internal Clock.
 - Time stepping loop: starting at current time, running to stop time of the internal Clock.
 - * Timestamp the Fields in the export State according to the current time of the internal Clock.

- * SPECIALIZATION REQUIRED: `label_Advance` to execute model code.
- * SPECIALIZATION OPTIONAL: `label_AdvanceClock` to advance the current time of the internal Clock. By default (without specialization) advance the current time of the internal Clock according to the time step of the internal Clock.
- SPECIALIZATION OPTIONAL/PROVIDED: `label_TimestampExport` to timestamp Fields in the export State.
 - * IF (Phase specific specialization present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: Timestamp all Fields in the export State according to the current time of the internal Clock, which now is identical to the stop time of the internal Clock.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.

3.3.1 Initialize Phase Specialization - `label_SetClock`

OPTIONAL, IMPLEMENTOR PROVIDED

Called from: IPDv00p3, IPDv01p4, IPDv02p4, IPDv03p6, IPDv04p6, IPDv05p7

The specialization method can change aspects of the internal clock, which defaults to a copy of the incoming parent clock. For example, the timeStep size may be changed and/or Alarms may be set on the clock.

The method `NUOPC_CompSetClock(comp, externalClock, stabilityTimeStep)` (3.6.34) can be used to set the internal clock as a copy of `externalClock`, but with a timeStep that is less than or equal to the `stabilityTimeStep`. At the same time it ensures that the timeStep of the external clock is a multiple of the timeStep of the internal clock. If the `stabilityTimeStep` argument is not provided then the internal clock will simply be set as a copy of the external clock.

3.3.2 Initialize Phase Specialization - `label_DataInitialize`

OPTIONAL, IMPLEMENTOR PROVIDED

Called from: IPDv00p4, IPDv01p5, IPDv02p5, IPDv03p7, IPDv04p7, IPDv05p8

The specialization method should initialize field data in the export state. Fields in the export state will be timestamped automatically by the calling phase for all fields that have the “Updated” attribute set to “true”.

3.3.3 Run Phase Specialization - `label_SetRunClock`

REQUIRED, NUOPC PROVIDED

Called from: default run phase

A specialization method to check and set the internal clock against the incoming clock. This method is called by the default run phase.

If not overridden, the default method will check that the internal clock and incoming clock agree on the current time and that the time step of the incoming clock is a multiple of the internal clock time step. Under these conditions set the internal stop time to one time step interval of the incoming clock. Otherwise exit with error, flagging an incompatibility.

3.3.4 Run Phase Specialization - label_CheckImport

REQUIRED, NUOPC PROVIDED

Called from: default run phase

A specialization method to verify import fields before advancing in time. If not overridden, the default method verifies that all import fields are at the current time of the internal clock.

3.3.5 Run Phase Specialization - label_Advance

REQUIRED, IMPLEMENTOR PROVIDED

Called from: default run phase

A specialization method that advances the model forward in time by one timestep of the internal clock. This method will be called iteratively by the default run phase until reaching the stop time on the internal clock.

3.3.6 Run Phase Specialization - label_TimestampExport

REQUIRED, NUOPC PROVIDED

Called from: default run phase

A specialization method to set the timestamp on export fields after the model has advanced. If not overridden, the default method sets the timestamp on all export fields to the stop time on the internal clock (which is also now the current model time).

3.3.7 Finalize Phase Specialization - label_Finalize

OPTIONAL, IMPLEMENTOR PROVIDED

Called from: default finalize phase

An optional specialization method for custom finalization code and deallocations of user data structures.

3.3.8 NUOPC_ModelGet - Get info from a Model

INTERFACE:

```
subroutine NUOPC_ModelGet(model, driverClock, modelClock, &
    importState, exportState, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)           :: model
type(ESMF_Clock),             intent(out), optional :: driverClock
type(ESMF_Clock),             intent(out), optional :: modelClock
type(ESMF_State),             intent(out), optional :: importState
type(ESMF_State),             intent(out), optional :: exportState
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access Model information.

3.4 Generic Component: NUOPC_Mediator

MODULE:

```
module NUOPC_Mediator
```

DESCRIPTION:

Mediator component with a default *explicit* time dependency. Each time the `Run` method is called, the time stamp on the imported Fields must match the current time (on both the incoming and internal clock). Before returning, the Mediator time stamps the exported Fields with the same current time, before advancing the internal clock one `timeStep` forward.

SUPER:

```
NUOPC_ModelBase
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

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

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
 - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 for a precise definition). The default implementation sets the following mapping:
 - * IPDv00p1 = 1: (REQUIRED, IMPLEMENTOR PROVIDED)
 - Advertise Fields in import and export States.
 - * IPDv00p2 = 2: (REQUIRED, IMPLEMENTOR PROVIDED)
 - Realize the advertised Fields in import and export States.
 - * IPDv00p3 = 3: (REQUIRED, NUOPC PROVIDED)
 - Check compatibility of the Fields' Connected status.
 - * IPDv00p4 = 4: (REQUIRED, NUOPC PROVIDED)
 - Handle Field data initialization. Time stamp the export Fields.
- IPDv00p3, IPDv01p4, IPDv02p4: (NUOPC PROVIDED)
 - Set the Mediator internal clock as a copy of the incoming clock.
 - Check compatibility, ensuring all advertised import Fields are connected.
- IPDv00p4, IPDv01p5: (NUOPC PROVIDED)
 - *Optional specialization* to initialize export Fields: `label_DataInitialize`
 - Time stamp Fields in import and export States for compatibility checking.
- IPDv02p5: (NUOPC PROVIDED)

- *Optional specialization* to initialize export Fields: `label_DataInitialize`
- Time stamp Fields in export State for compatibility checking.
- Set Component metadata used to resolve initialize data dependencies.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - SPECIALIZATION REQUIRED/PROVIDED: `label_SetRunClock` to check and set the internal Clock against the incoming Clock.
 - * IF (Phase specific specialization present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: By default check that internal Clock and incoming Clock agree on current time and that the time step of the incoming Clock is a multiple of the internal Clock time step. Under these conditions set the internal stop time to one time step interval of the incoming Clock. Otherwise exit with error, flagging an incompatibility.
 - SPECIALIZATION REQUIRED/PROVIDED: `label_CheckImport` to check Fields in the import State.
 - * IF (Phase specific specialization is present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: By default check that all import Fields are at the current time of the internal Clock.
 - Time stepping loop: starting at current time, running to stop time of the internal Clock.
 - * Timestamp the Fields in the export State according to the current time of the internal Clock.
 - * SPECIALIZATION REQUIRED: `label_Advance` to execute mediation code.
 - * SPECIALIZATION OPTIONAL: `label_AdvanceClock` to advance the current time of the internal Clock. By default (without specialization) advance the current time of the internal Clock according to the time step of the internal Clock.
 - SPECIALIZATION OPTIONAL/PROVIDED: `label_TimestampExport` to timestamp Fields in the export State.
 - * IF (Phase specific specialization present): Execute the phase specific specialization.
 - * ELSE: Execute the phase independent specialization. PROVIDED: Timestamp all Fields in the export State according to the current time of the internal Clock when *entering* the RUN method.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.

3.4.1 NUOPC_MediatorGet - Get info from a Mediator

INTERFACE:

```
subroutine NUOPC_MediatorGet (mediator, driverClock, mediatorClock, &
                             importState, exportState, rc)
```

ARGUMENTS:


```

type (ESMF_GridComp)                                :: mediator
type (ESMF_Clock),      intent(out), optional :: driverClock
type (ESMF_Clock),      intent(out), optional :: mediatorClock
type (ESMF_State),      intent(out), optional :: importState
type (ESMF_State),      intent(out), optional :: exportState
integer,                intent(out), optional :: rc

```

DESCRIPTION:

Access Mediator information.

3.5 Generic Component: NUOPC_Connector

MODULE:

```
module NUOPC_Connector
```

DESCRIPTION:

Component that makes a unidirectional connection between model, mediator, and or driver components. During initialization field pairing is performed between the import and export side according to section 2.4.3, and paired fields are connected. By default the bilinear regridding method is used during Run to transfer data from the connected import Fields to the connected export Fields.

SUPER:

```
ESMF_CplComp
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```

subroutine SetServices(connector, rc)
  type (ESMF_CplComp)      :: connector
  integer, intent(out)     :: rc

```

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
 - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 04 (see section 2.4.2 for a precise definition). The default implementation sets the following mapping:
 - * IPDv04p1a = phase : (REQUIRED, NUOPC PROVIDED)
 - * IPDv04p1b = phase : (REQUIRED, NUOPC PROVIDED)
 - * IPDv04p2 = phase : (REQUIRED, NUOPC PROVIDED)
 - * IPDv04p3 = phase : (REQUIRED, NUOPC PROVIDED)
 - * IPDv04p4 = phase : (REQUIRED, NUOPC PROVIDED)
 - * IPDv04p5a = phase : (REQUIRED, NUOPC PROVIDED)
 - * IPDv04p5b = phase : (REQUIRED, NUOPC PROVIDED)

- IPDv01p1, IPDv02p1: (NUOPC PROVIDED)
 - Construct a list of matching Field pairs between import and export State based on the `StandardName` Field metadata.
 - Store this list of `StandardName` entries in the `CplList` attribute of the Connector Component metadata.
- IPDv01p2, IPDv02p2: (NUOPC PROVIDED)
 - Allocate and initialize the internal state.
 - Use the `CplList` attribute to construct `srcFields` and `dstFields` FieldBundles in the internal state that hold matched Field pairs.
 - Set the `Connected` attribute to `true` in the Field metadata for each Field that is added to the `srcFields` and `dstFields` FieldBundles.
- IPDv01p3, IPDv02p3: (NUOPC PROVIDED)
 - Use the `CplList` attribute to construct `srcFields` and `dstFields` FieldBundles in the internal state that hold matched Field pairs.
 - Set the `Connected` attribute to `true` in the Field metadata for each Field that is added to the `srcFields` and `dstFields` FieldBundles.
 - *Optional specialization* to precompute a Connector operation: `label_ComputeRouteHandle`. Simple custom implementations store the precomputed communication RouteHandle in the `rh` member of the internal state. More complex implementations use the `state` member in the internal state to store auxiliary Fields, FieldBundles, and RouteHandles.
 - By default (if `label_ComputeRouteHandle` was *not* provided) precompute the Connector RouteHandle as a bilinear Regrid operation between `srcFields` and `dstFields`, with `unmappedaction` set to `ESMF_UNMAPPEDACTION_IGNORE`. The resulting RouteHandle is stored in the `rh` member of the internal state.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - *Optional specialization* to execute a Connector operation: `label_ExecuteRouteHandle`. Simple custom implementations access the `srcFields`, `dstFields`, and `rh` members of the internal state to implement the required data transfers. More complex implementations access the `state` member in the internal state, which holds the auxiliary Fields, FieldBundles, and RouteHandles that potentially were added during the optional `label_ComputeRouteHandle` method during initialize.
 - By default (if `label_ExecuteRouteHandle` was *not* provided) execute the precomputed Connector RouteHandle between `srcFields` and `dstFields`.
 - Update the time stamp on the Fields in `dstFields` to match the time stamp on the Fields in `srcFields`.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
 - *Optional specialization* to release the custom Connector operation: `label_ReleaseRouteHandle`; or by default, if `label_ReleaseRouteHandle` was *not* provided, release the default Connector RouteHandle.
 - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.
 - Internal clean-up.

3.5.1 NUOPC_ConnectorGet - Get parameters from a Connector

INTERFACE:

```
subroutine NUOPC_ConnectorGet(connector, srcFields, dstFields, rh, state, &
    CplSet, cplSetList, srcVM, dstVM, driverClock, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                :: connector
type(ESMF_FieldBundle), intent(out), optional :: srcFields
type(ESMF_FieldBundle), intent(out), optional :: dstFields
type(ESMF_RouteHandle), intent(out), optional :: rh
type(ESMF_State),                intent(out), optional :: state
character(*),                    intent(in), optional :: CplSet
character(ESMF_MAXSTR), pointer, optional :: cplSetList(:)
type(ESMF_VM),                   intent(out), optional :: srcVM
type(ESMF_VM),                   intent(out), optional :: dstVM
type(ESMF_Clock),                intent(out), optional :: driverClock
integer,                          intent(out), optional :: rc
```

DESCRIPTION:

Get parameters from the `connector` internal state.

The Connector keeps information about the connection that it implements in its internal state. When customizing a Connector, it is often necessary to access and sometimes modify these data objects.

The arguments are:

connector The Connector component.

[srcFields] The FieldBundle under which the Connector keeps track of all connected source side fields. The order in which the fields are stored in `srcFields` is significant, as it corresponds to the order of fields in `dstFields`. Consequently, when accessing and modifying the fields inside of `srcFields`, it is important to use the `itemorderflag=ESMF_ITEMORDER_ADDORDER` option to `ESMF_FieldBundleGet()`.

[dstFields] The FieldBundle under which the Connector keeps track of all connected destination side fields. The order in which the fields are stored in `dstFields` is significant, as it corresponds to the order of fields in `srcFields`. Consequently, when accessing and modifying the fields inside of `dstFields`, it is important to use the `itemorderflag=ESMF_ITEMORDER_ADDORDER` option to `ESMF_FieldBundleGet()`.

[rh] The RouteHandle that the Connector uses to move data from `srcFields` to `dstFields`.

[state] A State object that the Connector keeps to make customization of the Connector more convenient. The generic Connector code handles creation and destruction of `state`, but does *not* access it directly for information.

[CplSet] If present, all of the returned information is specific to the specified coupling set.

[cplSetList] The list of coupling sets currently known to the Connector. This argument must enter the call *unassociated* or an error is returned. This means that the user code must explicitly call `nullify()` or use the `=>null()` syntax on the variable passed in as `cplSetList` argument. On return, the `cplSetList` argument will be associated, potentially of size zero. The responsibility for deallocation transfers to the caller.

[srcVM] The VM of the source side component.

[**dstVM**] The VM of the destination side component.

[**driverClock**] The Clock object used by the current RunSequence level to drive this component.

[**rc**] Return code; equals ESMF_SUCCESS if there are no errors.

3.5.2 NUOPC_ConnectorSet - Set parameters in a Connector

INTERFACE:

```
subroutine NUOPC_ConnectorSet(connector, srcFields, dstFields, rh, state, &
                             CplSet, srcVM, dstVM, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp)                :: connector
type (ESMF_FieldBundle), intent(in), optional :: srcFields
type (ESMF_FieldBundle), intent(in), optional :: dstFields
type (ESMF_RouteHandle), intent(in), optional :: rh
type (ESMF_State),                intent(in), optional :: state
character(*),                      intent(in), optional :: CplSet
type (ESMF_VM),                    intent(in), optional :: srcVM
type (ESMF_VM),                    intent(in), optional :: dstVM
integer,                           intent(out), optional :: rc
```

DESCRIPTION:

Set parameters in the `connector` internal state.

The Connector keeps information about the connection that it implements in its internal state. When customizing a Connector, it is often necessary to access and sometimes modify these data objects.

The arguments are:

connector The Connector component.

[**srcFields**] The FieldBundle under which the Connector keeps track of all connected source side fields. The order in which the fields are stored in `srcFields` is significant, as it corresponds to the order of fields in `dstFields`. Consequently, when setting `srcFields`, it is important to add them in the same order as for `dstFields`.

[**dstFields**] The FieldBundle under which the Connector keeps track of all connected destination side fields. The order in which the fields are stored in `dstFields` is significant, as it corresponds to the order of fields in `srcFields`. Consequently, when setting `dstFields`, it is important to add them in the same order as for `srcFields`.

[**rh**] The RouteHandle that the Connector uses to move data from `srcFields` to `dstFields`.

[**state**] A State object that the Connector keeps to make customization of the Connector more convenient. Only in very rare cases would the user want to replace the `state` that is managed by the generic Connector implementation. If `state` is set by this call, the user essentially claims ownership of the previous `state` object, and becomes responsible for its destruction. Ownership of the new `state` is transferred to the Connector and must not be explicitly destroyed by the user code.

[CplSet] If present, all of the passed in information is set under the specified coupling set.

