# WRF Tiger Team Documentation: The Registry

John Michalakes, NCAR
Daniel Schaffer, NOAA/FSL

Revision history:

- WRF Software Design and Implementation Document, Section 5. August, 2001. Michalakes
- Updated to WRF 2.0. June, 2004. Schaffer; Broken out as separate document. June 2004. Michalakes

CONTENTS

# 1. Introduction

The WRF software infrastructure provides a high level of flexibility with respect to computer architectures, dynamical cores, applications, and external libraries. The Registry is a computer aided software engineering (CASE) mechanism built into the WRF software framework to help manage this complexity as the model continues to develop. In addition, the Registry is serving as a first prototype for the eventual development of a sophisticated Application-Specific Interactive Develop Environment (ASIDE), the next step beyond computational frameworks.

The Registry as currently implemented in the WRF framework consists of a database of information about a source code as well as a program for manipulating that information to help manage the code. The registry provides a great deal of high-level single-point-of-control over the fundamental structure of the model data, and thus provides considerable flexibility for supporting multiple dynamical cores, application-specific data structures (e.g. arbitrary number of tracer arrays), and transparent and package independent management of parallel communication. The registry automatically generates sections of code that would be the most error-prone and effort intensive to manage by hand: generating code to declare, allocate, and initialize state data; generating dummy argument lists and their declarations, as well as actual argument lists used when passing state data between subroutines at the interface between layers in the WRF software architecture (driver, mediation, and model layers); generating calls to routines for initial (first guess), restart, history and boundary I/O for selected state data fields; generating halo-exchange, periodic boundary updates, transpose, and nesting communications for selected state data fields in selected patterns; generating code that defines, sets defaults for, inputs, broadcasts among processors, and makes available to the code the variable=value (namelist) information used to configure the running application; etc. Adding or modifying a state variable to a model involves modifying a single line of a single file. The registry provides a single point of control -- the registry data base -- for making changes that affect many different aspects of the code.

A number of functions of the registry have or will be mentioned elsewhere:

- Define state fields in the domain DDT (module_domain.F). Page 9.
- Generate ALLOCATE statements for state arrays in domain DDT (module_domain.F). Page 9.
- Generate model configuration inquiry routines (module_configure.F). Page 9.
- Generate assignments of namelist configuration data to fields in the domain DDT. Page 27.
- Generate actual arguments to the solve routine, dereferencing the domain DDT. Page 30.

There are other functions as well:

- Generate dummy arguments and their definitions in the solve routine.
- Generate code to read and write fields in I/O modules (e.g. module_io_wrf.F).
- Generate communication package specific code (e.g. inc/rsl_rk_data_calls.inc).

The Registry also provides a means of defining packages and associating fields and symbolic names for indices into the 4D scalar arrays with these packages. It defines the configuration variables that appear in the WRF namelist file and their meanings. The Registry is currently implemented as a flat ASCII text file, Registry/Registry. The program that interprets the registry and generates files in the inc directory is tools/registry. Source code (C language) for the program and a Makefile are also included in that directory. The registry program is designed to be extensible; for example, to add interfaces to new communications, I/O, or other external packages to WRF.

Logically, the registry data base is a collection of tables that describe the dimensions, derived types, state fields, i1 fields, packages, and communication operations (for halo updates, periodic boundary communications, transposes, and nest interpolations). Registry/Registry contains all the entries of all the tables as white-space delimited tuples. The table membership of each tuple is determined by the first element; thus, entries from different tables may be listed in any order and intermixed, though for readability one generally organizes the entries from each table in a block. The caveat for this is that the registry mechanism is single-pass; thus, if an entry of the registry depends on definitions of other entries, these must have been defined already (above) the definition that uses them. Comments in the registry begin with a # character and extend to the right to the end of the line. Certain elements are allowed to contain spaces; these must be enclosed within double quotes ("). Elements may not contain quote characters.

# 2. Registry Contents

The Registry file contains entries for a number of tables:

- Dimspec -- Describes dimensions that are used to define arrays in the model
- State – Describes state variables and arrays in the domain DDT
- I1 – Describes local variables and arrays in solve
- Typedef -- Describes derived types that are subtypes of the domain DDT[1]
- Rconfig – Describes a configuration (e.g. namelist) variable or array
- Package – Describes attributes of a package (e.g. physics)
- Halo -- Describes halo update interprocessor communications
- Period -- Describes communications for periodic boundary updates
- Xpose -- Describes communication for transposition of a variable between decompositions
- Initialization -- Describes communications and data for nest initialization from the coarse domain
- Force -- Describes communications and data for forcing of nest boundary arrays
- Feedback -- Describes communications and data for nest feedback onto the coarse domain

## 2.1. DIMSPEC ENTRIES

Dimspec Table entries provide a way to define dimensions that can then be used in the definitions of arrays in the State, I1, and Typedef Tables of the Registry. As with any Registry-defined entity, a dimension must be defined before it can be used in the Registry file. The fields of a Dimspec entry are:

- *Entry*: The keyword "dimspec"
- *DimName*: The name of the dimension (single character)
- *Order*: The order of the dimension in the WRF framework (integer: 1, 2, 3, or '-')
- *HowDefined*: specification of how the range of the dimension is defined
- *CoordAxis*: which axis the dimension corresponds to, if any (X, Y, Z, or C)
- *DatName*: metadata name of dimension

*DimName* is a single character name that will identify the dimension in specification strings used in State, I1, and TypeDef entries.

