The NetCDF Fortran 90 Interface Guide

NetCDF Version 4.1.2
29 March 2011

Robert Pincus
Russ Rew

Table of Contents

1 Use of the NetCDF Library 1

1.1 Creating a NetCDF Dataset 1
1.2 Reading a NetCDF Dataset with Known Names 2
1.3 Reading a netCDF Dataset with Unknown Names 2
1.4 Writing Data in an Existing NetCDF Dataset 3
1.5 Adding New Dimensions, Variables, Attributes 4
1.6 Error Handling 5
1.7 Compiling and Linking with the NetCDF Library 6

2 Datasets 7

2.1 Datasets Introduction 7
2.2 NetCDF Library Interface Descriptions 7
2.3 NF90_STRERROR 8
2.4 Get netCDF library version: NF90_INQ_LIBVERS 8
2.5 NF90_CREATE 9
2.6 NF90_OPEN 11
2.7 NF90_REDEF 13
2.8 NF90_ENDDEF 14
2.9 NF90_CLOSE 16
2.10 NF90_INQUIRE Family 16
2.11 NF90_SYNC 18
2.12 NF90_ABORT 19
2.13 NF90_SET_FILL 20

3 Groups 23

3.1 Find a Group ID: NF90_INQ_NCID 23
3.2 Get a List of Groups in a Group: NF90_INQ_GRPS 24
3.3 Find all the Variables in a Group: NF90_INQ_VARIDS 25
3.4 Find all Dimensions Visible in a Group: NF90_INQ_DIMIDS 25
3.5 Find the Length of a Group's Full Name: NF90_INQ_GRPNAME_LEN 26
3.6 Find a Group's Name: NF90_INQ_GRPNAME 27
3.7 Find a Group's Full Name: NF90_INQ_GRPNAME_FULL 28
3.8 Find a Group's Parent: NF90_INQ_GRP_PARENT 29
3.9 Find a Group by Name: NF90_INQ_GRP_NCID 30
3.10 Find a Group by its Fully-qualified Name: NF90_INQ_GRP_FULL_NCID 31
3.11 Create a New Group: NF90_DEF_GRP 32

4 Dimensions ... 35
4.1 Dimensions Introduction ... 35
4.2 NF90_DEF_DIM ... 35
4.3 NF90_INQ_DIMID ... 36
4.4 NF90_INQUIRE_DIMENSION ... 37
4.5 NF90_RENAME_DIM ... 38

5 User Defined Data Types ... 41
5.1 User Defined Types Introduction ... 41
5.2 Learn the IDs of All Types in Group: NF90_INQ_TYPEIDS ... 41
5.3 Find a Typeid from Group and Name: nf90_inq_typeid ... 42
5.4 Learn About a User Defined Type: NF90_INQ_TYPE ... 42
5.5 Learn About a User Defined Type: NF90_INQ_USER_TYPE ... 43
5.5.1 Set a Variable Length Array with NF90_PUT_VLEN_ELEMENT ... 44
5.5.2 Set a Variable Length Array with NF90_GET_VLEN_ELEMENT ... 45
5.6 Compound Types Introduction ... 46
5.6.1 Creating a Compound Type: NF90_DEF_COMPOUND ... 46
5.6.2 Inserting a Field into a Compound Type: NF90_INSERT_COMPOUND ... 48
5.6.3 Inserting an Array Field into a Compound Type: NF90_INSERT_ARRAY_COMPOUND ... 49
5.6.4 Learn About a Compound Type: NF90_INQ_COMPOUND ... 50
5.6.5 Learn About a Field of a Compound Type: NF90_INQ_COMPOUND_FIELD ... 51
5.7 Variable Length Array Introduction ... 53
5.7.1 Define a Variable Length Array (VLEN): NF90_DEF_VLEN ... 54
5.7.2 Learning about a Variable Length Array (VLEN) Type: NF90_INQ_VLEN ... 55
5.7.3 Releasing Memory for a Variable Length Array (VLEN) Type: NF90_FREE_VLEN ... 56
5.8 Opaque Type Introduction ... 56
5.8.1 Creating Opaque Types: NF90_DEF_OPAQUE ... 56
5.8.2 Learn About an Opaque Type: NF90_INQ_OPAQUE ... 57
5.9 Enum Type Introduction ... 58
5.9.1 Creating a Enum Type: NF90_DEF_ENUM ... 58
5.9.2 Inserting a Field into a Enum Type: NF90_INSERT_ENUM ... 59
5.9.3 Learn About a Enum Type: NF90_INQ_ENUM ... 60
5.9.4 Learn the Name of a Enum Type: nf90_inq_enum_member ... 61
5.9.5 Learn the Name of a Enum Type: NF90_INQ_ENUM_IDENT ... 62