[srcVM] The VM of the source side component.

[dstVM] The VM of the destination side component.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.6 General Generic Component Methods

3.6.1 NUOPC_GridCompAreServicesSet - Check if SetServices was called

INTERFACE:

```
! Private name; call using NUOPC_GridCompAreServicesSet()
function NUOPC_GridCompAreServicesSet(comp, rc)
```

RETURN VALUE:

```
logical :: NUOPC_GridCompAreServicesSet
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)          :: comp
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if SetServices has been called for `comp`. Otherwise return `.false..`

3.6.2 NUOPC_CplCompAreServicesSet - Check if SetServices was called

INTERFACE:

```
! Private name; call using NUOPC_CplCompAreServicesSet()
function NUOPC_CplCompAreServicesSet(comp, rc)
```

RETURN VALUE:

```
logical :: NUOPC_CplCompAreServicesSet
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in)          :: comp
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if SetServices has been called for `comp`. Otherwise return `.false..`

3.6.3 NUOPC_GridCompAttributeAdd - Add NUOPC GridComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_GridCompAttributeAdd()
subroutine NUOPC_GridCompAttributeAdd(comp, attrList, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)           :: comp
character(len=*), intent(in)   :: attrList(:)
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Add Attributes to the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

3.6.4 NUOPC_CplCompAttributeAdd - Add NUOPC CplComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CplCompAttributeAdd()
subroutine NUOPC_CplCompAttributeAdd(comp, attrList, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)           :: comp
character(len=*), intent(in)   :: attrList(:)
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Add Attributes to the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

3.6.5 NUOPC_GridCompAttributeEgest - Egest NUOPC GridComp Attributes in FreeFormat

INTERFACE:

```
! Private name; call using NUOPC_GridCompAttributeEgest()
subroutine NUOPC_GridCompAttributeEgest(comp, freeFormat, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp),      intent (in)           :: comp
type (NUOPC_FreeFormat),   intent (out)          :: freeFormat
integer,                   intent (out), optional :: rc
```

DESCRIPTION:

Egest the Attributes of the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance") as a FreeFormat object. It is the caller's responsibility to destroy the created freeFormat object.

3.6.6 NUOPC_CompAttributeEgest - Egest NUOPC CplComp Attributes in FreeFormat

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeEgest()
subroutine NUOPC_CplCompAttributeEge(comp, freeFormat, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp),      intent (in)           :: comp
type (NUOPC_FreeFormat),  intent (out)          :: freeFormat
integer,                   intent (out), optional :: rc
```

DESCRIPTION:

Egest the Attributes of the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance") as a FreeFormat object. It is the caller's responsibility to destroy the created freeFormat object.

3.6.7 NUOPC_CompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_GridCompAttributeGet(comp, name, value, isPresent, isSet, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp),      intent (in)           :: comp
character(*),               intent (in)           :: name
character(*),               intent (out)          :: value
logical,                    intent (out), optional :: isPresent
logical,                    intent (out), optional :: isSet
integer,                    intent (out), optional :: rc
```


DESCRIPTION:

Access the Attribute name inside of comp using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Unless isPresent and/or isSet are provided, return with error if the Attribute is not present or not set. isSet will be .false. for not present Attributes and not set Attributes.

3.6.8 NUOPC_CplCompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CplCompAttributeGet()
subroutine NUOPC_CplCompAttributeGet(comp, name, value, isPresent, isSet, rc)
```

ARGUMENTS:

type(ESMF_CplComp),	intent(in)	:: comp
character(*),	intent(in)	:: name
character(*),	intent(out)	:: value
logical,	intent(out), optional	:: isPresent
logical,	intent(out), optional	:: isSet
integer,	intent(out), optional	:: rc

DESCRIPTION:

Access the Attribute name inside of comp using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Unless isPresent and/or isSet are provided, return with error if the Attribute is not present or not set. isSet will be .false. for not present Attributes and not set Attributes.

3.6.9 NUOPC_GridCompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_GridCompAttributeGetI()
subroutine NUOPC_GridCompAttributeGetI(comp, name, value, isPresent, isSet, rc)
```

ARGUMENTS:

type(ESMF_GridComp),	intent(in)	:: comp
character(*),	intent(in)	:: name
integer,	intent(out)	:: value
logical,	intent(out), optional	:: isPresent
logical,	intent(out), optional	:: isSet
integer,	intent(out), optional	:: rc

DESCRIPTION:

Access the Attribute name inside of comp using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Unless isPresent and/or isSet are provided, return with error if the Attribute is not present or not set. isSet will be .false. for not present Attributes and not set Attributes.

3.6.10 NUOPC_CplCompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CplCompAttributeGet()
subroutine NUOPC_CplCompAttributeGetI(comp, name, value, isPresent, isSet, rc)
```

ARGUMENTS:

type(ESMF_CplComp),	intent(in)	:: comp
character(*),	intent(in)	:: name
integer,	intent(out)	:: value
logical,	intent(out), optional	:: isPresent
logical,	intent(out), optional	:: isSet
integer,	intent(out), optional	:: rc

DESCRIPTION:

Access the Attribute name inside of comp using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Unless isPresent and/or isSet are provided, return with error if the Attribute is not present or not set. isSet will be .false. for not present Attributes and not set Attributes.

3.6.11 NUOPC_GridCompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_GridCompAttributeGet()
subroutine NUOPC_GridCompAttributeGetSL(comp, name, valueList, isPresent, &
    isSet, itemCount, typekind, rc)
```

ARGUMENTS:

type(ESMF_GridComp),	intent(in)	:: comp
character(*),	intent(in)	:: name
character(*),	intent(out), optional	:: valueList(:)

logical,	intent(out), optional :: isPresent
logical,	intent(out), optional :: isSet
integer,	intent(out), optional :: itemCount
type(ESMF_TypeKind_Flag),	intent(out), optional :: typekind
integer,	intent(out), optional :: rc

DESCRIPTION:

Access the Attribute name inside of comp using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Unless isPresent and/or isSet are provided, return with error if the Attribute is not present or not set. isSet will be .false. for not present Attributes and not set Attributes.

3.6.12 NUOPC_CplCompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CplCompAttributeGet()
subroutine NUOPC_CplCompAttributeGetSL(comp, name, valueList, isPresent, &
    isSet, itemCount, typekind, rc)
```

ARGUMENTS:

type(ESMF_CplComp),	intent(in)	:: comp
character(*),	intent(in)	:: name
character(*),	intent(out), optional	:: valueList(:)
logical,	intent(out), optional	:: isPresent
logical,	intent(out), optional	:: isSet
integer,	intent(out), optional	:: itemCount
type(ESMF_TypeKind_Flag),	intent(out), optional	:: typekind
integer,	intent(out), optional	:: rc

DESCRIPTION:

Access the Attribute name inside of comp using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Unless isPresent and/or isSet are provided, return with error if the Attribute is not present or not set. isSet will be .false. for not present Attributes and not set Attributes.

3.6.13 NUOPC_GridCompAttributeIngest - Ingest free format NUOPC GridComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_GridCompAttributeIngest()
subroutine NUOPC_GridCompAttributeIng(comp, freeFormat, addFlag, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp),      intent (in)           :: comp
type (NUOPC_FreeFormat),   intent (in)           :: freeFormat
logical,                   intent (in), optional :: addFlag
integer,                   intent (out), optional :: rc
```

DESCRIPTION:

Ingest the Attributes from a FreeFormat object onto the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

If addFlag is `.false.` (default), an error will be returned if an attribute is to be ingested that was not previously added to the `comp` object. If addFlag is `.true.`, all missing attributes will be added by this method automatically as needed.

Each line in `freeFormat` is of this format:

```
attributeName = attributeValue
```

For example:

```
Verbosity   = 0
Profiling    = 0
Diagnostic   = 0
```

could directly be ingested as Attributes for any instance of the four standard NUOPC component kinds. This is because `Verbosity`, `Profiling`, and `Diagnostic` are pre-defined Attributes of the NUOPC component kinds according to sections 2.3.1, 2.3.2, 2.3.3, and 2.3.4.

When Attributes are specified in `freeFormat` that are not pre-defined for a specific component kind, they can still be ingested by a component instance using the `addFlag=.true.` option. For instance:

```
ModelOutputChoice = 2
```

specifies a user-level Attribute, which is not part of the pre-defined Attributes of any of the standard NUOPC component kinds.

3.6.14 NUOPC_CompAttributeIngest - Ingest free format NUOPC CplComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeIngest()
subroutine NUOPC_CplCompAttributeIngest(comp, freeFormat, addFlag, rc)
```

ARGUMENTS:

```

type (ESMF_CplComp),      intent (in)           :: comp
type (NUOPC_FreeFormat), intent (in)           :: freeFormat
logical,                  intent (in), optional :: addFlag
integer,                  intent (out), optional :: rc

```

DESCRIPTION:

Ingest the Attributes from a FreeFormat object onto the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

If addFlag is `.false.` (default), an error will be returned if an attribute is to be ingested that was not previously added to the comp object. If addFlag is `.true.`, all missing attributes will be added by this method automatically as needed.

Each line in freeFormat is of this format:

```
attributeName = attributeValue
```

For example:

```

Verbosity  = 0
Profiling  = 0
Diagnostic = 0

```

could directly be ingested as Attributes for any instance of the four standard NUOPC component kinds. This is because `Verbosity`, `Profiling`, and `Diagnostic` are pre-defined Attributes of the NUOPC component kinds according to sections 2.3.1, 2.3.2, 2.3.3, and 2.3.4.

When Attributes are specified in freeFormat that are not pre-defined for a specific component kind, they can still be ingested by a component instance using the `addFlag=.true.` option. For instance:

```
ModelOutputChoice = 2
```

specifies a user-level Attribute, which is not part of the pre-defined Attributes of any of the standard NUOPC component kinds.

3.6.15 NUOPC_CompAttributeReset - Reset NUOPC GridComp Attributes

INTERFACE:

```

! Private name; call using NUOPC_CompAttributeReset()
subroutine NUOPC_GridCompAttributeReset(comp, attrList, rc)

```

ARGUMENTS:

```

type (ESMF_GridComp)           :: comp
character(len=*), intent (in)  :: attrList(:)
integer,                  intent (out), optional :: rc

```

DESCRIPTION:

Reset Attributes on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

3.6.16 NUOPC_CompAttributeReset - Reset NUOPC CplComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeReset()
subroutine NUOPC_CplCompAttributeReset(comp, attrList, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                :: comp
character(len=*), intent(in)      :: attrList(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

Reset Attributes on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

3.6.17 NUOPC_CompAttributeSet - Set a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_GridCompAttributeSetS(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: comp
character(*), intent(in)            :: name
character(*), intent(in)            :: value
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set the Attribute name inside of comp on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.18 NUOPC_CompAttributeSet - Set a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_CplCompAttributeSetS(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)           :: comp
character(*), intent(in)     :: name
character(*), intent(in)     :: value
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the Attribute name inside of comp on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.19 NUOPC_CompAttributeSet - Set a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_GridCompAttributeSetI(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)          :: comp
character(*), intent(in)     :: name
integer,          intent(in) :: value
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the Attribute name inside of comp on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.20 NUOPC_CompAttributeSet - Set a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_CplCompAttributeSetI(comp, name, value, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp)           :: comp
character(*), intent(in)      :: name
integer,          intent(in)  :: value
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the Attribute name inside of comp on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.21 NUOPC_CompAttributeSet - Set a NUOPC GridComp List Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_GridCompAttributeSetSL(comp, name, valueList, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)           :: comp
character(*), intent(in)      :: name
character(*), intent(in)      :: valueList(:)
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the Attribute name inside of comp on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.22 NUOPC_CompAttributeSet - Set a NUOPC CplComp List Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_CplCompAttributeSetSL(comp, name, valueList, rc)
```


ARGUMENTS:

```
type (ESMF_CplComp)                :: comp
character(*), intent(in)           :: name
character(*), intent(in)           :: valueList(:)
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the Attribute name inside of comp on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.23 NUOPC_CompCheckSetClock - Check Clock compatibility and set stopTime

INTERFACE:

```
! Private name; call using NUOPC_CompCheckSetClock()
subroutine NUOPC_GridCompCheckSetClock(comp, externalClock, forceTimeStep, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp),      intent(inout)        :: comp
type (ESMF_Clock),         intent(in)           :: externalClock
logical,                   intent(in), optional :: forceTimeStep
integer,                   intent(out), optional :: rc
```

DESCRIPTION:

Compare externalClock to the internal clock of comp to make sure they match in their current time. Also ensure that the time step of the external clock is a multiple of the time step of the internal clock. If both conditions are satisfied then set the stop time of the internal clock so it is reached in one time step of the external clock. Otherwise leave the internal clock unchanged and return with error. The direction of the involved clocks is taking into account. Setting the forceTimeStep argument to .true. forces the timeStep of the externalClock to be used to reset the timeStep of the internal clock.

3.6.24 NUOPC_CompDerive - Derive a GridComp from a generic component

INTERFACE:

```
! Private name; call using NUOPC_CompDerive()
recursive subroutine NUOPC_GridCompDerive(comp, genericSetServicesRoutine, rc)
```

ARGUMENTS:

```

type(ESMF_GridComp), intent(in)          :: comp
interface
  subroutine genericSetServicesRoutine(gridcomp, rc)
    use ESMF
    implicit none
    type(ESMF_GridComp)          :: gridcomp ! must not be optional
    integer, intent(out)         :: rc       ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Derive a GridComp (i.e. Model, Mediator, or Driver) from a generic component by calling into the specified `SetServices()` routine of the generic component. This is typically the first call in the `SetServices()` routine of the specializing component, and is followed by `NUOPC_CompSpecialize()` calls.

3.6.25 NUOPC_CompDerive - Derive a CplComp from a generic component

INTERFACE:

```

! Private name; call using NUOPC_CompDerive()
recursive subroutine NUOPC_CplCompDerive(comp, genericSetServicesRoutine, rc)

```

ARGUMENTS:

```

type(ESMF_CplComp), intent(in)          :: comp
interface
  subroutine genericSetServicesRoutine(cplcomp, rc)
    use ESMF
    implicit none
    type(ESMF_CplComp)          :: cplcomp ! must not be optional
    integer, intent(out)         :: rc       ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Derive a CplComp (i.e. Connector) from a generic component by calling into the specified `SetServices()` routine of the generic component. This is typically the first call in the `SetServices()` routine of the specializing component, and is followed by `NUOPC_CompSpecialize()` calls.

3.6.26 NUOPC_CompFilterPhaseMap - Filter the Phase Map of a GridComp

INTERFACE:

```

! Private name; call using NUOPC_GridCompFilterPhaseMap()
subroutine NUOPC_GridCompFilterPhaseMap(comp, methodflag, acceptStringList, &
rc)

```

ARGUMENTS:

```

type(ESMF_GridComp)                :: comp
type(ESMF_Method_Flag), intent(in)  :: methodflag
character(len=*),      intent(in)    :: acceptStringList(:)
integer,                intent(out), optional :: rc

```

DESCRIPTION:

Filter all PhaseMap entries in a GridComp (i.e. Model, Mediator, or Driver) that do *not* match any entry in the acceptStringList.

3.6.27 NUOPC_GridCompFilterPhaseMap - Filter the Phase Map of a CplComp

INTERFACE:

```

! Private name; call using NUOPC_GridCompFilterPhaseMap()
subroutine NUOPC_CplCompFilterPhaseMap(comp, methodflag, acceptStringList, &
rc)

```

ARGUMENTS:

```

type(ESMF_CplComp)                :: comp
type(ESMF_Method_Flag), intent(in)  :: methodflag
character(len=*),      intent(in)    :: acceptStringList(:)
integer,                intent(out), optional :: rc

```

DESCRIPTION:

Filter all PhaseMap entries in a CplComp (i.e. Connector) that do *not* match any entry in the acceptStringList.

3.6.28 NUOPC_GridCompGet - Access info from GridComp

INTERFACE:

```

! Private name; call using NUOPC_GridCompGet()
subroutine NUOPC_GridCompGet(comp, name, verbosity, profiling, diagnostic, rc)

```

ARGUMENTS:

```

type (ESMF_GridComp)                                :: comp
character(len=*), intent(out), optional :: name
integer,          intent(out), optional :: verbosity
integer,          intent(out), optional :: profiling
integer,          intent(out), optional :: diagnostic
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Access information from a GridComp.