*Order* specifies with which of the internal WRF framework sets of dimension variables the dimension being defined here is associated. In other words, is it associated with the sd31:ed31,[...], sd32:ed32,[...], or sd33:ed33,[...] set of dimension variables (these are defined in frame/module_domain.F). Note that this does not determine the storage order (the order of indices) of the individual state arrays; individual storage order is specified in the State, I1, or TypeDef table entry for each array.

The registry infers the mapping of WRF framework internal dimensions to the coordinate axes of the domain according to the combination of the order specification, here, and the coordinate axis (below) specified in this table, and will set the WRF global variable MODEL_ORDER accordingly in the registry-generated file model_data_order.inc, included by the WRF source file frame/module_driver_constants.F. Note that it is all right to have more than one dimension name for, say, the x dimension. However, the Order and Coord-axis relationship must be consistent throughout.

*HowDefined* specifies how the dimension is defined for the domain. This may be done in one of three ways:

- standard_domain
- namelist=[<start namelist var>:]<end namelist variable>
- constant=[<start constant>:]<integer constant>

A dimension may be defined as being a "standard domain" dimension, in which case the size of the dimension is specified by the associated internal set of dimension variables in the WRF framework. A dimension's range may be specified by namelist variables. Two namelist variables may be used to specify a starting and ending indices; if only one namelist variable is specified, the beginning index is assumed to be 1. The namelist variables must be defined in the registry's Rconfig table. Lastly, the dimension may be a constant start and end. If only one integer constant is provided, the starting range is assumed to be one.

*CoordAxis* specifies which coordinate axis the dimension is defined for. Allowable entries are X, Y, Z, or C. An entry of C means that the dimension is not associated with a coordinate axis of the domain.

*DatName* is a string that provides a meaningful name by which this dimension is known in the metadata contained in WRF data sets that are read/written by the program. If this is not specified, the particular implementation of the WRF I/O API may choose its own metadata name or leave it without a name.

## 2.2. STATE ENTRIES

The State Table is used to define WRF state variables that will be fields in the domain derived-data-type in module_domain.F. State variables may have simple types or may themselves be derived data types in the registry; these types, in turn may contain derived data types. Although the WRF model itself does not use derived subtypes in the domain DDT, other applications may require this. The fields of a state entry are:

- *Entry*: The keyword "state"
- *Type*: The type of the state variable or array (real, double, integer, logical, character, or *derived*)
- *Sym*: The symbolic name of the variable or array
- *Dims*: A string denoting the dimensionality of the array or a hyphen (-)
- *Use*: A string denoting association with a solver or 4D species array, or a hyphen
- *NumTLev*: An integer indicating the number of time levels (for arrays) or hypen (for scalars)
- *Stagger*: String indicating staggered dimensions of variable (X, Y, Z, or hyphen for no staggering)
- *IO*: String indicating whether and how the variable is subject to I/O
- *DName*: Metadata name for the variable
- *Descrip*: Metadata description of the variable
- *Units*: Metadata units of the variable

*Type* may be simple or derived. A derived data type must have been previously defined in the registry in the Typedef table (see below). Note, derived data types are discouraged in the WRF model registry but may be used in other applications implemented under the WRF framework.

*Sym* is the base name of the variable as it will be known in the model except when the variable is a member of a four dimensional scalar array. If the variable has only one time level ('1' or '-' in the NumTLev column, #6) then the name in the registry is the name in the code; if there is more than one time-level then there are separate names in the code for each time-level, named <name>_1, <name>_2, and so on.

If the variable is a member of a 4D species array, the modifier 'f' appears in the Dims column (#4) and the name of the 4D species array appears in the Use column (#5). In all cases, F90 naming rules apply for this entry.

Lastly, if a variable is associated with a particular dynamical core (dyn_<corename> appears in the Use column (#5)), the variable is known as <corename>_name within the driver layer; that is, this is the name of the field in the TYPE(domain) in module_domain.F. Below the point in the mediation layer where fields are dereferenced from TYPE(domain) the variable is known without the <corename>_ prefix.

*Dims* specifies the dimensionality of the state field. The entry is a string of single-character dimension names followed by single character modifiers, if any. The dimension names are those that have been previously defined in the Dimspec table (see above). The number of dimensions is inferred from the number of dimension names in the string. The index order is denoted by the order that the names appear, where the fastest varying stride-one variable is leftmost. Zero-dimensional state variables may be indicated with an hyphen (-) in this column.