6 Variables **63**
6.1 Variables Introduction 63
6.2 Language Types Corresponding to netCDF external data types 63
6.3 Create a Variable: `NF90_DEF_VAR` 64
6.4 Define Fill Parameters for a Variable: `nf90_def_var_fill` 68
6.5 Learn About Fill Parameters for a Variable: `NF90_INQ_VAR_FILL` 69
6.6 Get Information about a Variable from Its ID: NF90_INQUIRE_VARIABLE 70
6.7 Get the ID of a variable from the name: NF90_INQ_VARID 71
6.8 Writing Data Values: NF90_PUT_VAR 72
6.9 Reading Data Values: NF90_GET_VAR 77
6.10 Reading and Writing Character String Values 82
6.11 Fill Values 84
6.12 NF90_RENAME_VAR 85
6.13 Change between Collective and Independent Parallel Access: NF90_VAR_PAR_ACCESS 86

7 Attributes **89**
7.1 Attributes Introduction 89
7.2 Create an Attribute: NF90_PUT_ATT 89
7.3 Get Information about an Attribute: NF90_INQUIRE_ATTRIBUTE and NF90_INQ_ATTNAME 91
7.4 Get Attribute's Values: NF90_GET_ATT 93
7.5 Copy Attribute from One NetCDF to Another: NF90_COPY_ATT 94
7.6 Rename an Attribute: NF90_RENAME_ATT 96
7.7 NF90_DEL_ATT 97

Appendix A Appendix A - Summary of Fortran 90 Interface **99**

Appendix B Appendix B - FORTRAN 77 to Fortran 90 Transition Guide **103**
The new Fortran 90 interface 103
Changes to Inquiry functions 103
Changes to put and get function 103

Index **105**

1 Use of the NetCDF Library

You can use the netCDF library without knowing about all of the netCDF interface. If you are creating a netCDF dataset, only a handful of routines are required to define the necessary dimensions, variables, and attributes, and to write the data to the netCDF dataset. (Even less are needed if you use the ncgen utility to create the dataset before running a program using netCDF library calls to write data. See Section “ncgen” in *NetCDF Users Guide*.) Similarly, if you are writing software to access data stored in a particular netCDF object, only a small subset of the netCDF library is required to open the netCDF dataset and access the data. Authors of generic applications that access arbitrary netCDF datasets need to be familiar with more of the netCDF library.

In this chapter we provide templates of common sequences of netCDF calls needed for common uses. For clarity we present only the names of routines; omit declarations and error checking; omit the type-specific suffixes of routine names for variables and attributes; indent statements that are typically invoked multiple times; and use ... to represent arbitrary sequences of other statements. Full parameter lists are described in later chapters.

1.1 Creating a NetCDF Dataset

Here is a typical sequence of netCDF calls used to create a new netCDF dataset:

```
NF90_CREATE           ! create netCDF dataset: enter define mode
     ...
   NF90_DEF_DIM       ! define dimensions: from name and length
     ...
   NF90_DEF_VAR       ! define variables: from name, type, dims
     ...
   NF90_PUT_ATT       ! assign attribute values
     ...
NF90_ENDDEF           ! end definitions: leave define mode
     ...
   NF90_PUT_VAR       ! provide values for variable
     ...
NF90_CLOSE            ! close: save new netCDF dataset
```

Only one call is needed to create a netCDF dataset, at which point you will be in the first of two netCDF modes. When accessing an open netCDF dataset, it is either in define mode or data mode. In define mode, you can create dimensions, variables, and new attributes, but you cannot read or write variable data. In data mode, you can access data and change existing attributes, but you are not permitted to create new dimensions, variables, or attributes.