3.6.29 NUOPC_CompGet - Access info from CplComp

INTERFACE:

```

! Private name; call using NUOPC_CompGet()
subroutine NUOPC_CplCompGet(comp, name, verbosity, profiling, diagnostic, rc)

```

ARGUMENTS:

```

type (ESMF_CplComp)                                :: comp
character(len=*), intent(out), optional :: name
integer,          intent(out), optional :: verbosity
integer,          intent(out), optional :: profiling
integer,          intent(out), optional :: diagnostic
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Access information from a CplComp.

3.6.30 NUOPC_CompSearchPhaseMap - Search the Phase Map of a GridComp

INTERFACE:

```

! Private name; call using NUOPC_CompSearchPhaseMap()
subroutine NUOPC_GridCompSearchPhaseMap(comp, methodflag, internalflag, &
  phaseLabel, phaseIndex, rc)

```

ARGUMENTS:

```

type (ESMF_GridComp)                                :: comp
type (ESMF_Method_Flag), intent(in)                  :: methodflag
logical,          intent(in), optional :: internalflag
character(len=*), intent(in), optional :: phaseLabel
integer,          intent(out)           :: phaseIndex
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Search all PhaseMap entries in a GridComp (i.e. Model, Mediator, or Driver) to see if phaseLabel is found. Return the associated ESMF phaseIndex, or -1 if not found. If phaseLabel is not specified, set phaseIndex to the first entry in the PhaseMap, or -1 if there are no entries. The internalflag argument allows to search the internal phase maps of driver components. The default is internalflag=.false..

3.6.31 NUOPC_CompSearchPhaseMap - Search the Phase Map of a CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompSearchPhaseMap()
subroutine NUOPC_CplCompSearchPhaseMap(comp, methodflag, phaseLabel, &
    phaseIndex, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp)                :: comp
type (ESMF_Method_Flag), intent(in) :: methodflag
character(len=*),      intent(in), optional :: phaseLabel
integer,               intent(out)         :: phaseIndex
integer,               intent(out), optional :: rc
```

DESCRIPTION:

Search all PhaseMap entries in a CplComp (i.e. Connector) to see if phaseLabel is found. Return the associated ESMF phaseIndex, or -1 if not found. If phaseLabel is not specified, set phaseIndex to the first entry in the PhaseMap, or -1 if there are no entries.

3.6.32 NUOPC_CompSearchRevPhaseMap - Reverse Search the Phase Map of a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSearchRevPhaseMap()
subroutine NUOPC_GridCompSearchRevPhaseMap(comp, methodflag, internalflag, &
    phaseIndex, phaseLabel, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                :: comp
type (ESMF_Method_Flag), intent(in) :: methodflag
logical,               intent(in), optional :: internalflag
integer,               intent(in), optional :: phaseIndex
character(len=*),      intent(out)         :: phaseLabel
integer,               intent(out), optional :: rc
```

DESCRIPTION:

Search all PhaseMap entries in a GridComp (i.e. Model, Mediator, or Driver) to see if the ESMF phaseIndex is found. Return the associated phaseLabel, or an empty string if not found. If phaseIndex is not specified, set phaseLabel to the first entry in the PhaseMap, or an empty string if there are no entries. The internalflag argument allows to search the internal phase maps of driver components. The default is internalflag=.false..

3.6.33 NUOPC_CompSearchRevPhaseMap - Reverse Search the Phase Map of a CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompSearchRevPhaseMap()
subroutine NUOPC_CplCompSearchRevPhaseMap(comp, methodflag, phaseIndex, &
    phaseLabel, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp)                :: comp
type (ESMF_Method_Flag), intent(in) :: methodflag
integer,          intent(in), optional :: phaseIndex
character(len=*), intent(out)          :: phaseLabel
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Search all PhaseMap entries in a CplComp (i.e. Connector) to see if the ESMF phaseIndex is found. Return the associated phaseLabel, or an empty string if not found. If phaseIndex is not specified, set phaseLabel to the first entry in the PhaseMap, or an empty string if there are no entries.

3.6.34 NUOPC_CompSetClock - Initialize and set the internal Clock of a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetClock()
subroutine NUOPC_GridCompSetClock(comp, externalClock, stabilityTimeStep, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp), intent(inout) :: comp
type (ESMF_Clock),    intent(in)    :: externalClock
type (ESMF_TimeInterval), intent(in), optional :: stabilityTimeStep
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Set the component internal clock as a copy of `externalClock`, but with a `timeStep` that is less than or equal to the `stabilityTimeStep`. At the same time ensure that the `timeStep` of the external clock is a multiple of the `timeStep` of the internal clock. If the `stabilityTimeStep` argument is not provided then the internal clock will simply be set as a copy of the external clock.

3.6.35 NUOPC_CompSetEntryPoint - Set entry point for a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetEntryPoint()
subroutine NUOPC_GridCompSetEntryPoint(comp, methodflag, phaseLabelList, &
    userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                :: comp
type(ESMF_Method_Flag), intent(in) :: methodflag
character(len=*),      intent(in)  :: phaseLabelList(:)
interface
  subroutine userRoutine(gridcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type(ESMF_GridComp)      :: gridcomp      ! must not be optional
    type(ESMF_State)         :: importState    ! must not be optional
    type(ESMF_State)         :: exportState    ! must not be optional
    type(ESMF_Clock)         :: clock          ! must not be optional
    integer, intent(out)     :: rc             ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set an entry point for a GridComp (i.e. Model, Mediator, or Driver). Publish the new entry point in the correct PhaseMap component attribute.

3.6.36 NUOPC_CompSetEntryPoint - Set entry point for a CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetEntryPoint()
subroutine NUOPC_CplCompSetEntryPoint(comp, methodflag, phaseLabelList, &
    userRoutine, rc)
```

ARGUMENTS:

```
type (ESMF_CplComp)                :: comp
type (ESMF_Method_Flag), intent(in) :: methodflag
character(len=*),          intent(in) :: phaseLabelList(:)
interface
  subroutine userRoutine(cplcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type (ESMF_CplComp)          :: cplcomp      ! must not be optional
    type (ESMF_State)            :: importState  ! must not be optional
    type (ESMF_State)            :: exportState  ! must not be optional
    type (ESMF_Clock)            :: clock        ! must not be optional
    integer, intent(out)         :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set an entry point for a CplComp (i.e. Connector). Publish the new entry point in the correct PhaseMap component attribute.

3.6.37 NUOPC_CompSetInternalEntryPoint - Set internal entry point for a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetInternalEntryPoint()
subroutine NUOPC_GridCompSetIntEntryPoint(comp, methodflag, phaseLabelList, &
  userRoutine, rc)
```

ARGUMENTS:

```
type (ESMF_GridComp)                :: comp
type (ESMF_Method_Flag), intent(in) :: methodflag
character(len=*),          intent(in) :: phaseLabelList(:)
interface
  subroutine userRoutine(gridcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type (ESMF_GridComp)          :: gridcomp    ! must not be optional
    type (ESMF_State)            :: importState  ! must not be optional
    type (ESMF_State)            :: exportState  ! must not be optional
    type (ESMF_Clock)            :: clock        ! must not be optional
    integer, intent(out)         :: rc           ! must not be optional
  end subroutine
```



```

end interface
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Set an *internal* entry point for a GridComp (i.e. Driver). Only Drivers currently utilize internal entry points. Internal entry points allow user specialization on the driver level during initialization and run sequencing.

3.6.38 NUOPC_CompSetServices - Try to find and call SetServices in a shared object

INTERFACE:

```

! Private name; call using NUOPC_CompSetServices()
recursive subroutine NUOPC_GridCompSetServices(comp, sharedObj, userRc, rc)

```

ARGUMENTS:

```

type(ESMF_GridComp),      intent(inout)          :: comp
character(len=*),         intent(in), optional  :: sharedObj
integer,                  intent(out), optional :: userRc
integer,                  intent(out), optional :: rc

```

DESCRIPTION:

Try to find a routine called "SetServices" in the sharedObj file and execute the routine. An attempt is made to find a routine that is close in name to "SetServices", allowing for compiler name mangling, i.e. upper and lower case, as well as trailing underscores.

3.6.39 NUOPC_CompSpecialize - Specialize a derived GridComp

INTERFACE:

```

! Private name; call using NUOPC_CompSpecialize()
subroutine NUOPC_GridCompSpecialize(comp, specLabel, specPhaseLabel, &
    specRoutine, rc)

```

ARGUMENTS:

```

type(ESMF_GridComp)          :: comp
character(len=*), intent(in)  :: specLabel
character(len=*), intent(in), optional :: specPhaseLabel
interface
    subroutine specRoutine(gridcomp, rc)
        use ESMF

```

```

        implicit none
        type(ESMF_GridComp)          :: gridcomp ! must not be optional
        integer, intent(out)          :: rc       ! must not be optional
    end subroutine
end interface
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Specialize a derived GridComp (i.e. Model, Mediator, or Driver). If specPhaseLabel is specified, the specialization only applies to the associated phase. Otherwise the specialization applies to all phases.

3.6.40 NUOPC_CompSpecialize - Specialize a derived CplComp

INTERFACE:

```

! Private name; call using NUOPC_CompSpecialize()
subroutine NUOPC_CplCompSpecialize(comp, specLabel, specPhaseLabel, &
    specRoutine, rc)

```

ARGUMENTS:

```

type(ESMF_CplComp)          :: comp
character(len=*), intent(in) :: specLabel
character(len=*), intent(in), optional :: specPhaseLabel
interface
    subroutine specRoutine(cplcomp, rc)
        use ESMF
        implicit none
        type(ESMF_CplComp)          :: cplcomp ! must not be optional
        integer, intent(out)          :: rc       ! must not be optional
    end subroutine
end interface
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Specialize a derived CplComp (i.e. Connector). If specPhaseLabel is specified, the specialization only applies to the associated phase. Otherwise the specialization applies to all phases.

3.7 Field Dictionary Methods

3.7.1 NUOPC_FieldDictionaryAddEntry - Add an entry to the NUOPC Field dictionary

INTERFACE:

```

subroutine NUOPC_FieldDictionaryAddEntry(standardName, canonicalUnits, rc)

```

ARGUMENTS:

character(*) ,	intent(in)	:: standardName
character(*) ,	intent(in)	:: canonicalUnits
integer,	intent(out), optional	:: rc

DESCRIPTION:

Add an entry to the NUOPC Field dictionary. If necessary the dictionary is first set up.

3.7.2 NUOPC_FieldDictionaryEgest - Egest NUOPC Field dictionary into FreeFormat

INTERFACE:

```
subroutine NUOPC_FieldDictionaryEgest (freeFormat, iofmt, rc)
```

ARGUMENTS:

type (NUOPC_FreeFormat),	intent(out)	:: freeFormat
type (ESMF_IOFmt_Flag),	intent(in), optional	:: iofmt
integer,	intent(out), optional	:: rc

DESCRIPTION:

Egest the contents of the NUOPC Field dictionary into a FreeFormat object. If I/O format option `iofmt` is provided and equal to `ESMF_IOFmt_YAML`, the FreeFormat object will contain the NUOPC Field dictionary expressed in YAML format. Other values for `iofmt` are ignored and this method behaves as if the optional `iofmt` argument were missing. In such a case, `freeFormat` will contain NUOPC Field dictionary entries in the traditional format. It is the caller's responsibility to destroy the created `freeFormat` object.

3.7.3 NUOPC_FieldDictionaryGetEntry - Get information about a NUOPC Field dictionary entry

INTERFACE:

```
subroutine NUOPC_FieldDictionaryGetEntry (standardName, canonicalUnits, rc)
```

ARGUMENTS:

character(*) ,	intent(in)	:: standardName
character(*) ,	intent(out), optional	:: canonicalUnits
integer,	intent(out), optional	:: rc

DESCRIPTION:

Return the canonical units that the NUOPC Field dictionary associates with the `standardName`.

3.7.4 NUOPC_FieldDictionaryHasEntry - Check whether the NUOPC Field dictionary has a specific entry

INTERFACE:

```
function NUOPC_FieldDictionaryHasEntry(standardName, rc)
```

RETURN VALUE:

```
logical :: NUOPC_FieldDictionaryHasEntry
```

ARGUMENTS:

character(*),	intent(in)	:: standardName
integer,	intent(out), optional	:: rc

DESCRIPTION:

Return `.true.` if the NUOPC Field dictionary has an entry with the specified `standardName`, `.false.` otherwise.

3.7.5 NUOPC_FieldDictionaryMatchSyno - Check whether the NUOPC Field dictionary considers the standard names synonyms

INTERFACE:

```
function NUOPC_FieldDictionaryMatchSyno(standardName1, standardName2, rc)
```

RETURN VALUE:

```
logical :: NUOPC_FieldDictionaryMatchSyno
```

ARGUMENTS:

character(*),	intent(in)	:: standardName1
character(*),	intent(in)	:: standardName2
integer,	intent(out), optional	:: rc

DESCRIPTION:

Return `.true.` if the NUOPC Field dictionary considers `standardName1` and `standardName2` synonyms, `.false.` otherwise. Also, if `standardName1` and/or `standardName2` do not correspond to an existing dictionary entry, `.false.` will be returned.

3.7.6 NUOPC_FieldDictionarySetSyno - Set synonyms in the NUOPC Field dictionary

INTERFACE:

```
subroutine NUOPC_FieldDictionarySetSyno(standardNames, rc)
```

ARGUMENTS:

```
character(*),          intent(in)          :: standardNames(:)
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Set all of the elements of the `standardNames` argument to be considered synonyms by the field dictionary. Every element in `standardNames` must correspond to the standard name of already existing entries in the field dictionary, or else an error will be returned.

3.7.7 NUOPC_FieldDictionarySetup - Setup the default NUOPC Field dictionary

INTERFACE:

```
! Private name; call using NUOPC_FieldDictionarySetup()
subroutine NUOPC_FieldDictionarySetupDefault(rc)
```

ARGUMENTS:

```
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Setup the default NUOPC Field dictionary.

3.7.8 NUOPC_FieldDictionarySetup - Setup the NUOPC Field dictionary from file

INTERFACE:

```
! Private name; call using NUOPC_FieldDictionarySetup()
subroutine NUOPC_FieldDictionarySetupFile(fileName, rc)
```

ARGUMENTS:

```
character(len=*),      intent(in)          :: fileName
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Setup the NUOPC Field dictionary by reading its content from YAML file. If the NUOPC Field dictionary already exists, remove it and create a new one. This feature requires ESMF built with YAML support. Please see the ESMF User's Guide for details.

3.8 Free Format Methods

3.8.1 NUOPC_FreeFormatAdd - Add lines to a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatAdd(freeFormat, stringList, line, rc)
```

ARGUMENTS:

type(NUOPC_FreeFormat),	intent(inout)	:: freeFormat
character(len=*),	intent(in)	:: stringList(:)
integer,	optional, intent(in)	:: line
integer,	optional, intent(out)	:: rc

DESCRIPTION:

Add lines to a FreeFormat object. The capacity of `freeFormat` may increase during this operation. The new lines provided in `stringList` are added starting at position `line`. If `line` is greater than the current `lineCount` of `freeFormat`, blank lines are inserted to fill the gap. By default, i.e. without specifying the `line` argument, the elements in `stringList` are added to the *end* of the `freeFormat` object.

3.8.2 NUOPC_FreeFormatCreate - Create a FreeFormat object

INTERFACE:

```
! Private name; call using NUOPC_FreeFormatCreate()
function NUOPC_FreeFormatCreateDefault(freeFormat, stringList, capacity, rc)
```

RETURN VALUE:

```
type(NUOPC_FreeFormat) :: NUOPC_FreeFormatCreateDefault
```

ARGUMENTS:

type(NUOPC_FreeFormat),	optional, intent(in)	:: freeFormat
character(len=*),	optional, intent(in)	:: stringList(:)
integer,	optional, intent(in)	:: capacity
integer,	optional, intent(out)	:: rc

DESCRIPTION:

Create a new FreeFormat object, which by default is empty. If `freeFormat` is provided, then the newly created object starts as a copy of `freeFormat`. If `stringList` is provided, then it is added to the end of the newly created object. If `capacity` is provided, it is used for the *initial* creation of the newly created FreeFormat object. However, if the `freeFormat` or `stringList` arguments are present, the final capacity may be larger than specified by `capacity`.