Modifiers appear in the string after (to the right) of the last dimension name. Modifiers are:

f The variable is a member of 4D array of fields. The name of the 4D array appears in the Use column (#5). The variables must be three dimensional, and the index order must match the index order of the other members of the array.[2]

| | |
|---|---|
| t | This applies only to 4D arrays (so this must be specified only with the f modifier). It causes the registry to also generate a 4D tendency array in the I1 data called <arrayname>_tend. |
| x | The 2D decomposition of the 3D array is such that all elements in the X dimension are on-processor (the default is for all elements in the Z dimension to be on processor.) Applies only to 3D arrays. |
| y | The 2D decomposition of the 3D array is such that all elements in the Y dimension are on-processor. Applies only to 3D arrays. |
| b | The 2D or 3D array is a lateral boundary array. The registry is allowed to define and allocate this array in a way that excludes the interior.[3] |

The following examples assume that i, j, and k have been defined (using DimSpec entries) to refer to the X, Y, and Z coordinate axes of the model domain:

| **Dims** | **Description** |
|---|---|
| ikj | 3D array whose dimensions are ordered XZY and whose Z dimension is non-decomposed |
| ikjx | 3D array whose dimensions are ordered XZY and whose X dimension is non-decomposed |
| ikjy | 3D array whose dimensions are ordered XZY and whose Y dimension is non-decomposed |
| ij | 2D array whose dimensions are ordered XY |
| k | 1D array whose dimension is Z |
| ikjf | A 3D array that is a member of the 4D array whose name is specified in the <use> field (see below) |
| ikjft | A 3D array that is a member of the 4D array whose name is specified in the <use> field (see below) and that has an associated tendency in the 4D tendency array whose name is <use>_tend. |
| ikjb | Defines a 4D array with dimensions: (max(x,y),spec_bdy_width,z,4) |
| ijb | Defines a 4D array with dimensions: (max(x,y),spec_bdy_width,1,4) |
| - | A scalar |

*Use* is a string that gives additional usage information about the variable. It can be used to specify whether a variable is associated with a particular dynamical core, in which case the first four characters of the use string must be "dyn_". In the case the field is a members of a 4D species array (dimstring has 'f' modifier), the use field tells the name of the 4D array (e.g. moist, or chem). Otherwise the use field is strictly informational and can have any string, or just '-'.

In the case of 4D species arrays, the use field provides the name of the array as it is stored in the domain derived data type and passed in through the argument list to the mediation/model layer routines. If the number of time levels (*NumTLev*) is greater than one, there is a 4D state array for each time level with _1, _2, and so on, appended to the array name.

In the case where the state variable being defined is associated with a particular dynamic core (e.g. dyn_eh, for Eulerian/Height-based vertical coordinate), the variable will be defined as a field in the driver layer derived data type but will only be allocated if that dynamics option is selected in the namelist.

The registry keeps track of the set of dynamic cores that are specified in the use entries of all the registry state arrays and then generates a set of include files in the inc directory for each dynamical core:

| | |
|---|---|
| <core>_allocs.inc | allocate statements for <core>'s state arrays |
| <core>_actual_args.inc | actual argument list for <core> |
| <core>_dummy_args.inc | dummy argument list for <core> |
| <core>_dummy_decl.inc | declarations of dummy arguments for <core> |

The set of arrays in each of these files will be the dyncore specific fields plus the non-dyncore-associated fields.

Four-dimensional arrays of fields ('f' modifier in dimstring) may not be core associated.

*NumTLev* is an integer that specifies the number of time-levels for a variable: may be 1, 2, 3, or '-' (which implies 1). If a variable has only one time level it is know by its registry name in the code (with a core prefix at the driver level if the variable is core-associated). If the variable is two time level or greater, multiple instances are created and the name of each instance has an underscore and a time level appended as discussed previously.

In the case of fields that are members of 4D arrays, the name of the 4D array (not the field name) is appended with underscore and time level.

Boundary arrays (*Dims* has 'b' modifier) may not have multiple time levels.

*Stagger* is a string that specifies the staggering, if any, for a variable. The string may consist of 'X' (denoting an extra element in the X dimension), 'Y' (extra element in Y) or 'Z' (extra element in Z) in any order or combination. An entry of '-' denotes no staggering.

(In the current implementation of WRF, all state arrays are over-dimensioned to include the extra element in each dimension whether or not the field is staggered. The specification of staggering in the registry is important for I/O, which does not include the extra element when the field is written to or read from a dataset.)

*IO* is a string that specifies if the variable is to be subject to initial, restart, history, boundary I/O or nesting interpolation. The string may consist of 'h' (subject to history I/O), 'i' (initial dataset), 'r' (restart dataset), or 'b' (lateral boundary dataset). The nesting section below lists other possible letters. The 'h', 'r', and 'i' letters may appear in any order or combination. Any dimension variable except one whose *Dims* element contains the 'b' modifier may be included in history, initial, or restart I/O. In addition, the ‘h’ and ‘i’ letters may be followed by an optional integer string consisting of ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, and/or ‘5’. Zero denotes that the variable is part of the principal input or history I/O stream. The characters ‘1’ through ‘5’ denote one of five auxiliary input or history I/O streams. If an integer string appears after ‘i’ or ‘h’ then the variable is removed from the principal input or history data stream unless it is explicitly added to the stream using the character ‘0’. Examples:

`irh` -- The state variable will be included in the input, restart, and history I/O streams,

`irh13` -- The state variable has been added to the first and third auxiliary history output streams; it has been removed from the principal history output stream, because zero is not among the integers in the integer string that follows the character 'h',

`rh01` -- The state variable has been added to the first auxiliary history output stream; it is also retained in the principal history output,

`i205hr` -- Now the state variable is included in the principal input stream as well as auxiliary inputs 2 and 5. Note that the order of the integers is unimportant. The variable is also in the principal history output stream, and

`ir12h` -- No effect; there is only 1 restart data stream.