3.8.3 NUOPC_FreeFormatCreate - Create a FreeFormat object from Config

INTERFACE:

```
! Private name; call using NUOPC_FreeFormatCreate()
function NUOPC_FreeFormatCreateRead(config, label, relaxedflag, rc)
```

RETURN VALUE:

```
type (NUOPC_FreeFormat) :: NUOPC_FreeFormatCreateRead
```

ARGUMENTS:

```
type (ESMF_Config)           :: config
character(len=*)             intent(in)      :: label
logical,                     intent(in), optional :: relaxedflag
integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Create a new FreeFormat object from ESMF_Config object. The `config` object must exist, and `label` must reference a table attribute within `config`.

By default an error is returned if `label` is not found in `config`. This error can be suppressed by setting `relaxedflag=.true.`, and an empty FreeFormat object will be returned.

3.8.4 NUOPC_FreeFormatDestroy - Destroy a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatDestroy(freeFormat, rc)
```

ARGUMENTS:

```
type (NUOPC_FreeFormat),      intent(inout) :: freeFormat
integer,                     optional, intent(out) :: rc
```

DESCRIPTION:

Destroy a FreeFormat object. All internal memory is deallocated.

3.8.5 NUOPC_FreeFormatGet - Get information from a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatGet (freeFormat, lineCount, capacity, stringList, rc)
```

ARGUMENTS:

type (NUOPC_FreeFormat),	intent (in)	:: freeFormat
integer,	optional, intent (out)	:: lineCount
integer,	optional, intent (out)	:: capacity
character (len=NUOPC_FreeFormatLen),	optional, pointer	:: stringList (:)
integer,	optional, intent (out)	:: rc

DESCRIPTION:

Get information from a FreeFormat object.

3.8.6 NUOPC_FreeFormatGetLine - Get line info from a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatGetLine (freeFormat, line, lineString, tokenCount, &  
tokenList, rc)
```

ARGUMENTS:

type (NUOPC_FreeFormat),	intent (in)	:: freeFormat
integer,	intent (in)	:: line
character (len=NUOPC_FreeFormatLen),	optional, intent (out)	:: lineString
integer,	optional, intent (out)	:: tokenCount
character (len=NUOPC_FreeFormatLen),	optional, intent (out)	:: tokenList (:)
integer,	optional, intent (out)	:: rc

DESCRIPTION:

Get information about a specific line in a FreeFormat object.

3.8.7 NUOPC_FreeFormatLog - Write a FreeFormat object to the default Log

INTERFACE:

```
subroutine NUOPC_FreeFormatLog (freeFormat, rc)
```

ARGUMENTS:

```
type (NUOPC_FreeFormat), intent (in)      :: freeFormat  
integer, optional, intent (out)          :: rc
```

DESCRIPTION:

Write a FreeFormat object to the default Log.

3.8.8 NUOPC_FreeFormatPrint - Print a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatPrint (freeFormat, rc)
```

ARGUMENTS:

```
type (NUOPC_FreeFormat), intent (in)      :: freeFormat  
integer, optional, intent (out)          :: rc
```

DESCRIPTION:

Print a FreeFormat object.

3.9 Utility Routines

3.9.1 NUOPC_AddNamespace - Add a nested state with Namespace to a State

INTERFACE:

```
subroutine NUOPC_AddNamespace (state, Namespace, nestedStateName, &  
    nestedState, rc)
```

ARGUMENTS:

```
type (ESMF_State), intent (inout)          :: state  
character (len=*) , intent (in)            :: Namespace  
character (len=*) , intent (in), optional :: nestedStateName  
type (ESMF_State), intent (out), optional :: nestedState  
integer, intent (out), optional            :: rc
```

DESCRIPTION:

Add a Namespace to `state`. Namespaces are implemented via nested states. This creates a nested state inside of `state`. The nested state is returned as `nestedState`. If provided, `nestedStateName` will be used to name the newly created nested state. The default name of the nested state is equal to `Namespace`.

The arguments are:

state The `ESMF_State` object to which the Namespace is added.

Namespace The Namespace string.

[nestedStateName] Name of the nested state. Defaults to `Namespace`.

[nestedState] Optional return of the newly created nested state.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.2 NUOPC_AddNestedState - Add a nested state to a state with NUOPC attributes

INTERFACE:

```
subroutine NUOPC_AddNestedState(state, Namespace, CplSet, nestedStateName, &
    nestedState, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)      :: state
character(len=*), intent(in), optional :: Namespace
character(len=*), intent(in), optional :: CplSet
character(len=*), intent(in), optional :: nestedStateName
type(ESMF_State), intent(out), optional :: nestedState
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Create a nested state inside of `state`. The arguments `Namespace` and `tt CplSet` are used to set NUOPC attributes on the newly created state. The nested state is returned as `nestedState`. If provided, `nestedStateName` will be used to name the newly created nested state. The default name of the nested state is equal to `Namespace_CplSet`, `Namespace`, or `CplSet` if the arguments are provided.

The arguments are:

state The `ESMF_State` object to which the namespace is added.

[Namespace] Optional The Namespace string. Defaults to `"__UNSPECIFIED__"`.

[CplSet] Optional The CplSet string. Defaults to `"__UNSPECIFIED__"`.

[nestedStateName] Name of the nested state. Defaults to `Namespace_CplSet`, `Namespace`, or `CplSet` if arguments are provided.

[**nestedState**] Optional return of the newly created nested state.

[**rc**] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.3 NUOPC_Advertise - Advertise a single Field in a State

INTERFACE:

```
! Private name; call using NUOPC_Advertise()
subroutine NUOPC_AdvertiseField(state, StandardName, Units, &
    LongName, ShortName, name, TransferOfferGeomObject, SharePolicyField, &
    SharePolicyGeomObject, vm, field, rc)
```

ARGUMENTS:

type (ESMF_State),	intent (inout)	:: state
character (*),	intent (in)	:: StandardName
character (*),	intent (in), optional	:: Units
character (*),	intent (in), optional	:: LongName
character (*),	intent (in), optional	:: ShortName
character (*),	intent (in), optional	:: name
character (*),	intent (in), optional	:: TransferOfferGeomObject
character (*),	intent (in), optional	:: SharePolicyField
character (*),	intent (in), optional	:: SharePolicyGeomObject
type (ESMF_VM),	intent (in), optional	:: vm
type (ESMF_Field),	intent (out), optional	:: field
integer,	intent (out), optional	:: rc

DESCRIPTION:

Advertise a field in a state. This creates an empty field and adds it to `state`. The "StandardName", "Units", "LongName", "ShortName", and "TransferOfferGeomObject" attributes of the field are set according to the provided input..

The call checks the provided information against the NUOPC Field Dictionary to ensure correctness. Defaults are set according to the NUOPC Field Dictionary.

The arguments are:

state The ESMF_State object through which the field is advertised.

StandardName The "StandardName" attribute of the advertised field. Must be a StandardName found in the NUOPC Field Dictionary.

NOTE that if by below default rules, StandardName is also used as the input for name, then it must not contain the slash ("/") character.

[**Units**] The "Units" attribute of the advertised field. Must be convertible to the canonical units specified in the NUOPC Field Dictionary for the specified StandardName. (Currently this is restricted to be identical to the canonical units specified in the NUOPC Field Dictionary.) If omitted, the default is to use the canonical units associated with the StandardName in the NUOPC Field Dictionary.

[**LongName**] The "LongName" attribute of the advertised field. NUOPC does not restrict the value of this attribute. If omitted, the default is to use the StandardName.

[ShortName] The "ShortName" attribute of the advertised field. NUOPC does not restrict the value of this attribute. If omitted, the default is to use the StandardName.

NOTE that if by below default rules, ShortName is also used as the input for name, then it must not contain the slash ("/") character.

[name] The actual name of the advertised field by which it is accessed in the state object. The string provided for name must not contain the slash ("/") character. If omitted, the default is to use the value of the ShortName.

[TransferOfferGeomObject] If the state intent of state is ESMF_STATEINTENT_EXPORT, the "ProducerTransferOffer" attribute of the advertised field is set. If the state intent of state is ESMF_STATEINTENT_IMPORT, the "ConsumerTransferOffer" attribute of the advertised field is set. NUOPC controls the vocabulary of this attribute. Valid options are "will provide", "can provide", "cannot provide". If omitted, the default is "will provide".

[SharePolicyField] The "SharePolicyField" attribute of the advertised field. NUOPC controls the vocabulary of this attribute. Valid options are "share", and "not share". If omitted, the default is "not share".

[SharePolicyGeomObject] The "SharePolicyGeomObject" attribute of the advertised field. NUOPC controls the vocabulary of this attribute. Valid options are "share", and "not share". If omitted, the default is equal to SharePolicyField.

[vm] If present, the Field object used during advertising is created on the specified ESMF_VM object. The default is to create the Field object on the VM of the current component context.

[field] Returns the empty field object that was used to advertise.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.4 NUOPC_Advertise - Advertise a list of Fields in a State

INTERFACE:

```
! Private name; call using NUOPC_Advertise()
subroutine NUOPC_AdvertiseFields(state, StandardNames, &
    TransferOfferGeomObject, SharePolicyField, SharePolicyGeomObject, rc)
```

ARGUMENTS:

```
type (ESMF_State), intent (inout)      :: state
character(*),      intent (in)         :: StandardNames(:)
character(*),      intent (in), optional :: TransferOfferGeomObject
character(*),      intent (in), optional :: SharePolicyField
character(*),      intent (in), optional :: SharePolicyGeomObject
integer,           intent (out), optional :: rc
```

DESCRIPTION:

Advertise a list of fields in a state. This creates a list of empty fields and adds it to the state. The "StandardName", "TransferOfferGeomObject", "SharePolicyField", and "SharePolicyGeomObject" attributes of all the fields are set according to the provided input. The "Units", "LongName", and "ShortName" attributes for each field are set according to the defaults documented under method 3.9.3

The call checks the provided information against the NUOPC Field Dictionary to ensure correctness.

The arguments are:

state The `ESMF_State` object through which the fields are advertised.

StandardNames A list of "StandardName" attributes of the advertised fields. Must be StandardNames found in the NUOPC Field Dictionary.

[TransferOfferGeomObject] The "TransferOfferGeomObject" attribute of the advertised fields. This setting applies to all the fields advertised in this call. NUOPC controls the vocabulary of this attribute. Valid options are "will provide", "can provide", "cannot provide". If omitted, the default is "will provide".

[SharePolicyField] The "SharePolicyField" attribute of the advertised fields. This setting applies to all the fields advertised in this call. NUOPC controls the vocabulary of this attribute. Valid options are "share", and "not share". If omitted, the default is "not share".

[SharePolicyGeomObject] The "SharePolicyGeomObject" attribute of the advertised fields. This setting applies to all the fields advertised in this call. NUOPC controls the vocabulary of this attribute. Valid options are "share", and "not share". If omitted, the default is equal to `SharePolicyField`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.5 NUOPC_AdjustClock - Adjust the timestep in a clock

INTERFACE:

```
subroutine NUOPC_AdjustClock(clock, maxTimestep, rc)
```

ARGUMENTS:

```
type(ESMF_Clock)                :: clock
type(ESMF_TimeInterval), intent(in), optional :: maxTimestep
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Adjust the `clock` to have a potentially smaller timestep. The timestep on the incoming `clock` object is compared to the `maxTimestep`, and reset to the smaller of the two.

The arguments are:

clock The clock to be adjusted.

[maxTimestep] Upper bound of the timestep allowed in `clock`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.6 NUOPC_CheckSetClock - Check a Clock for compatibility and set its values

INTERFACE:

```
subroutine NUOPC_CheckSetClock(setClock, checkClock, setStartTimeToCurrent, &  
    currTime, forceCurrTime, forceTimeStep, rc)
```

ARGUMENTS:

type(ESMF_Clock),	intent(inout)	:: setClock
type(ESMF_Clock),	intent(in)	:: checkClock
logical,	intent(in), optional	:: setStartTimeToCurrent
type(ESMF_Time),	intent(in), optional	:: currTime
logical,	intent(in), optional	:: forceCurrTime
logical,	intent(in), optional	:: forceTimeStep
integer,	intent(out), optional	:: rc

DESCRIPTION:

By default compare `setClock` to `checkClock` to ensure they match in their current time. Further ensure that the `timeStep` of `checkClock` is a multiple of the `timeStep` of `setClock`. If both conditions are satisfied then the `stopTime` of the `setClock` is set one `checkClock` `timeStep`, or `setClock` `runDuration`, ahead of the current time, which ever is shorter. The direction of `checkClock` is considered when setting the `stopTime`.

By default the `startTime` of the `setClock` is not modified. However, if `setStartTimeToCurrent` == `.true.` the `startTime` of `setClock` is set to the `currentTime` of `checkClock`.

The arguments are:

setClock The ESMF_Clock object to be checked and set.

checkClock The reference clock object.

[setStartTimeToCurrent] If `.true.` then also set the `startTime` in `setClock` according to the `startTime` in `checkClock`. The default is `.false.`

[currTime] If provided, use `currTime` instead of `checkClock` when checking or setting the current time of `setClock`.

[forceCurrTime] If `.true.` then do *not* check the current time of the `setClock`, but instead force it to align with the `checkClock`, or `currTime`, if it was provided. The default is `.false.`

[forceTimeStep] If `.true.` then do *not* use the `timeStep` of the `setClock` to check if the next increment on the `checkClock` can be reached in an integer number of steps. Instead set the `timeStep` of the `setClock` to the `timeStep` of the `checkClock`. The default is `.false.`

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.7 NUOPC_GetAttribute - Get the value of a NUOPC Field Attribute

INTERFACE:

```
! Private name; call using NUOPC_GetAttribute()
subroutine NUOPC_GetAttributeFieldVal(field, name, value, isPresent, isSet, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)           :: field
character(*),      intent(in)           :: name
character(*),      intent(out)          :: value
logical,           intent(out), optional :: isPresent
logical,           intent(out), optional :: isSet
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Access the attribute name inside of `field` using the convention `NUOPC` and `purpose` Instance.

Unless `isPresent` and `isSet` are provided, return with error if the Attribute is not present or not set, respectively.

The arguments are:

field The `ESMF_Field` object to be queried.

name The name of the queried attribute.

value The value of the queried attribute.

[isPresent] Set to `.true.` if the queried attribute is present, `.false.` otherwise.