Variables with multiple time levels are handled as follows:

| | |
|---|---|
| History | Only time level two is input/output |
| Initial | Only time level two is input/output and timestep one is copied from two on input |
| Restart | Both time levels are input/output separately and *DName* is appended with underscore and time-level in metadata |

State variables that are members of 4D arrays ('f' modifier in dimstring) are input and output to history, initial, or restart dataset as individual fields. There is no mechanism for performing I/O on an entire 4D array.

If 'b' is specified it must appear by itself and the dimstring must also contain the 'b' modifier. Only boundary arrays are read/ written to a boundary I/O dataset.

A '-' indicates that no I/O is performed on the variable.

Only state variables may be subject to I/O in WRF.

NESTING: If a variable is subject to nest interpolation or feedback, operators are also specified using the *IO* entry of the state entry. There are three streams that a variable may take between a coarse domain and a nested domain: *down,* indicated by a ‘d’ character in the *IO* string; *up*, indicated by a ‘u’; and *force*, a special form of *down*, indicated with an ‘f’. If the stream identifier is specified by itself, a default interpolation subroutine is used. Down uses interp_fcn(), defined in share/interp_fcn.F, which is the semi-Lagrangian interpolator, SINT, from MM5 nesting. Up uses copy_fcn() by default, also defined in that source file. There is no default for force; however, there is a function bdy_interp() (which also uses SINT) provided in share/interp_fcn.F. When these are specified, the state variable is passed as an argument to the interpolation routine on both the coarse domain and the nest. If the state

variable has multiple time levels, the highest numbered time level is passed.

Different functions can be specified, and additional fields can be passed into those functions, using the following syntax:

```
f=(my_bdy_fcn:dt,u_b,u_bt)
```

This will cause a different subroutine, named my_bdy_fcn, to be called instead of the default and the additional state variables dt, u_b, and u_bt (boundary and boundary tendency arrays, respectively) will be passed for both the coarse and nested domains. The interface to the subroutine should be as follows. Note the extra arguments defined for dt, u_b, and u_bt on coarse and nested domains. Note also that the registry-generated call to this routine will also provide two logical arguments to the routine indicating whether the variable is x-staggered or y-staggered.

```
SUBROUTINE my_bdy_fcn ( cfld,                              &  ! CD field
                        cids, cide, ckds, ckde, cjds, cjde,   &  ! CD domain dims
                        cims, cime, ckms, ckme, cjms, cjme,   &  ! CD mem dims
                        cits, cite, ckts, ckte, cjts, cjte,   &  ! CD patch dims
                        nfld,                                 &  ! ND field
                        nids, nide, nkds, nkde, njds, njde,   &  ! ND domain dims
                        nims, nime, nkms, nkme, njms, njme,   &  ! ND mem dims
                        nits, nite, nkts, nkte, njts, njte,   &  ! ND patch dims
                        shw,                                  &  ! stencil half width
                        xstag, ystag,                         &  ! staggering of field
                        ipos, jpos,                           &  ! Nest lower left in CD
                        nri, nrj,                             &  ! nest ratios
                        cdt, ndt,                             &  ! extra vars on CD and ND
                        cbdy, nbdy,                           &  !  "     "    "    "
                        cbdy_t, nbdy_t                        &  !  "     "    "    "
                        )
  IMPLICIT NONE

  INTEGER, INTENT(IN) :: cids, cide, ckds, ckde, cjds, cjde,   &
                         cims, cime, ckms, ckme, cjms, cjme,   &
                         cits, cite, ckts, ckte, cjts, cjte,   &
                         nids, nide, nkds, nkde, njds, njde,   &
                         nims, nime, nkms, nkme, njms, njme,   &
                         nits, nite, nkts, nkte, njts, njte,   &
                         shw,                                  &
                         ipos, jpos,                           &
                         nri, nrj

  LOGICAL, INTENT(IN) :: xstag, ystag

  REAL, DIMENSION ( cims:cime, ckms:ckme, cjms:cjme ) :: cfld
  REAL, DIMENSION ( nims:nime, nkms:nkme, njms:njme ) :: nfld
  REAL,  DIMENSION( * ), INTENT(INOUT) :: cbdy, cbdy_t, nbdy, nbdy_t
  REAL cdt, ndt
```

The down, up, and force descriptions may be included in the same IO field for a state-entry. Here is an example:

```
i01rhu=(my_feedback)d=(my_interp:mask)f=(bdy_interp:dt,u_b,u_bt)
```

This entry would specify that the state variable is input in the main input stream as well as the auxiliary-1 stream, it is part of restart and history data, it is downward forced using the user-supplied routine my_interp() which also takes the state variable mask as an argument; it is upward forced using the my_feedback() routine; and it is forced using the bdy_interp() routine, which takes as extra arguments the dt, u_b, and u_bt state variables. The mask might be a land/sea mask. It must have been previously declared as a state variable.

-

*DName* is the data name of the variable; the one by which it is known externally in the WRF metadata. If omitted or specified with '-' then the Sym name (above) is used.

*Descrip* is a short description that may accompany the variable in the dataset metadata. It may include spaces as long as the entire string is quoted using double quotes.

*Units* is a short unit description string that may accompany the variable in the dataset metadata. It may include spaces as long as

the entire string is quoted using double quotes.

## 2.3. I1 ENTRIES

Intermediate Level 1 (I1) data are similar to state data except that they do not persist from time step to time step and are defined local to the solver (stack allocated). I1 entries are distinguished from state entries in the registry by the keyword 'i1' as the first entry and by the lack of IO, DName, Descrip, and Units fields (since no I/O is allowed on I1 data).

An I1 variable may be multi-time level, and if so, the name has an underscore and time-level appended as is done with state data.

The 'b', 'f', 't', 'x', and 'y' modifiers are not valid for the *Dims* of an I1 variable. Four-dimensional species arrays may not be I1; however, as noted above, if a member of a 4D species array has 't' in its DimString, a 4D tendency array is autmoatically created as an I1 variable. This does not need to be specified in the I1 table of the registry.

## 2.4. TYPEDEF ENTRIES

Entries of the registry State table are, in effect, field definitions in the WRF Framework derived data type (DDT), of TYPE (domain). In addition to simple types, however, the fields in the WRF domain DDT may also be derived data types. The Typedef Specification Table in the registry provides a means for defining DDTs that can then be used as types in State table entries or other Typedef entries. The entries of the Typedef Specification Table are similar to State Table entries except the first entry is typedef instead of state and there is an additional entry in the second position which provides the name of the type being defined. The entries are:

- *Entry*: The keyword “typedef”
- *TypeSym*: The name of the derived type being defined.
- *Type*: The type of the state variable or array (real, double, integer, logical, character, or *derived*)
- *Sym*: The symbolic name of the variable or array
- *Dims*: A string denoting the dimensionality of the array or a hyphen (-)
- *Use*: A string denoting association with a solver or 4D scalar array, or a hyphen
- *NumTLev*: An integer indicating the number of time levels (for arrays) or hypen (for variables)
- *Stagger*: String indicating staggered dimensions of variable (X, Y, Z, or hyphen for no staggering)
- *IO*: String indicating whether and how the variable is subject to I/O
- *DName*: Metadata name for the variable
- *Descrip*: Metadata description of the variable
- *Units*: Metadata units of the variable

Fields are added to a type by listing additional entries. Types may be nested; that is, the field type element of a Typedef entry may be the name of a derived type defined by previous typedef entries. Like all other entries in the Registry, TypeDef entries for a particular type do not have to appear consecutively; however, since a type definition must be completely defined before it can be used in a State entry or another Typedef entry, the type definition must be completely defined before it is used. In other words, no further typedef entries for derived type may appear after the derived type is used in a State or other Typedef entry.

A crucial difference between a Typedef Table entry and a State Table entry in the registry is that Typedef entries do not cause any data to be allocated or defined. Only State entries and, to a more limited extent, I1 entries do that; thus, the name of the derived type or top-level type in a nested set of derived types must ultimately appear in a State Entry to be used. Registry defined derived types may also be used in I1 entries, in which case the data object is defined as I1 data in the program: local to the mediation layer (stack allocated) and existing for the duration of one call to the solve routine.

Although it is legal to list a core-association with a field in a Typedef Entry, it doesn't make sense to do so. If a derived data type is meant to be used only with a particular dynamical core, the core association should be listed with the State Table entry in which the type is used. Since the name of a field in a Typedef table entry will be defined as a field in a derived data type in the program, the name may be the same as the name of a State or I1 variable in the program. For example, a State variable U and a field U of a derived data type may exist together, because the derived data type will always be reference as a field in a derived data type (grid%typename%u versus grid%u) and no namespace collision will occur. This is not the case for metadata names in the registry,

however, and if derived data type fields are subject to I/O (i, r, or h in the IO string of the TypeDef entry) they must have unique DName strings.

State declarations that have a derived data type in the Type element may not be subject to I/O; they must have '-' in the IO element.

Here is an example:

```
# BEGIN XB_TYPE DEFINITION

typedef xb_type integer map

typedef xb_type  real grid_box_area   ij  -  1  -

typedef xb_type real dnw

# END XB_TYPE DEFINITION

state xb_type xb - -
```

In this example, *xb* is of type *xb_type*. It consists of 3 fields: *map* and *dnw* (scalars) and *grid_box_area* (a two-dimensional array).

## 2.5. RCONFIG ENTRIES

Rconfig entries specify variables and arrays that are part of the runtime configuration information that is input to the model at the beginning of the run. Rconfig entries may apply to variables or arrays. If they are variables, they apply model-wide; if they are arrays, they apply to individual domains (nest levels). In that case, the array is dimensioned from 1 to the number of domains in the run. The current (June 04) implementation of the model configuration is through a namelist and, for simplicity, the discussion in this section will assume this.