[isSet] Set to `.true.` if the queried attribute is set, `.false.` otherwise.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.8 NUOPC_GetAttribute - Get the typekind of a NUOPC Field Attribute

INTERFACE:

```
! Private name; call using NUOPC_GetAttribute()
subroutine NUOPC_GetAttributeFieldTK(field, name, isPresent, isSet, &
  itemCount, typekind, rc)
```

ARGUMENTS:

```
type(ESMF_Field),      intent(in)           :: field
character(*),          intent(in)           :: name
logical,                intent(out), optional :: isPresent
logical,                intent(out), optional :: isSet
integer,                intent(out), optional :: itemCount
type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Query the `typekind` of the attribute name inside of `field` using the convention `NUOPC` and purpose Instance. Unless `isPresent` is provided, return with error if the Attribute is not present.

The arguments are:

field The `ESMF_Field` object to be queried.

name The name of the queried attribute.

[isPresent] Set to `.true.` if the queried attribute is present, `.false.` otherwise.

[isSet] Set to `.true.` if the queried attribute is set, `.false.` otherwise.

[itemCount] Number of items in the attribute. Return 0 if not present or not set.

[typekind] The typekind of the queried attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.9 NUOPC_GetAttribute - Get the value of a NUOPC State Attribute

INTERFACE:

```
! Private name; call using NUOPC_GetAttribute()
subroutine NUOPC_GetAttributeState(state, name, value, isPresent, isSet, &
    itemCount, typekind, rc)
```

ARGUMENTS:

<code>type(ESMF_State),</code>	<code>intent(in)</code>	<code>:: state</code>
<code>character(*),</code>	<code>intent(in)</code>	<code>:: name</code>
<code>character(*),</code>	<code>intent(out), optional</code>	<code>:: value</code>
<code>logical,</code>	<code>intent(out), optional</code>	<code>:: isPresent</code>
<code>logical,</code>	<code>intent(out), optional</code>	<code>:: isSet</code>
<code>integer,</code>	<code>intent(out), optional</code>	<code>:: itemCount</code>
<code>type(ESMF_TypeKind_Flag),</code>	<code>intent(out), optional</code>	<code>:: typekind</code>
<code>integer,</code>	<code>intent(out), optional</code>	<code>:: rc</code>

DESCRIPTION:

Access the attribute name inside of `state` using the convention `NUOPC` and purpose Instance. Returns with error if the attribute is not present or not set.

The arguments are:

state The `ESMF_State` object to be queried.

name The name of the queried attribute.

[**value**] The value of the queried attribute.

[**isPresent**] Set to `.true.` if the queried attribute is present, `.false.` otherwise.

[**isSet**] Set to `.true.` if the queried attribute is set, `.false.` otherwise.

[**itemCount**] Number of items in the attribute. Return 0 if not present or not set.

[**typekind**] The typekind of the queried attribute.

[**rc**] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.10 NUOPC_GetStateMemberLists - Build lists of information of State members

INTERFACE:

```
recursive subroutine NUOPC_GetStateMemberLists(state, StandardNameList, &
  ConnectedList, NamespaceList, CplSetList, itemNameList, fieldList, &
  stateList, nestedFlag, rc)
```

ARGUMENTS:

<code>type(ESMF_State),</code>	<code>intent(in)</code>	<code>:: state</code>
<code>character(ESMF_MAXSTR),</code>	<code>pointer, optional</code>	<code>:: StandardNameList(:)</code>
<code>character(ESMF_MAXSTR),</code>	<code>pointer, optional</code>	<code>:: ConnectedList(:)</code>
<code>character(ESMF_MAXSTR),</code>	<code>pointer, optional</code>	<code>:: NamespaceList(:)</code>
<code>character(ESMF_MAXSTR),</code>	<code>pointer, optional</code>	<code>:: CplSetList(:)</code>
<code>character(ESMF_MAXSTR),</code>	<code>pointer, optional</code>	<code>:: itemNameList(:)</code>
<code>type(ESMF_Field),</code>	<code>pointer, optional</code>	<code>:: fieldList(:)</code>
<code>type(ESMF_State),</code>	<code>pointer, optional</code>	<code>:: stateList(:)</code>
<code>logical,</code>	<code>intent(in), optional</code>	<code>:: nestedFlag</code>
<code>integer,</code>	<code>intent(out), optional</code>	<code>:: rc</code>

DESCRIPTION:

Construct lists containing the StandardNames, field names, and connected status of the fields in `state`. Return this information in the list arguments. Recursively parse through nested States.

All pointer arguments present must enter this method unassociated. This means that the user code must explicitly call `nullify()` or use the `=> null()` syntax on the variables passed in as any of the pointer arguments. On return, the pointer arguments may either be unassociated or associated. Consequently the user code must first check the status of any of the returned pointer arguments via the `associated()` intrinsic before accessing the argument. The responsibility for deallocation of associated pointer arguments transfers to the caller.

The arguments are:

state The `ESMF_State` object to be queried.

[**StandardNameList**] If present, return a list of the "StandardName" attribute of each member. See the note about pointer arguments in the description section above for correct usage.

- [ConnectedList]** If present, return a list of the "Connected" attribute of each member. See the note about pointer arguments in the description section above for correct usage.
- [NamespaceList]** If present, return a list of the "Namespace" attribute of each member. See the note about pointer arguments in the description section above for correct usage.
- [CplSetList]** If present, return a list of the "CplSet" attribute of each member. See the note about pointer arguments in the description section above for correct usage.
- [itemNameList]** If present, return a list of each member name. See the note about pointer arguments in the description section above for correct usage.
- [fieldList]** If present, return a list of the member fields. See the note about pointer arguments in the description section above for correct usage.
- [stateList]** If present, return a list of the states corresponding to the owner of the fields returned under `fieldList`. See the note about pointer arguments in the description section above for correct usage.
- [nestedFlag]** When set to `.true.`, returns information from nested States (default). When set to `.false.`, returns information at the current State level only.
- [rc]** Return code; equals `ESMF_SUCCESS` if there are no errors.
-

3.9.11 NUOPC_GetTimestamp - Get the timestamp of a Field

INTERFACE:

```
subroutine NUOPC_GetTimestamp(field, isValid, time, rc)
```

ARGUMENTS:

```

type(ESMF_Field), intent(in)           :: field
logical,          intent(out), optional :: isValid
type(ESMF_Time),  intent(out), optional :: time
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Access the timestamp on `field` in form of an `ESMF_Time` object.

The arguments are:

field The `ESMF_Field` object to be checked.

[isValid] Set to `.true.` if the timestamp is valid, `.false.` otherwise.

[time] The timestamp as `ESMF_Time` object.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.12 NUOPC_IsAtTime - Check if a Field is at the given Time

INTERFACE:

```
! Private name; call using NUOPC_IsAtTime()
function NUOPC_IsAtTimeField(field, time, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsAtTimeField
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)           :: field
type(ESMF_Time),   intent(in)           :: time
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Returns `.true.` if `field` has a timestamp attribute that matches `time`. Otherwise returns `.false.`. On PETs with only a proxy instance of the field, `.true.` is returned regardless of the actual timestamp attribute.

The arguments are:

field The `ESMF_Field` object to be checked.

time The time to compare against.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.13 NUOPC_IsAtTime - Check if Field(s) in a State are at the given Time

INTERFACE:

```
! Private name; call using NUOPC_IsAtTime()
function NUOPC_IsAtTimeState(state, time, fieldName, count, fieldList, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsAtTimeState
```

ARGUMENTS:

```
type(ESMF_State),           intent(in)           :: state
type(ESMF_Time),           intent(in)           :: time
character(*),              intent(in), optional :: fieldName
integer,                   intent(out), optional :: count
type(ESMF_Field), allocatable, intent(out), optional :: fieldList(:)
integer,                   intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the field(s) in `state` have a timestamp attribute that matches `time`. Otherwise return `.false.`

The arguments are:

state The `ESMF_State` object to be checked.

time The time to compare against.

[fieldName] The name of the field in `state` to be checked. If provided, and the state does not contain a field with `fieldName`, return an error in `rc`. If not provided, check *all* the fields contained in `state` and return `.true.` if all the fields are at the correct time.

[count] If provided, the number of fields that are at `time` are returned. If `fieldName` is present then `count` cannot be greater than 1.

[fieldList] If provided, the fields that are *not* at `time` are returned. If `fieldName` is present then `fieldList` can contain a maximum of 1 field.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.14 NUOPC_IsConnected - Check if a Field is connected

INTERFACE:

```
! Private name; call using NUOPC_IsConnected()
function NUOPC_IsConnectedField(field, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsConnectedField
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)           :: field
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the `field` is connected. Otherwise return `.false.`

The arguments are:

field The `ESMF_Field` object to be checked.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.15 NUOPC_IsConnected - Check if Field(s) in a State are connected

INTERFACE:

```
! Private name; call using NUOPC_IsConnected()
function NUOPC_IsConnectedState(state, fieldName, count, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsConnectedState
```

ARGUMENTS:

```
type(ESMF_State), intent(in)           :: state
character(*),      intent(in), optional :: fieldName
integer,           intent(out), optional :: count
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the field(s) in `state` are connected. Otherwise return `.false.`.

The arguments are:

state The `ESMF_State` object to be checked.

[fieldName] The name of the field in `state` to be checked. If provided, and the state does not contain a field with `fieldName`, return an error in `rc`. If not provided, check *all* the fields contained in `state` and return `.true.` if all the fields are connected.

[count] If provided, the number of fields that are connected are returned. If `fieldName` is present then `count` cannot be greater than 1.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.16 NUOPC_IsUpdated - Check if a Field is marked as updated

INTERFACE:

```
! Private name; call using NUOPC_IsUpdated()
function NUOPC_IsUpdatedField(field, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsUpdatedField
```

ARGUMENTS:

```

type(ESMF_Field), intent(in)           :: field
integer,          intent(out), optional :: rc

```

DESCRIPTION:

Return `.true.` if the field has its "Updated" attribute set to "true". Otherwise return `.false.`

The arguments are:

field The ESMF_Field object to be checked.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.17 NUOPC_IsUpdated - Check if Field(s) in a State are marked as updated

INTERFACE:

```

! Private name; call using NUOPC_IsUpdated()
function NUOPC_IsUpdatedState(state, fieldName, count, rc)

```

RETURN VALUE:

```

logical :: NUOPC_IsUpdatedState

```

ARGUMENTS:

```

type(ESMF_State), intent(in)           :: state
character(*),      intent(in), optional :: fieldName
integer,           intent(out), optional :: count
integer,           intent(out), optional :: rc

```

DESCRIPTION:

Return `.true.` if the field(s) in state have the "Updated" attribute set to "true". Otherwise return `.false.`

The arguments are:

state The ESMF_State object to be checked.

[fieldName] The name of the field in state to be checked. If provided, and the state does not contain a field with `fieldName`, return an error in `rc`. If not provided, check *all* the fields contained in state and return `.true.` if all the fields are updated.

[count] If provided, the number of fields that are updated are returned. If `fieldName` is present then `count` cannot be greater than 1.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.18 NUOPC_NoOp - No-Operation attachable method for GridComp

INTERFACE:

```
subroutine NUOPC_NoOp(gcomp, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)    :: gcomp
integer, intent(out)    :: rc
```

DESCRIPTION:

No-Op method with an interface that matches the requirements for a attachable method for ESMF_GridComp objects.

The arguments are:

gcomp The ESMF_GridComp object to which this method is attached.

rc Return code; equals ESMF_SUCCESS if there are no errors.

3.9.19 NUOPC_Realize - Realize previously advertised Fields inside a State on a single Grid with internal allocation

INTERFACE:

```
! Private name; call using NUOPC_Realize()
subroutine NUOPC_RealizeCompleteG(state, grid, fieldName, typekind, &
    staggerloc, selection, dataFillScheme, rc)
```

ARGUMENTS:

```
type(ESMF_State)                :: state
type(ESMF_Grid),                intent(in)    :: grid
character(*),                   intent(in), optional :: fieldName
type(ESMF_TypeKind_Flag),       intent(in), optional :: typekind
type(ESMF_StaggerLoc),          intent(in), optional :: staggerloc
character(len=*),               intent(in), optional :: selection
character(len=*),               intent(in), optional :: dataFillScheme
integer,                        intent(out), optional :: rc
```

DESCRIPTION:

Realize or remove fields inside of state according to selection. All of the fields that are realized are created internally on the same grid object, allocating memory for as many field dimensions as there are grid dimensions.

The type and kind of the created fields is according to argument typekind.

Realized fields are filled with data according to the dataFillScheme argument.

The arguments are:

state The ESMF_State object in which the fields are realized.

grid The ESMF_Grid object on which to realize the fields.

[fieldName] The name of the field in `state` to be realized, or removed, according to `selection`. If provided, and the state does not contain a field with name `fieldName`, return an error in `rc`. If not provided, realize *all* the fields contained in `state` according to `selection`.

[typekind] The typekind of the internally created field(s). The valid options are ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8, ESMF_TYPEKIND_R4, and ESMF_TYPEKIND_R8. By default use the typekind of the partially created field used during advertise, or ESMF_TYPEKIND_R8, if the advertised field did not have a typekind defined.

[staggerloc] Stagger location of data in grid cells. By default use the same stagger location as the advertising field, or ESMF_STAGGERLOC_CENTER if the advertising field was created empty.

[selection] Selection of mode of operation:

- "realize_all" (default)
- "realize_connected_remove_others"
- "realize_connected+provide_remove_others"

[dataFillScheme] Realized fields will be filled according to the selected fill scheme. See ?? for fill schemes. Default is to leave the data in realized fields uninitialized.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.20 NUOPC_Realize - Realize previously advertised Fields inside a State on a single LocStream with internal allocation

INTERFACE:

```
! Private name; call using NUOPC_Realize()
subroutine NUOPC_RealizeCompleteLS(state, locstream, fieldName, typekind, selection, &
    dataFillScheme, rc)
```

ARGUMENTS:

```
type (ESMF_State)                :: state
type (ESMF_LocStream),          intent (in)    :: locstream
character(*),                   intent (in), optional :: fieldName
type (ESMF_TypeKind_Flag),      intent (in), optional :: typekind
character(len=*),               intent (in), optional :: selection
character(len=*),               intent (in), optional :: dataFillScheme
integer,                        intent (out), optional :: rc
```

DESCRIPTION:

Realize or remove fields inside of `state` according to `selection`. All of the fields that are realized are created internally on the same `locstream` object, allocating memory accordingly.

The type and kind of the created fields is according to argument `typekind`.

Realized fields are filled with data according to the `dataFillScheme` argument.

The arguments are:

state The `ESMF_State` object in which the fields are realized.

locstream The `ESMF_LocStream` object on which to realize the fields.

[fieldName] The name of the field in `state` to be realized, or removed, according to `selection`. If provided, and the state does not contain a field with name `fieldName`, return an error in `rc`. If not provided, realize *all* the fields contained in `state` according to `selection`.

[typekind] The typekind of the internally created field(s). The valid options are `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`, `ESMF_TYPEKIND_R4`, and `ESMF_TYPEKIND_R8`. By default use the typekind of the partially created field used during advertise, or `ESMF_TYPEKIND_R8`, if the advertised field did not have a typekind defined.

[selection] Selection of mode of operation:

- "realize_all" (default)
- "realize_connected_remove_others"

[dataFillScheme] Realized fields will be filled according to the selected fill scheme. See ?? for fill schemes. Default is to leave the data in realized fields uninitialized.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.21 NUOPC_Realize - Realize previously advertised Fields inside a State on a single Mesh with internal allocation

INTERFACE:

```
! Private name; call using NUOPC_Realize()
subroutine NUOPC_RealizeCompleteM(state, mesh, fieldName, typekind, &
    meshloc, selection, dataFillScheme, rc)
```

ARGUMENTS:

<code>type (ESMF_State)</code>		<code>:: state</code>
<code>type (ESMF_Mesh),</code>	<code>intent (in)</code>	<code>:: mesh</code>
<code>character (*),</code>	<code>intent (in),</code>	<code>optional :: fieldName</code>
<code>type (ESMF_TypeKind_Flag),</code>	<code>intent (in),</code>	<code>optional :: typekind</code>
<code>type (ESMF_MeshLoc),</code>	<code>intent (in),</code>	<code>optional :: meshloc</code>
<code>character (len=*),</code>	<code>intent (in),</code>	<code>optional :: selection</code>
<code>character (len=*),</code>	<code>intent (in),</code>	<code>optional :: dataFillScheme</code>
<code>integer,</code>	<code>intent (out),</code>	<code>optional :: rc</code>

DESCRIPTION:

Realize or remove fields inside of `state` according to `selection`. All of the fields that are realized are created internally on the same `mesh` object, allocating memory accordingly.

The type and kind of the created fields is according to argument `typekind`.

Realized fields are filled with data according to the `dataFillScheme` argument.

The arguments are:

state The `ESMF_State` object in which the fields are realized.

mesh The `ESMF_Mesh` object on which to realize the fields.

[fieldName] The name of the field in `state` to be realized, or removed, according to `selection`. If provided, and the state does not contain a field with name `fieldName`, return an error in `rc`. If not provided, realize *all* the fields contained in `state` according to `selection`.

[typekind] The typekind of the internally created field(s). The valid options are `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`, `ESMF_TYPEKIND_R4`, and `ESMF_TYPEKIND_R8`. By default use the `typekind` of the partially created field used during `advertise`, or `ESMF_TYPEKIND_R8`, if the advertised field did not have a `typekind` defined.