The rconfig entries have the following fields:

- *Entry:* the keyword “rconfig”
- *Type:* the type of the namelist variable (integer, real, logical – no strings yet)
- *Sym:* the name of the namelist variable or array
- *How set:* indicates how the variable is set: e.g. namelist or derived, and if namelist, in which block (i.e. time_control, domains) of the namelist it is set.
- *Nentries:* specifies the dimensionality of the namelist variable or array. Either 1 or max_domains (an integer parameter defined in module_driver_constants.F) may be specified. If 1 is specified, the variable has the same value for all nests domains. Otherwise it varies over domain.
- *Default:* the default value of the variable to be used if none is specified in the namelist; hyphen (-) for no default

## 2.6. PACKAGE ENTRIES

Package entries in the Registry are used to define a package, for example a cumulus physics package, and associate it with an rconfig variable for toggling between different packages, for example, between other cumulus schemes. The package entry also indicates which species within the 4D arrays (defined in the state entries of the registry) the package uses. Based on the runtime selection of packages, the model is able to dimension the 4D species arrays to contain the species fields that will be used on that domain. The species indices are also set accordingly. Package entries have the following fields:

- *Entry:* the keyword “package”,
- *Package name:* the name of the package: e.g. “kesslerscheme”
- *Associated rconfig choice:* the name of a rconfig variable and the value of that variable that chooses this package
- *Package state vars:* unused at present; specify hyphen (-)
- *Associated 4D scalars:* the names of 4D scalar arrays and the fields within those arrays this package uses

The following is a set of example package entries and an associated rconfig entry in the Registry for microphysics schemes:

```
package   passiveqv     mp_physics==0          -            moist:qv
package   kesslerscheme mp_physics==1          -            moist:qv,qc,qr
package   linscheme     mp_physics==2          -            moist:qv,qc,qr,qi,qs,qg
rconfig   integer       mp_physics      namelist,namelist_04     max_domains          0
```

This specifies that three microphysics options are associated with the namelist variable mp_physics. If mp_physics is set to 1 in the namelist, the Kessler microphysics option is selected. If mp_physics is 2, the Lin scheme is selected. If mp_physics is set to 0, then no microphysics is selected (passiveqv is a dummy name) but this ensures that QV will still be a field in the 4D species arrays moist_1, moist_2, and moist_tend. If a package uses fields from additional 4D species arrays, this can be specified in the *associated 4d scalars* field by separating the lists with a semicolon (but no spaces): "moist:qv,qc,qr;chem:no2,o3".

Species indices are defined in the Registry-generated source file, module_state_description.F (in the src directory). The indices are set to values at run time, based on which packages are selected in the namelist. This is done in the routine set_scalar_indices_from_config (module_configure.F) using code in the Registry-generated include file set_scalar_indices.inc.

The *package name* is also used to generate integer parameters in the Registry-generated source file module_state_description.F:

```
INTEGER, PARAMETER :: PASSIVEQV = 0
INTEGER, PARAMETER :: KESSLERSCHEME = 1
INTEGER, PARAMETER :: LINSCHEME = 2
```

These may be used in F90 SELECT statements or IF-THEN-ELSE constructs within the code by testing on the rconfig variable mp_physics (from microphysics_driver.F):

```
SELECT CASE(config_flags%mp_physics)  ! select microphysics based on namelist
      CASE (KESSLERSCHEME)
           CALL kessler( th_phy,                          &
                  moist_new(ims,kms,jms,P_QV),            &
                  moist_new(ims,kms,jms,P_QC),            &
                  moist_old(ims,kms,jms,P_QC),            &
                  moist_new(ims,kms,jms,P_QR),            &
                  moist_old(ims,kms,jms,P_QR),            &
                  rho, pi_phy, RAINNC, dt, z,             &
                  ids,ide, jds,jde, kds,kde,              &
                  ims,ime, jms,jme, kms,kme,              &
                  its,ite, jts,jte, kts,kte               )
      CASE (LINSCHEME)
           CALL lin_et_al( th_phy,                        &
             . . .
    CASE DEFAULT
       WRITE(6,*) ' no microphysics for n_moist = ',n_moist
       WRITE(6,*) ' error stop in subroutine microphyscis '
       STOP
END SELECT
```

## 2.7. HALO AND PERIOD ENTRIES

Halo and period entries in the registry define communication operations in the model. Halo entries specify halo updates around a patch; period entries define updates of periodic boundary conditions (in both cases these apply only to horizontal dimensions). The first field is the keyword "halo" or the keyword "period". The second entry is the name that will be used to refer to the communication operation being defined. The third entry is a list of information about the operation.

For halos, the third entry comprises a list of the form:

*npts:f1,f2,...[;npts:f1,f2,...]**

No spaces separate the elements of the list. *Npts* specifies the number of points of the stencil to be used in updating the state arrays named in the list following the colon. A single halo update operation may involve different stencils on different arrays and these may be listed separated by semi-colons. There may be two or more update operations that use the same variables. However if *npts* differ between the two operations then you will need separate entries in the Registry. Currently defined values for *npts* are:

- 4 point: one cell in N,S,E, and W
- 8 point: 4 point plus corners (all eight neighbors around a cell)
- 12 point: 8 point plus second cell in N,S,E, and W
- 24 point: 12 point plus all corner cells
- 48 point: 24 point plus third cell in N,S,E, and W, plus all corners

Graphical representations of values of *npts* is shown in the figure below.

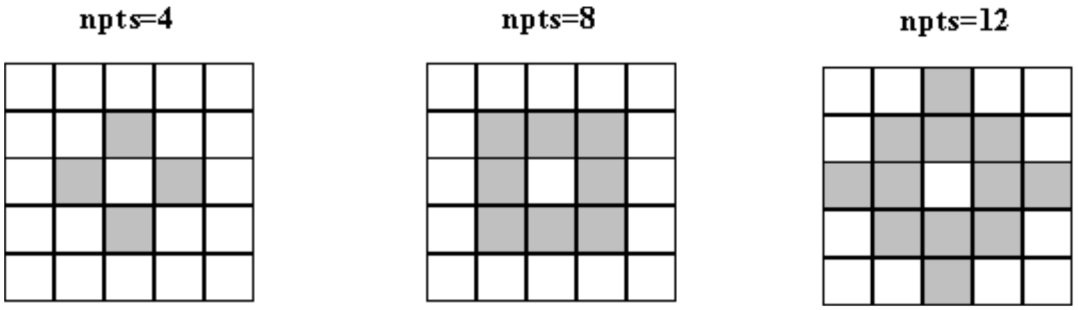


**Figure 1. Representation of the halo points updated for npts=4,8,12. Updated halo points are shown in grey.**

Period entries are similar to halo entries except the form of the third entry is:

*width:f1,f2,...[;width:f1,f2,...]**

where *width* is the number of cells to update in the periodic boundary.

## 2.8. XPOSE ENTRIES

Xpose entries are used to define transpositions of a variable between one decomposition and another. The three decompositions are Z non-decomposed, X non-decomposed, Y non-decomposed. These decompositions are shown in the figure below. Xpose entries have the following fields:

- *Entry:* the keyword "xpose",
- *XposeName:* This name will be used by the code to refer to the transpose operation being defined as shown in the example below.
- *Use:* A string that associates the transpose with a dynamical core or gives descriptive information.
- *XposeVariables:* A list of state variables that are the source or target of a transposition. Order is significant. The first variable must be non-decomposed in Z, the second in X and the third in Y.

Here is an example Registry entry:

Xpose TRANS_A dyn_em U_z, U_x, U_y

In this case, the model code would need to contain the following at the location where the transpose is required:

#include "TRANS_A_z2x.incl"

In this case, TRANS_A_z2x.incl will contain code that transposes of U_z (Z non-decomposed) to U_x (X non-decomposed). TRANS_A_x2y.incl will contain code that transposes U_x (X non-decomposed) to U_y (Y non-decomposed) and so on.