[meshloc] Location of data in the mesh cell. By default use the same mesh location as the advertising field, or `ESMF_STAGGERLOC_NODE` if the advertising field was created empty.

[selection] Selection of mode of operation:

- "realize_all" (default)
- "realize_connected_remove_others"

[dataFillScheme] Realized fields will be filled according to the selected fill scheme. See ?? for fill schemes. Default is to leave the data in realized fields uninitialized.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.22 NUOPC_Realize - Realize a previously advertised Field in a State

INTERFACE:

```
! Private name; call using NUOPC_Realize()
subroutine NUOPC_RealizeField(state, field, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)      :: state
type(ESMF_Field), intent(in)         :: field
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Realize a previously advertised field in `state` by replacing the advertised field with `field` of the same name.

The arguments are:

state The ESMF_State object in which the fields are realized.

field The new field to put in place of the previously advertised (empty) field.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.23 NUOPC_Realize - Realize a previously advertised Field in a State after Transfer of GeomObject

INTERFACE:

```
! Private name; call using NUOPC_Realize()
subroutine NUOPC_RealizeTransfer(state, fieldName, typekind, gridToFieldMap, &
    ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, &
    realizeOnlyConnected, removeNotConnected, realizeOnlyNotShared, &
    realizeOnlyNotComplete, field, rc)
```

ARGUMENTS:

type (ESMF_State)		:: state
character(*),	intent(in)	:: fieldName
type (ESMF_TypeKind_Flag),	intent(in), optional	:: typekind
integer, target,	intent(in), optional	:: gridToFieldMap(:)
integer, target,	intent(in), optional	:: ungriddedLBound(:)
integer, target,	intent(in), optional	:: ungriddedUBound(:)
integer,	intent(in), optional	:: totalLWidth(:)
integer,	intent(in), optional	:: totalUWidth(:)
logical,	intent(in), optional	:: realizeOnlyConnected
logical,	intent(in), optional	:: removeNotConnected
logical,	intent(in), optional	:: realizeOnlyNotShared
logical,	intent(in), optional	:: realizeOnlyNotComplete
type (ESMF_Field),	intent(out), optional	:: field
integer,	intent(out), optional	:: rc

DESCRIPTION:

Realize a field where GeomObject has been set by the NUOPC GeomObject transfer protocol.

The data of the realized field is left uninitialized by this method.

The arguments are:

state The ESMF_State object in which the field is realized.

fieldName The name of the field in state to be realized. If state does not contain a field with name fieldName, return an error in rc.

[typekind] The typekind of the internally created field(s). The valid options are ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8, ESMF_TYPEKIND_R4, and ESMF_TYPEKIND_R8. By default use the typekind of the connected provider field.

- [gridToFieldMap]** The mapping of grid/mesh dimensions against field dimensions. The argument is of rank 1 and with a size of dimCount. The elements correspond to the grid/mesh elements in order, and associates it with the indicated field dimension. Only entries between 1 and the field rank are allowed. There must be no duplicate entries in gridToFieldMap. By default use the gridToFieldMap of the connected provider field.
- [ungriddedLBound]** Lower bounds of the ungridded dimensions of the field. The number of elements defines the number of ungridded dimensions of the field and must be consistent with ungriddedUBound. By default use the ungriddedLBound of the connected provider field.
- [ungriddedUBound]** Upper bounds of the ungridded dimensions of the field. The number of elements defines the number of ungridded dimensions of the field and must be consistent with ungriddedLBound. By default use the ungriddedLBound of the connected provider field.
- [totalLWidth]** *This argument is only supported for fields defined on ESMF_Grid.* The number elements outside the lower bound of the exclusive region. The argument is of rank 1 and with a size of dimCount, the number of gridded dimensions of the field. The ordering of the dimensions is that of the field (considering gridToFieldMap). By default a zero vector is used, resulting in no elements outside the exclusive region.
- [totalUWidth]** *This argument is only supported for fields defined on ESMF_Grid.* The number elements outside the upper bound of the exclusive region. The argument is of rank 1 and with a size of dimCount, the number of gridded dimensions of the field. The ordering of the dimensions is that of the field (considering gridToFieldMap). By default a zero vector is used, resulting in no elements outside the exclusive region.
- [realizeOnlyConnected]** If set to `.false.`, realize the specified field irregardless of the connected status. If set to `.true.`, only a connected field will be realized. The default is `.true.`.
- [removeNotConnected]** If set to `.false.`, do not remove a field from the state due to its connected status. If set to `.true.`, remove the field if it is not connected. This requires realizeOnlyConnected to be `.true.`, and a runtime error will be returned otherwise. The default is `.true.`.
- [realizeOnlyNotShared]** If set to `.false.`, realize the specified field irregardless of its shared status. If set to `.true.`, only a field that has "ShareStatusField" set to "not shared" will be realized. The default is `.true.`.
- [realizeOnlyNotComplete]** If set to `.false.`, realize the specified field irregardless of its complete status. If set to `.true.`, only a field that has not yet been completed will be realized. The default is `.true.`.
- [field]** Returns the completed field that was realized by this method. An invalid field object will be returned if the conditions were such that the field was not realized.
- [rc]** Return code; equals ESMF_SUCCESS if there are no errors.
-

3.9.24 NUOPC_SetAttribute - Set the value of a NUOPC Field Attribute

INTERFACE:

```
! Private name; call using NUOPC_SetAttribute()
subroutine NUOPC_SetAttributeField(field, name, value, rc)
```

ARGUMENTS:

type (ESMF_Field)	:: field
character(*), intent(in)	:: name
character(*), intent(in)	:: value
integer, intent(out), optional	:: rc

DESCRIPTION:

Set the attribute `name` inside of `field` using the convention `NUOPC` and purpose `Instance`.

The arguments are:

field The `ESMF_Field` object on which to set the attribute.

name The name of the set attribute.

value The value of the set attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.25 NUOPC_SetAttribute - Set the value of a NUOPC State Attribute

INTERFACE:

```
! Private name; call using NUOPC_SetAttribute()
subroutine NUOPC_SetAttributeState(state, name, value, rc)
```

ARGUMENTS:

<code>type(ESMF_State)</code>	<code>:: state</code>
<code>character(*), intent(in)</code>	<code>:: name</code>
<code>character(*), intent(in)</code>	<code>:: value</code>
<code>integer, intent(out), optional</code>	<code>:: rc</code>

DESCRIPTION:

Set the attribute `name` inside of `state` using the convention `NUOPC` and purpose `Instance`.

The arguments are:

state The `ESMF_State` object on which to set the attribute.

name The name of the set attribute.

value The value of the set attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.26 NUOPC_SetTimestamp - Set the TimeStamp on a Field

INTERFACE:

```
! Private name; call using NUOPC_SetTimestamp()
subroutine NUOPC_SetTimestampField(field, time, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout)      :: field
type(ESMF_Time),  intent(in)         :: time
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the "TimeStamp" attribute according to `time` on `field`.

This call should rarely be needed in user written code.

The arguments are:

field The `ESMF_Field` object to be updated.

time The `ESMF_Time` object defining the TimeStamp.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.27 NUOPC_SetTimestamp - Set the TimeStamp on all the Fields in a State

INTERFACE:

```
! Private name; call using NUOPC_SetTimestamp()
subroutine NUOPC_SetTimestampState(state, time, selective, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)      :: state
type(ESMF_Time),  intent(in)         :: time
logical,          intent(in), optional :: selective
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the "TimeStamp" attribute according to `clock` on all the fields in `state`. Depending on `selective`, all or only some fields may be updated.

This call should rarely be needed in user written code. It is used by the generic Connector.

The arguments are:

state The `ESMF_State` object holding the fields.

time The `ESMF_Time` object defining the TimeStamp.

[selective] If `.true.`, then only set the "TimeStamp" attributes on those fields for which the "Updated" attribute is equal to "true". Otherwise set the "TimeStamp" attribute on all the fields. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.28 NUOPC_SetTimestamp - Set the TimeStamp on all the Fields in a State from Clock

INTERFACE:

```
! Private name; call using NUOPC_SetTimestamp()
subroutine NUOPC_SetTimestampStateClk(state, clock, selective, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)      :: state
type(ESMF_Clock), intent(in)         :: clock
logical,          intent(in), optional :: selective
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set the "TimeStamp" attribute according to `clock` on all the fields in `state`. Depending on `selective`, all or only some fields may be updated.

This call should rarely be needed in user written code. It is used by the generic Connector.

The arguments are:

state The `ESMF_State` object holding the fields.

clock The `ESMF_Clock` object defining the TimeStamp by its current time.

[selective] If `.true.`, then only set the "TimeStamp" attributes on those fields for which the "Updated" attribute is equal to "true". Otherwise set the "TimeStamp" attribute on all the fields. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.10 Auxiliary Routines

Auxiliary routines are provided with the NUOPC Layer as a convenience to the user. Typically more work is needed on these methods before considering them NUOPC core functionality.

3.10.1 NUOPC_Write - Write a distributed interpolation matrix to file in SCRIP format

INTERFACE:

```
! Private name; call using NUOPC_Write()
subroutine NUOPC_SCRIPWrite(factorList, factorIndexList, fileName, &
    relaxedflag, rc)
```

ARGUMENTS:

```
real (ESMF_KIND_R8), intent(in), target    :: factorList(:)
integer,              intent(in), target    :: factorIndexList(:, :)
character(*),         intent(in)           :: fileName
logical,              intent(in), optional :: relaxedflag
integer,              intent(out), optional :: rc
```

DESCRIPTION:

Write the destributed interpolaton matrix provided by `factorList` and `factorIndexList` to a SCRIP formatted NetCDF file. Each PET calls with its local list of factors and indices. The call then writes the distributed factors into a single file. If the file already exists, the contents is replaced by this call.

The arguments are:

factorList The distributed factor list.

factorIndexList The distributed list of source and destination indices.

fileName The name of the file to be written to.

[relaxedflag] If `.true.`, then no error is returned even if the call cannot write the file due to library limitations. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.10.2 NUOPC_Write - Write a distributed factorList to file

INTERFACE:

```
! Private name; call using NUOPC_Write()
subroutine NUOPC_FactorsWrite(factorList, fileName, rc)
```

ARGUMENTS:

```
real (ESMF_KIND_R8), pointer                :: factorList(:)
character(*),         intent(in)            :: fileName
integer,              intent(out), optional :: rc
```

DESCRIPTION:

THIS METHOD IS DEPRECATED. Use 3.10.1 instead.

Write the destributed `factorList` to file. Each PET calls with its local list of factors. The call then writes the distributed factors into a single file. The order of the factors in the file is first by PET, and within each PET the PET-local order is preserved. Changing the number of PETs for the same regrid operation will likely change the order of factors across PETs, and therefore files written will differ.

The arguments are:

factorList The distributed factor list.

fileName The name of the file to be written to.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.10.3 NUOPC_Write - Write Field data to file

INTERFACE:

```
! Private name; call using NUOPC_Write()
subroutine NUOPC_FieldWrite(field, fileName, overwrite, status, timeslice, &
    iofmt, relaxedflag, rc)
```

ARGUMENTS:

type (ESMF_Field),	intent (in)	:: field
character (*),	intent (in)	:: fileName
logical,	intent (in), optional	:: overwrite
type (ESMF_FileStatus_Flag),	intent (in), optional	:: status
integer,	intent (in), optional	:: timeslice
type (ESMF_IOFmt_Flag),	intent (in), optional	:: iofmt
logical,	intent (in), optional	:: relaxedflag
integer,	intent (out), optional	:: rc

DESCRIPTION:

Write the data in `field` to `file` under the field's "StandardName" attribute if supported by the `iofmt`.

The arguments are:

field The ESMF_Field object whose data is to be written.

fileName The name of the file to write to.

[overwrite] A logical flag, the default is `.false.`, i.e., existing Field data may *not* be overwritten. If `.true.`, the data corresponding to each field's name will be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Valid options are ESMF_FILESTATUS_NEW, ESMF_FILESTATUS_OLD, ESMF_FILESTATUS_REPLACE, and ESMF_FILESTATUS_UNKNOWN (default).

[timeslice] Time slice counter. Must be positive. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.:` If the `timeslice` value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.:` Any positive `timeslice` value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[iofmt] The I/O format. Valid options are `ESMF_IOFMT_BIN` and `ESMF_IOFMT_NETCDF`. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

[relaxedflag] If `.true.`, then no error is returned even if the call cannot write the file due to library limitations, or because field does not contain any data. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.10.4 NUOPC_Write - Write the Fields within a State to NetCDF files

INTERFACE:

```
! Private name; call using NUOPC_Write()
subroutine NUOPC_StateWrite(state, fieldNameList, fileNamePrefix, overwrite, &
    status, timeslice, relaxedflag, rc)
```

ARGUMENTS:

<code>type (ESMF_State),</code>	<code>intent (in)</code>	<code>:: state</code>
<code>character (len=*),</code>	<code>intent (in), optional</code>	<code>:: fieldNameList (:)</code>
<code>character (len=*),</code>	<code>intent (in), optional</code>	<code>:: fileNamePrefix</code>
<code>logical,</code>	<code>intent (in), optional</code>	<code>:: overwrite</code>
<code>type (ESMF_FileStatus_Flag),</code>	<code>intent (in), optional</code>	<code>:: status</code>
<code>integer,</code>	<code>intent (in), optional</code>	<code>:: timeslice</code>
<code>logical,</code>	<code>intent (in), optional</code>	<code>:: relaxedflag</code>
<code>integer,</code>	<code>intent (out), optional</code>	<code>:: rc</code>

DESCRIPTION:

Write the data of the fields within a `state` to NetCDF files. Each field is written to an individual file using the "StandardName" attribute as NetCDF attribute.

The arguments are:

state The `ESMF_State` object containing the fields.

[fieldNameList] List of names of the fields to be written. By default write all the fields in `state`.

[fileNamePrefix] File name prefix, common to all the files written.

[overwrite] A logical flag, the default is `.false.`, i.e., existing Field data may *not* be overwritten. If `.true.`, the data corresponding to each field's name will be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Valid options are `ESMF_FILESTATUS_NEW`, `ESMF_FILESTATUS_OLD`, `ESMF_FILESTATUS_REPLACE`, and `ESMF_FILESTATUS_UNKNOWN` (default).

[timeslice] Time slice counter. Must be positive. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.:` If the `timeslice` value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.:` Any positive `timeslice` value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a `timeslice` one greater than the maximum will be written.

[relaxedflag] If `.true.`, then no error is returned even if the call cannot write the file due to library limitations. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

4 Standardized Component Dependencies

Most of the NUOPC Layer deals with specifying the interaction between ESMF components within a running ESMF application. ESMF provides several mechanisms of how an application can be made up of individual Components. This chapter deals with reigning in the many options supported by ESMF and setting up a standard way for assembling NUOPC compliant components into a working application.

ESMF supports single executable as well as some forms of multiple executable applications. Currently the NUOPC Layer only addresses the case of single executable applications. While it is generally true that executing single executable applications is easier and more widely supported than executing multiple executable applications, building a single executable from multiple components can be challenging. This is especially true when the individual components are supplied by different groups, and the assembly of the final application happens apart from the component development. The purpose of standardizing component dependencies as part of the NUOPC Layer is to provide a solution to the technical aspect of assembling applications built from NUOPC compliant components.

As with the other parts of the NUOPC Layer, the standardized component dependencies specify aspects that ESMF purposefully leaves unspecified. Having a standard way to deal with component dependencies has several advantages. It makes reading and understand NUOPC compliant applications more easily. It also provides a means to promote best practices across a wide range of application systems. Ultimately the goal of standardizing the component dependencies is to support "plug & build" between NUOPC compliant components and applications, where everything needed to use a component by a upper level software layer is supplied in a standard way, ready to be used by the software.

There is one aspect of the standardized component dependency that affects the component code itself: **The name of the public set services entry point into a NUOPC compliant component must be called "SetServices"**. The only exception to this rule are components that are written in C/C++ and made available for static linking. In this case, because of lack of namespace protection, the `SetServices` part must be followed by a component specific suffix. This will be discussed later in this chapter. For all other cases, unique namespaces exist that allow the entry point to be called `SetServices` across all components.