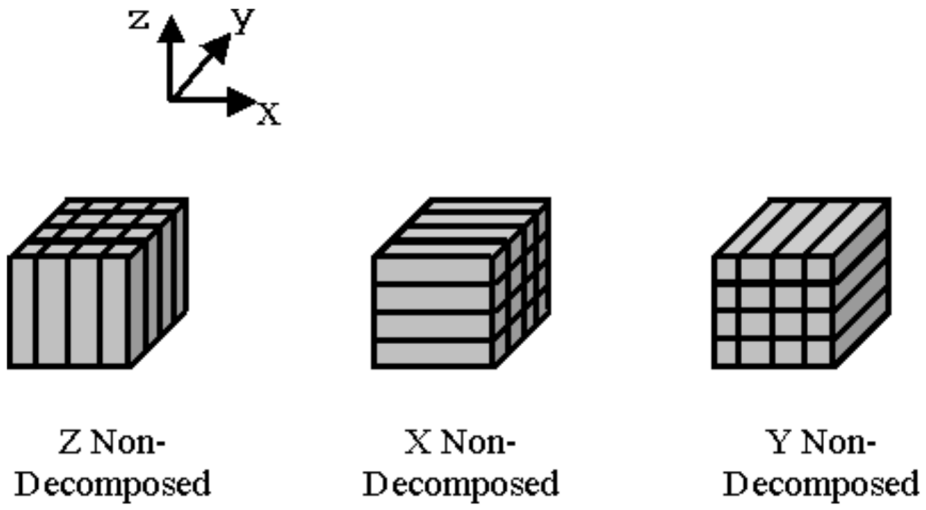


**Figure 2. Decompositions supported by the Xpose registry entry.**

# Appendix: table of registry files

| Registry include file | Description | Included by | Generated by |
|---|---|---|---|
| rk_allocs.inc | Field allocate statements for domain data structure | src/module_domain.F | args_and_i1.pm |
| rk_actual_args.inc | Actual argument list of state fields | src/module_dm.F<br>src/module_initialize.F<br>src/module_initialize_b_wave.F<br>src/module_initialize_hill2d_x.F<br>src/module_initialize_quarter_ss.F<br>src/module_initialize_real.F<br>src/module_initialize_squall2d_x.F<br>src/module_initialize_squall2d_y.F<br>src/module_start.F<br>src/solve_interface.F | args_and_i1.pm |
| rk_dummy_args.inc | Dummy argument list of state fields | src/module_dm.F<br>src/module_initialize.F<br>src/module_initialize_b_wave.F<br>src/module_initialize_hill2d_x.F<br>src/module_initialize_quarter_ss.F<br>src/module_initialize_real.F<br>src/module_initialize_squall2d_x.F<br>src/module_initialize_squall2d_y.F<br>src/module_start.F<br>src/solve_rk.F | args_and_i1.pm |
| rk_dummy_arg_defines.inc | Definition statements for dummy argument list | src/module_dm.F<br>src/module_initialize.F<br>src/module_initialize_b_wave.F<br>src/module_initialize_hill2d_x.F<br>src/module_initialize_quarter_ss.F<br>src/module_initialize_real.F<br>src/module_initialize_squall2d_x.F<br>src/module_initialize_squall2d_y.F<br>src/module_start.F<br>src/solve_rk.F | args_and_i1.pm |
| rk_i1.inc | Definition of I1 (intermediate) fields | src/module_dm.F<br>src/solve_rk.F | args_and_i1.pm |
| state_namelist_defines.inc | Definitions of namelist variables | src/module_configure.F | config.pm |
| state_namelist_defines2.inc | Definitions without dimensions | src/module_configure.F | config.pm |
| state_namelist_statements.inc | NAMELIST statements for namelist variables | src/module_configure.F | config.pm |
| state_namelist_defaults.inc | Default values of namelist variables | src/module_configure.F | config.pm |
| state_namelist_assigns.inc | Statements for assigning namelist variables | src/module_configure.F | config.pm |
| state_namelist_reads.inc | Statements for reading namelist variables | src/module_configure. | config.pm |
| config_assign_*.inc | Statements for assigning namelist variables | src/add_config_info_to_grid.F<br>src/module_configure.F | config.pm |
| config_*.inc | Set/get subroutines for data in model_config_rec | src/module_configure.F | config.pm |
| set_scalar_indices.inc | Sets up indices into 4-D scalar arrays for a domain | src/module_configure.F | config.pm |
| module_state_description.F | Parameter statements used by model | | decls.pm |
| state_struct_items.inc | Definitions of fields in domain data structure | src/module_domain.F | decls.pm |
| rsl_cpp_flags | Definitions of RSL comm descriptors (page 27) | arch/configure.defaults | rsl_comms.pm |
| rslhalos.inc | RSL halo communications | src/module_dm.F | rsl_comms.pm |
| rslperiods.inc | RSL periodic boundary communcations | src/module_dm.F | rsl_comms.pm |
| rk_i1.inc | I1 (intermediate) field definitions for RK solver | src/module_dm.F<br>src/solve_rk.F | rsl_comms.pm |
| rsl_rk_data_calls.inc | Calls to register F90 state variables with RSL (page 28) | src/module_dm.F | rsl_comms.pm |

| | | | |
|---|---|---|---|
| | | src/module_initialize.F<br>src/module_initialize_b_wave.F<br>src/module_initialize_hill2d_x.F<br>src/module_initialize_quarter_ss.F<br>src/module_initialize_real.F<br>src/module_initialize_squall2d_x.F<br>src/module_initialize_squall2d_y.F<br>src/module_start.F<br>src/solve_rk.F | |
| wrf_histout.inc | Calls to output state to history | src/module_io_wrf.F | wrf_io.pm |
| wrf_restartout.inc | Calls to output state to restart | src/module_io_wrf.F | wrf_io.pm |
| wrf_initialout.inc | Calls to output state to initial input data | src/module_io_wrf.F | wrf_io.pm |
| wrf_bdyout.inc | Calls to output state to lateral boundary file | src/module_io_wrf.F | wrf_io.pm |
| wrf_histin.inc | Calls to input state from history | src/module_io_wrf.F | wrf_io.pm |
| wrf_restartin.inc | Calls to input state from restart | src/module_io_wrf.F | wrf_io.pm |
| wrf_initialin.inc | Calls to input state from initial input data | src/module_io_wrf.F | wrf_io.pm |
| wrf_bdyin.inc | Calls to input state from lateral boundary file | src/module_io_wrf.F | wrf_io.pm |

---

[1] The WRF model itself does not use subtypes; however, this was added to accommodate other Fortran90 programs that may be incorporated into the WRF framework in the future; for example, 3DVAR.

[2] When the field is an entry in a 4D array, the field name specified as the Sym entry (see above) is used to produce a symbolic (integer variable) index into the fourth dimension of the array. This symbol is P_<symname> and is defined in the registry-generated include file inc/scalar_indices.inc. Mediation and model layer subroutines may use these indices to access specific fields from the 4D arrays provided that the symname is also specified with a package (see Package Table) and the package is turned on via the namelist. Otherwise, no space is allocated in the 4D array for the field and the P_<symname> variable is set to an invalid value and should not be used.

[3] In the current WRF framework these arrays are declared 4D: the first dimension is the maximum of the two horizontal dimensions, the second dimension is the width of the boundary, (an integer namelist variable named spec_bdy_width -- user must define in Registry), and the third is the number of vertical layers (1 for 2D data). The fourth dimension is the index over boundaries: 1=boundary at start of X dimension (typically the west boundary), 2=boundary at end of X dimension (east), 3=boundary at start of Y dimension (south), and boundary at end of Y dimension (north). The boundary indices are defined as integer parameters P_XSB, P_XEB, P_YSB, and P_YEB in the file frame/module_state_description.F, which is generated by the registry.