Having standardized the name of the single public entry point into a component solves the issue of having to communicate its name to the software layer that intends to use the component. At the same time, limiting the public entry point to a single accepted name does not remove any flexibility that is generally leveraged by ESMF applications. Within the context of the NUOPC Layer, there is great flexibility designed into the initialize steps. Removing the need to have to deal with alternative set services routines focuses and clarifies the NUOPC approach.

The remaining aspects of component dependency standardization all deal with build specific issues, i.e. how does the software layer that uses a component compile and link against the component code. For now the NUOPC Layer does not deal with the question on how the component itself is being built. Instead the focus is on the information that a component must provide about itself, and the format of this information, in order to be usable by another piece of software. This clear separation allows components to provide their own independent build system, which often is critical to ensure bit-for-bit reproducibility. At the same time it does not prevent build systems to be connected top-down if that is desirable.

Technically the problem of passing component specific build information up the build hierarchy is solved by using GNU makefile fragments that allow every component to provide information in form of variables to the upper level build system. The NUOPC Layer standardization requires that: **Every component must provide a makefile fragment that defines 6 variables:**

```
ESMF_DEP_FRONT
ESMF_DEP_INCPATH
ESMF_DEP_CMPL_OBJS
ESMF_DEP_LINK_OBJS
ESMF_DEP_SHRD_PATH
ESMF_DEP_SHRD_LIBS
```

The convention for makefile fragments is to provide them in files with a suffix of `.mk`. The NUOPC Layer currently adds no further restriction to the name of the makefile fragment file of a component. There seems little gain in

standardizing the name of the NUOPC compliant makefile fragment of a component since the location must be made available anyway, and adding the specific file name at the end of the supplied path does not appear inappropriate.

The meaning of the 6 makefile variables is defined in a manner that supports many different situations, ranging from simple statically linked components to situations where components are made available in shared objects, not loaded by the application until needed during runtime. The design idea of the NUOPC Layer component makefile fragment is to have each component provide a simple makefile fragment that is self-describing. Usage of advanced options requires a more sophisticated build system on the software layer that *uses* the component, while at the same time the same standard format is able to keep simple situations simple.

An indepth understanding of the capabilities of the NUOPC Layer build dependency standard requires looking at various common cases in detail. The remainder of this chapter is dedicated to this effort. Here a general definition of each variable is provided.

- `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 to find shared libraries during link-time (and during run-time unless overridden by `LD_LIBRARY_PATH`). Must be specified as absolute path.
- `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.

The following sections discuss how the standard makefile fragment is utilized in common use cases. It shows how the `.mk` file would need to look like in these cases. Each section further contains hints of how a compliant `.mk` file can be auto-generated by the component build system (provider side), as well as hints on how it can be used by an upper level software layer (consumer side). Makefile segments provided in these hint sections are *not* part of the NUOPC Layer component dependency standard. They are only provided here as a convenience to the user, showing best practices of how the standard `.mk` files can be used in practice. Any specific compiler and linker flags shown in the hint sections are those compliant with the GNU Compiler Collection.

The NUOPC Layer standard only covers the contents of the `.mk` file itself.

4.1 Fortran components that are statically built into the executable

Statically building a component into the executable requires that the associated files (object files, and for Fortran the associated module files) are available when the application is being built. It makes the component code part of the executable. A change in the component code requires re-compilation and re-linking of the executable.

A NUOPC compliant Fortran component that defines its public entry point in a module called "ABC", where all component code is contained in a single object file called "abc.o", makes itself available by providing the following `.mk` file:

```
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
ESMF_DEP_SHRD_PATH =
ESMF_DEP_SHRD_LIBS =

```

If, however, the component implementation is spread across several object files (e.g. abc.o and xyz.o), they must all be listed in the ESMF_DEP_LINK_OBJS variable:

```

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  =

```

In cases that require a large number of object files to be linked into the executable it is often more convenient to provide them in an archive file, e.g. "libABC.a". Archive files are also specified in ESMF_DEP_LINK_OBJS:

```

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>/libABC.a
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =

```

Hints for the provider side: A build rule for creating a compliant self-describing .mk file can be added to the component's makefile. For the case that component "ABC" is implemented in object files listed in variable "OBJS", a build rule that produces "abc.mk" could look like this:

```

.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)

```

Hints for the consumer side: The format of the NUOPC compliant .mk files allows the consumer side to collect the information provided by multiple components into one set of internal variables. Notice that in the makefile code below it is critical to use the := style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.

```

include abc.mk
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH))

```

```

DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

include xyz.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_XYZ=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

```

Besides the accumulation of information into the internal variables, there is a small amount of processing going on. The module name provided by the ESMF_DEP_FRONT variable is assigned to a pre-processor macro. The intention of this macro is to be used in a Fortran USE statement to access the Fortran module that contains the public access point of the component.

The include paths in ESMF_DEP_INCPATH are prepended with the appropriate compiler flag (here "-I"). The ESMF_DEP_SHRD_PATH and ESMF_DEP_SHRD_LIBS variables are also prepended by the respective compiler and linker flags in case a component brings in a shared library dependencies.

Once the .mk files of all component dependencies have been included and processed in this manner, the internal variables can be used in the build system of the application layer, as shown in the following example:

```

.SUFFIXES: .f90 .F90 .c .C

%.o : %.f90
    $(ESMF_F90COMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_F90COMPILEOPTS) $(ESMF_F90COMPILEPATHS) $(ESMF_F90COMPILEFRENOCPP) $<

%.o : %.F90
    $(ESMF_F90COMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_F90COMPILEOPTS) $(ESMF_F90COMPILEPATHS) $(ESMF_F90COMPILEFREECPP) \
$(ESMF_F90COMPILECPPFLAGS) $<

%.o : %.c
    $(ESMF_CXXCOMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_CXXCOMPILEOPTS) $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \
$(ESMF_CXXCOMPILECPPFLAGS) $<

%.o : %.C
    $(ESMF_CXXCOMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_CXXCOMPILEOPTS) $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \
$(ESMF_CXXCOMPILECPPFLAGS) $<

app: app.o appSub.o $(DEP_LINK_OBJS)
    $(ESMF_F90LINKER) $(ESMF_F90LINKOPTS) $(ESMF_F90LINKPATHS) \
$(ESMF_F90LINKRPATHS) -o $@ $^ $(DEP_SHRD_PATH) $(DEP_SHRD_LIBS) \
$(ESMF_F90ESMF_LINKLIBS)

app.o: appSub.o
appSub.o: $(DEP_CMPL_OBJS)

```

4.2 Fortran components that are provided as shared libraries

Providing a component in form of a shared library requires that the associated files (object files, and for Fortran the associated module files) are available when the application is being built. However, different from the statically linked case, the component code does *not* become part of the executable, instead it will be loaded separately each time the executable is loaded during start-up. This requires that the executable finds the component shared libraries, on which it depends, during start-up. A change in the component code typically does not require re-compilation and re-linking of the executable, instead a new version of the component shared library will be loaded automatically when it is available at execution start-up.

A NUOPC compliant Fortran component that defines its public entry point in a module called "ABC", where all component code is contained in a single shared library called "libABC.so", makes itself available by providing the following .mk file:

```
ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH     = <absolute path to associated ABC module file>
ESMF_DEP_CMPL_OBJS   =
ESMF_DEP_LINK_OBJS   =
ESMF_DEP_SHRD_PATH   = <absolute path to libABC.so>
ESMF_DEP_SHRD_LIBS   = libABC.so
```

Hints for the provider side: The following build rule will create a compliant self-describing .mk file ("abc.mk") for a component that is made available as a shared library. The case assumes that component "ABC" is implemented in object files listed in variable "OBJS".

```
.PRECIOUS: %.so
%.mk : %.so
    @echo "# ESMF self-describing build dependency makefile fragment" > $@
    @echo >> $@
    @echo "ESMF_DEP_FRONT      = ABC" >> $@
    @echo "ESMF_DEP_INCPATH     = `pwd`" >> $@
    @echo "ESMF_DEP_CMPL_OBJS   = " >> $@
    @echo "ESMF_DEP_LINK_OBJS   = " >> $@
    @echo "ESMF_DEP_SHRD_PATH   = `pwd`" >> $@
    @echo "ESMF_DEP_SHRD_LIBS = "$* >> $@

abc.mk:

abc.so: $(OBJS)
    $(ESMF_CXXLINKER) -shared -o $@ $<
    mv $@ lib$@
    rm -f $<
```

Hints for the consumer side: The format of the NUOPC compliant .mk files allows the consumer side to collect the information provided by multiple components into one set of internal variables. This is independent on whether some or all of the components are provided as shared libraries.

The path specified in ESMF_DEP_SHRD_PATH is required when building the executable in order for the linker to find the shared library. Depending on the situation, it may be desirable to also encode this search path into the executable through the RPATH mechanism as shown below. However, in some cases, e.g. when the actual shared library to be used during execution is *not* available from the same location as during build-time, it may not be useful to encode the RPATH. In either case, having set the LD_LIBRARY_PATH environment variable to the desired location of the shared library at run-time will ensure that the correct library file is found.

Notice that in the makefile code below it is critical to use the := style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.


```

include abc.mk
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
    $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

```

(Here COMMA is a variable that contains a single comma which would cause syntax issues if it was written into the "addprefix" command directly.)

The internal variables set by the above makefile code can then be used by exactly the same makefile rules shown for the statically linked case. In fact, component "ABC" that comes in through "abc.mk" could either be a statically linked component or a shared library component. The makefile code shown here for the consumer side handles both cases alike.

4.3 Components that are loaded during run-time as shared objects

Making components available in the form of shared objects allows the executable to be built in the complete absence of any information that depends on the component code. The only information required when building the executable is the name of the shared object file that will supply the component code during run-time. The shared object file of the component can be replaced at will, and it is not until run-time, when the executable actually tries to access the component, that the shared object must be available to be loaded.

A NUOPC compliant component where all component code, including its public access point, is contained in a single shared object called "abc.so", makes itself available by providing the following .mk file:

```

ESMF_DEP_FRONT      = abc.so
ESMF_DEP_INCPATH    =
ESMF_DEP_CMPL_OBJS  =
ESMF_DEP_LINK_OBJS  =
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =

```

The other parts of the .mk file may be utilized in special cases, but typically the shared object should be self-contained.

It is interesting to note that at this level of abstraction, there is no more difference between a component written in Fortran, and a component written in in C/C++. In both cases the public entry point available in the shared object must be SetServices as required by the NUOPC Layer component dependency standard. (NUOPC does allow for customary name mangling by the Fortran compiler.)

Hints for the provider side: The following build rule will create a compliant self-describing .mk file ("abc.mk") for a component that is made available as a shared object. The case assumes that component "ABC" is implemented in object files listed in variable "OBJS".

```

.PRECIOUS: %.so
%.mk : %.so
    @echo "# ESMF self-describing build dependency makefile fragment" > $@
    @echo >> $@
    @echo "ESMF_DEP_FRONT      = "$< >> $@
    @echo "ESMF_DEP_INCPATH    = " >> $@
    @echo "ESMF_DEP_CMPL_OBJS = " >> $@
    @echo "ESMF_DEP_LINK_OBJS = " >> $@

```

```

@echo "ESMF_DEP_SHRD_PATH = "          >> $@
@echo "ESMF_DEP_SHRD_LIBS = "          >> $@

abc.mk:

abc.so: $(OBJS)
        $(ESMF_CXXLINKER) -shared -o $@ $<
        rm -f $<

```

Hints for the consumer side: The format of the NUOPC compliant .mk files still allows the consumer side to collect the information provided by multiple components into one set of internal variables. This still holds when some or all of the components are provided as shared objects. In fact it is very simple to make all of the component sections in the consumer makefile handle both cases.

Notice that in the makefile code below it is critical to use the := style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.

```

include abc.mk
ifneq (,$(findstring .so,$(ESMF_DEP_FRONT)))
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_SO_ABC="\$(ESMF_DEP_FRONT)\"
else
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
endif
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
        $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

```

The above makefile segment supports component "ABC" that is described in "abc.mk" to be made available as a Fortran static component, a Fortran shared library, or a shared object. The conditional around assigning variable DEP_FRONTS either leads to having set the macro FRONT_ABC as before, or setting a different macro FRONT_SO_ABC. The former indicates that a Fortran module is available for the component and requires a USE statement in the code. The latter macro indicates that the component is made available through a shared object, and the macro can be used to specify the name of the shared object in the associated call.

Again the internal variables set by the above makefile code can be used by the same makefile rules shown for the statically linked case.

4.4 Components that depend on components

The NUOPC Layer supports component hierarchies where a component can be a child of another component. This hierarchy of components translates into component build dependencies that must be dealt with in the NUOPC Layer standardization of component dependencies.

A component that sits in an intermediate level of the component hierarchy depends on the components "below" while at the same time it introduces a dependency by itself for the parent further "up" in the hierarchy. Within the NUOPC Layer component dependency standard this means that the intermediate component functions as a consumer of its child components' .mk files, and as a provider of its own .mk file that is then consumed by its parent. In practice this double role translates into passing link dependencies and shared library dependencies through to the parent, while the front and compile dependency is simply defined by the intermediate component itself.

Consider a NUOPC compliant component that defines its public entry point in a module called "ABC", and where all component code is contained in a single object file called "abc.o". Further assume that component "ABC" depends on two components "XXX" and "YYY", where "XXX" provides the .mk file:

```
ESMF_DEP_FRONT      = XXX
ESMF_DEP_INCPATH    = <absolute path to the associated XXX module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/xxx.o
ESMF_DEP_LINK_OBJS  = <absolute path>/xxx.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

and "YYY" provides the following:

```
ESMF_DEP_FRONT      = YYY
ESMF_DEP_INCPATH    = <absolute path to the associated XXX module file>
ESMF_DEP_CMPL_OBJS  =
ESMF_DEP_LINK_OBJS  =
ESMF_DEP_SHRD_PATH  = <absolute path to libYYY.so>
ESMF_DEP_SHRD_LIBS  = libYYY.so
```

Then the .mk file provided by "ABC" needs to contain the following information:

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

Hints for an intermediate component that is consumer and provider: For the consumer side it is convenient to collect the information provided by multiple component dependencies into one set of internal variables. However, the details on how some of the imported information is processed into the internal variables depends on whether the intermediate component is going to make itself available for static or dynamic access.

In the static case all link and shared library dependencies must be passed to the next higher level, and these dependencies should simply be collected and passed on to the next level:

```
include xxx.mk
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_XXX=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(ESMF_DEP_SHRD_PATH)
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(ESMF_DEP_SHRD_LIBS)

include yyy.mk
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_YYY=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(ESMF_DEP_SHRD_PATH)
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(ESMF_DEP_SHRD_LIBS)

.PRECIOUS: %.o
```

```

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

```

In the case where the intermediate component is linked into a dynamic library, or a dynamic object, all of its object and shared library dependencies can be linked in. In this case it is more useful to do some processing on the shared library dependencies, and not to include them in the produced .mk file.

```

include xxx.mk
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_XXX=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
    $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

include yyy.mk
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_YYY=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
    $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

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

```

4.5 Components written in C/C++

ESMF provides a basic C API that supports writing components in C or C++. There is currently no C version of the NUOPC Layer API available, making it harder, but not impossible to write NUOPC Layer compliant ESMF components in C/C++. For the sake of completeness, the NUOPC component dependency standardization does cover the case of components being written in C/C++.

The issue of whether a component is written in Fortran or C/C++ only matters when the dependent software layer has a compile dependency on the component. In other words, components that are accessed through a shared object have no compile dependency, and the language is of no effect (see 4.3). However, components that are statically linked or made available through shared libraries do introduce compile dependencies. These compile dependencies become language

dependent: a Fortran component must be accessed via the `USE` statement, while a component with a C interface must be accessed via `#include`.

The decision between the three cases: compile dependency on a Fortran component, compile dependency on a C/C++ component, or no compile dependency can be made on the `ESMF_DEP_FRONT` variable. By default it is assumed to contain the name of the Fortran module that provides the public entry point into a component written in Fortran. However, if the contents of the `ESMF_DEP_FRONT` variable ends in `.h`, it is interpreted as the header file of a component with a C interface. Finally, if it ends in `.so`, there is no compile dependency, and the component is accessible through a shared object.

A NUOPC compliant component written in C/C++ that defines its public access point in "abc.h", where all component code is contained in a single object file called "abc.o", makes itself available by providing the following `.mk` file:

```
ESMF_DEP_FRONT      = abc.h
ESMF_DEP_INCPATH    = <absolute path to abc.h>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/abc.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

Hints for the implementor:

There are a few subtle complications to cover for the case where a component with C interface comes in as a compile dependency. First there is Fortran name mangling of symbols which includes underscores, but also changes to lower or upper case letters. The ESMF C interface provides a macro (`FTN_X`) that deals with the underscore issue on the C component side, but it cannot address the lower/upper case issue. The ESMF convention for using C in Fortran assumes all external symbols lower case. The NUOPC Layer follows this convention in accessing components with C interface from Fortran.

Secondly, there is no namespace protection of the public entry points. For this reason, the public entry point cannot just be `setservices` for all components written in C. Instead, for components with C interface, the public entry point must be `setservices_name`, where "name" is the same as the root name of the header file specified in `ESMF_DEP_FRONT`. (The absence of namespace protection is still an issue where multiple C components with the same name are specified. This case requires that components are renamed to something more unique.)

Finally there is the issue of providing an explicit Fortran interface for the public entry point. One way of handling this is to provide the explicit Fortran interface as part of the components header file. This is essentially a few lines of Fortran code that can be used by the upper software layer to implement the explicit interface. As such it must be protected from being processed by the C/C++ compiler:

```
#if (defined __STDC__ || defined __cplusplus)

// ----- C/C++ block -----

#include "ESMC.h"
extern "C" {
    void FTN_X(setservices_abc) (ESMC_GridComp gcomp, int *rc);
}

#else

!! ----- Fortran block -----

interface
    subroutine setservices_abc(gcomp, rc)
        use ESMF
```

```

        type(ESMF_GridComp)    :: gcomp
        integer, intent(out) :: rc
    end subroutine
end interface

#endif

```

An upper level software layer that intends to use a component that comes with such a header file can then use it directly on the Fortran side to make the component available with an explicit interface. For example, assuming the macro `FRONT_H_ATMF` holds the name of the associated header file:

```

#ifdef FRONT_H_ATMF
module ABC
#include FRONT_H_ATMF
end module
#endif

```

This puts the explicit interface of the `setservices_abc` entry point into a module named "ABC". Except for this small block of code, the C/C++ component becomes indistinguishable from a component implemented in Fortran.

Hints for the provider side: Adding a build rule for creating a compliant self-describing `.mk` file into the component's makefile is straightforward. For the case that the component in "abc.h" is implemented in object files listed in variable "OBJS", a build rule that produces "abc.mk" could look like this:

```

.PRECIOUS: %.o
%.mk : %.o
@echo "# ESMF self-describing build dependency makefile fragment" > $@
@echo >> $@
@echo "ESMF_DEP_FRONT      = abc.h"          >> $@
@echo "ESMF_DEP_INCPATH    = `pwd`"          >> $@
@echo "ESMF_DEP_CMPL_OBJS  = `pwd`/"$<      >> $@
@echo "ESMF_DEP_LINK_OBJS  = `pwd`/"$<      >> $@
@echo "ESMF_DEP_SHRD_PATH  = "              >> $@
@echo "ESMF_DEP_SHRD_LIBS  = "              >> $@

abc.mk:

abc.o: abc.h

```

Hints for the consumer side: The format of the NUOPC compliant `.mk` files still allows the consumer side to collect the information provided by multiple components into one set of internal variables. This still holds even when any of the provided components could come in as a Fortran component for static linking, as a C/C++ component for static linking, or as a shared object. All of the component sections in the consumer makefile can be made capable of handling all three cases. However, if it is clear that a certain component is for sure supplied as one of these flavors, it may be clearer to hard-code support for only one mechanism for this component.

Notice that in the makefile code below it is critical to use the `:=` style assignment instead of a simple `=` in order to have the assignment be based on the *current* value of the right hand variables.

This example shows how the section for a specific component can be made compatible with all component dependency modes:

```

include abc.mk

```

```

ifneq (,$(findstring .h,$(ESMF_DEP_FRONT)))
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_H_ABC="\$(ESMF_DEP_FRONT)\ "
else ifneq (,$(findstring .so,$(ESMF_DEP_FRONT)))
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_SO_ABC="\$(ESMF_DEP_FRONT)\ "
else
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
endif
DEP_FRONTS      := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS        := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS   := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS   := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH   := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
$(addprefix -Wl,$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS   := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

```

The above makefile segment will end up setting macro `FRONT_H_ABC` to the header file name, if the component described in "abc.mk" is a C/C++ component. It will instead set macro `FRONT_SO_ABC` to the shared object if this is how the component is made available, or set macro `FRONT_ABC` to the Fortran module name if that is the mechanism for gaining access to the component code. The calling code can use these macros to activate the corresponding code, as well as has access to the required name string in each case

The internal variables set by the above makefile code can be used by the same makefile rules shown for the statically linked case. This usage implements the correct dependency rules, and passes the macros through the compiler flags.

5 NUOPC Layer Compliance

The NUOPC Layer introduces a modeling system architecture based on Models, Mediators, Connectors, and Drivers. The Layer defines the rules of engagement between these components. Many of these rules are formulated on the basis of metadata. This metadata can be expected for compliance.

One of the challenges when inspecting a component for NUOPC Layer compliance is that many of the rules of engagement are run-time rules. This means that they address the dynamical behavior of a component during run-time. For this reason, comprehensive compliance testing cannot be done statically but requires the execution of code.

Currently there are two sets of tools available to address the issue of NUOPC Layer compliance testing. The *Compliance Checker* is a runtime analysis tool that can be enabled by setting an ESMF environment variable at runtime. When active, the Compliance Checker intercepts all interactions between components that go through the ESMF component interface, and analyzes them with respect to the NUOPC Layer rules of engagement. Warnings are printed to the log files when issues or non-compliances are detected.

The *Component Explorer* is another compliance testing tool. It focuses on interacting with a single component, and analyzing it during the early initialization phases. The Component Explorer and Compliance Checker are compatible with each other and it is often useful to use them both at the same time.

5.1 The Compliance Checker

The NUOPC Compliance Checker is a run-time analysis tool that can be turned on for any ESMF application. The Compliance Checker is turned off by default, as to not negatively affect performance critical runs. The Compliance Checker is enabled by setting the following ESMF runtime environment variable:

```
ESMF_RUNTIME_COMPLIANCECHECK=ON
```

As a run-time variable, setting it does not require recompilation of the ESMF library or the user application. The same

executable and library will start to generate Compliance Checker output when the above variable is found set during execution.

The function of the Compliance Checker is to intercept all interactions between the components of an ESMF application, and to analyze them according to the NUOPC Layer rules of engagement. The following aspects are currently reported on:

- Presence of the standard ESMF Initialize, Run, and Finalize methods and the number of phases in each.
- Timekeeping and whether it conforms with the NUOPC Layer rules.
- Fields or FieldBundles (not Arrays/ArrayBundles) being passed between Components.
- Details about the Fields being passed through import and export States.
- Component and Field metadata.

Besides the above aspects, the output of the Compliance Checker also provides a means to easily get an idea of the exact dynamical control flow between the components of an application.

The Compliance Checker uses the ESMF Log facility to produce the compliance report during the execution of an ESMF application. The output is located in the default ESMF Log files. There are advantages of using the existing Log facility to generate the compliance report. First, the ESMF Log facility offers time stamping of messages, and deals with all of the file access and multi-PET issues. Second, going through the ESMF Log guarantees that all the output appears in the correct chronological order. This applies to all of the output, including entries from other ESMF system levels or from the user level.

A sample output of the Compliance Checker output in action:

```
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM:>START register compliance check.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: phase Zero for Initialize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: 5 phase(s) of Initialize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: 1 phase(s) of Run registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: 1 phase(s) of Finalize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM:>STOP register compliance check.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM2MED:>START register compliance check.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM2MED: phase Zero for Initialize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM2MED: 3 phase(s) of Initialize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM2MED: 1 phase(s) of Run registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM2MED: 1 phase(s) of Finalize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM2MED:>STOP register compliance check.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:MED2ATM:>START register compliance check.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:MED2ATM: phase Zero for Initialize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:MED2ATM: 3 phase(s) of Initialize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:MED2ATM: 1 phase(s) of Run registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:MED2ATM: 1 phase(s) of Finalize registered.
20131108 172844.458 INFO PET0 COMPLIANCECHECKER:|->|->|>:MED2ATM:>STOP register compliance check.
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: >START InitializePrologue for phase= 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: importState name: modelComp 1 Import State
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: importState stateintent: ESMF_STATEINTENT_IMPORT
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: importState itemCount: 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: exportState name: modelComp 1 Export State
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: exportState stateintent: ESMF_STATEINTENT_EXPORT
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: exportState itemCount: 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM:ESMF Stats: the virtual memory used by this PET (in KB): 974868
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM:ESMF Stats: the physical memory used by this PET (in KB): 49440
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM:ESMF Stats: ESMF Fortran objects referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM:ESMF Stats: ESMF objects (F & C++) referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|->|->|>:ATM: >STOP InitializePrologue for phase= 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: >START InitializeEpilogue for phase= 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: the virtual memory used by this PET (in KB): 974868
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: the physical memory used by this PET (in KB): 49448
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: ESMF Fortran objects referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: ESMF objects (F & C++) referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: GridComp level attribute check: convention: 'NUOPC', purpose: 'General'.
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ShortName> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <LongName> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <Description> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ModelType> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ReleaseDate> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <PreviousVersion> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ResponsiblePartyRole> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <Name> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <EmailAddress> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <PhysicalAddress> present but NOT set!
20131108 172844.459 WARNING PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <URL> present but NOT set!
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <Verbosity> present and set: high
20131108 172844.459 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[1] present and set: IPDv02p1=1
20131108 172844.460 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[2] present and set: IPDv02p3=2
20131108 172844.460 INFO PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[3] present and set: IPDv02p4=3
```



```

20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: Component level attribute: <InitializePhaseMap>[4] present and set: IPDv02p5=5
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: Component level attribute: <NestingGeneration> present and set: 0
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: Component level attribute: <Nestling> present and set: 0
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: importState name: modelComp 1 Import State
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: importState stateintent: ESMF_STATEINTENT_IMPORT
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: importState itemCount: 0
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: exportState name: modelComp 1 Export State
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: exportState stateintent: ESMF_STATEINTENT_EXPORT
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: exportState itemCount: 0
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: The incoming Clock was not modified.
20131108 172844.460 WARNING    PETO COMPLIANCECHECKER:|<|<|<:ATM: ==> The internal Clock is not present!
20131108 172844.460 INFO      PETO COMPLIANCECHECKER:|<|<|<:ATM: >STOP InitializeEpilogue for phase= 0

```

All of the output generated by the Compliance Checker contains the string `COMPLIANCECHECK`, which can be used to `grep` on. The checker currently generates two types of messages, `INFO` for general analysis output, and `WARNING` for when issues with respect to the NUOPC Layer rules are detected.

In practice, when dealing with applications that have been componentized down to a very low level of the model, the output generated by the Compliance Checker can become overwhelming. For this reason a `depth` parameter is available that can be specified for the Compliance Checker environment variable:

```
ESMF_RUNTIME_COMPLIANCECHECK=ON:depth=4
```

This will limit the number of component levels that the Compliance Checker parses (here 4 levels), starting from the top level application.

5.2 The Component Explorer

The NUOPC Component Explorer is a run-time tool that can be used to gain insight into a NUOPC Layer compliant component, or to test a component's compliance. The Component Explorer is currently available as a separate download from the prototype repository:

```
https://sourceforge.net/p/esmfcontrib/svn/HEAD/tree/NUOPC/trunk/ComponentExplorer/
```

There are two parts to the Component Explorer. First the script `nuopcExplorerScript` is used to compile and link the explorer application specifically against a specified component. This part of the explorer leverages and tests the standardized component dependencies discussed in section 4. This step is initiated by calling the explorer script with the component's `mk-file` as an argument:

```
./nuopcExplorerScript <component-mk-file>
```

Any issues found during this step are reported. The successful completion of this step will produce an executable called `nuopcExplorerApp`. Success is indicated by

```
SUCCESS: nuopcExplorerApp successfully built
...exiting nuopcExplorerScript.
```

and failure by

```
FAILURE: nuopcExplorerApp failed to build
...exiting nuopcExplorerScript.
```

The second part of the Component Explorer is the explorer application itself. It can either be built using the explorer script as outlined above (recommended when a makefile fragment for the component is available) or by using the makefile directly:

```
make nuopcExplorerApp
```

In the second case the resulting `nuopcExplorerApp` is not tied to a specific component, instead the executable expects a component in form of a shared object to be specified as a command line argument when executing `nuopcExplorerApp`. In either case the explorer application needs to be started according to the execution requirements of the component it attempts to explore. This may mean that input files must be present, and that the executable be launched on a sufficient number of processes. In terms of the common `mpirun` tool, launching of `nuopcExplorerApp` may look like this

```
mpirun -np X ./nuopcExplorerApp
```

for an executable that was built against a specific component. Or like this

```
mpirun -np X ./nuopcExplorerApp <component-shared-object-file>
```

for an executable that expects a the component in form of a shared object.

The `nuopcExplorerApp` expects to find a configuration file by the name of `explorer.config` in the run directory. The configuration file contains several basic model parameter used to explore the component. An example configuration file is shown here:

```
### NUOPC Component Explorer configuration file ###

start_year:          2009
start_month:         12
start_day:           01
start_hour:          00
start_minute:        0
start_second:        0

stop_year:           2009
stop_month:          12
stop_day:            03
stop_hour:           00
stop_minute:         0
stop_second:         0

step_seconds:        21600

filter_initialize_phases: no

enable_run:          yes
enable_finalize:     yes
```

The `nuopcExplorerApp` starts to interact with the specified component, using the information read in from the configuration file. During the interaction the findings are reported to stdout, with output that will look similar to this:

```
NUOPC Component Explorer App
-----
Exploring a component with a Fortran module front...
Model component # 1 InitializePhaseMap:
  IPDv00p1=1
  IPDv00p2=2
```

```

IPDv00p3=3
IPDv00p4=4
Model component # 1 // name = ocnA
ocnA: <LongName>      : Attribute is present but NOT set!
ocnA: <ShortName>     : Attribute is present but NOT set!
ocnA: <Description>   : Attribute is present but NOT set!
-----
ocnA: importState // itemCount = 2
ocnA: importState // item # 001 // [FIELD] name = pmsl
      <StandardName> = air_pressure_at_sea_level
      <Units> = Pa
      <LongName> = Air Pressure at Sea Level
      <ShortName> = pmsl
ocnA: importState // item # 002 // [FIELD] name = rsns
      <StandardName> = surface_net_downward_shortwave_flux
      <Units> = W m-2
      <LongName> = Surface Net Downward Shortwave Flux
      <ShortName> = rsns
-----
ocnA: exportState // itemCount = 1
ocnA: exportState // item # 001 // [FIELD] name = sst
      <StandardName> = sea_surface_temperature
      <Units> = K
      <LongName> = Sea Surface Temperature
      <ShortName> = sst

```

Turning on the Compliance Checker (see section 5.1) will result in additional information in the log files.

6 Appendix A: Run Sequence

The NUOPC Driver utilizes an internal class to parametrize the run sequence. The `NUOPC_RunSequence` provides a unified data structure that allows simple as well as complex time loops to be encoded and executed. There are entry points that allow different run phases to be mapped against distinctly different time loops. Figure 2 depicts the data structures surrounding the `NUOPC_RunSequence`, starting with the `InternalState` of the `NUOPC_Driver` generic component.



Figure 2: `NUOPC_RunSequence` class as it relates to the surrounding data structures.