One call to NF90_DEF_DIM is needed for each dimension created. Similarly, one call to NF90_DEF_VAR is needed for each variable creation, and one call to a member of the NF90_PUT_ATT family is needed for each attribute defined and assigned a value. To leave define mode and enter data mode, call NF90_ENDDEF.

Once in data mode, you can add new data to variables, change old values, and change values of existing attributes (so long as the attribute changes do not require more storage space). Data of all types is written to a netCDF variable using the NF90_PUT_VAR subroutine. Single values, arrays, or array sections may be supplied to NF90_PUT_VAR; optional arguments allow the writing of subsampled or mapped portions of the variable. (Subsampled and mapped access are general forms of data access that are explained later.)

Finally, you should explicitly close all netCDF datasets that have been opened for writing by calling NF90_CLOSE. By default, access to the file system is buffered by the netCDF library. If a program terminates abnormally with netCDF datasets open for writing, your most recent modifications may be lost. This default buffering of data is disabled by setting the NF90_SHARE flag when opening the dataset. But even if this flag is set, changes to attribute values or changes made in define mode are not written out until NF90_SYNC or NF90_CLOSE is called.

1.2 Reading a NetCDF Dataset with Known Names

Here we consider the case where you know the names of not only the netCDF datasets, but also the names of their dimensions, variables, and attributes. (Otherwise you would have to do "inquire" calls.) The order of typical C calls to read data from those variables in a netCDF dataset is:

```
NF90_OPEN               ! open existing netCDF dataset
     ...
   NF90_INQ_DIMID       ! get dimension IDs
     ...
   NF90_INQ_VARID       ! get variable IDs
     ...
   NF90_GET_ATT         ! get attribute values
     ...
   NF90_GET_VAR         ! get values of variables
     ...
NF90_CLOSE              ! close netCDF dataset
```

First, a single call opens the netCDF dataset, given the dataset name, and returns a netCDF ID that is used to refer to the open netCDF dataset in all subsequent calls.

Next, a call to NF90_INQ_DIMID for each dimension of interest gets the dimension ID from the dimension name. Similarly, each required variable ID is determined from its name by a call to NF90_INQ_VARID. Once variable IDs are known, variable attribute values can be retrieved using the netCDF ID, the variable ID, and the desired attribute name as input to NF90_GET_ATT for each desired attribute. Variable data values can be directly accessed from the netCDF dataset with calls to NF90_GET_VAR.

Finally, the netCDF dataset is closed with NF90_CLOSE. There is no need to close a dataset open only for reading.

1.3 Reading a netCDF Dataset with Unknown Names

It is possible to write programs (e.g., generic software) which do such things as processing every variable, without needing to know in advance the names of these variables. Similarly, the names of dimensions and attributes may be unknown.

Names and other information about netCDF objects may be obtained from netCDF datasets by calling inquire functions. These return information about a whole netCDF dataset, a dimension, a variable, or an attribute. The following template illustrates how they are used:

```
NF90_OPEN                   ! open existing netCDF dataset
  ...
NF90_INQUIRE                ! find out what is in it
     ...
   NF90_INQUIRE_DIMENSION ! get dimension names, lengths
     ...
   NF90_INQUIRE_VARIABLE  ! get variable names, types, shapes
        ...
      NF90_INQ_ATTNAME    ! get attribute names
        ...
      NF90_INQUIRE_ATTRIBUTE ! get other attribute information
        ...
      NF90_GET_ATT        ! get attribute values
        ...
   NF90_GET_VAR           ! get values of variables
     ...
NF90_CLOSE                ! close netCDF dataset
```

As in the previous example, a single call opens the existing netCDF dataset, returning a netCDF ID. This netCDF ID is given to the NF90_INQUIRE routine, which returns the number of dimensions, the number of variables, the number of global attributes, and the ID of the unlimited dimension, if there is one.

All the inquire functions are inexpensive to use and require no I/O, since the information they provide is stored in memory when a netCDF dataset is first opened.

Dimension IDs use consecutive integers, beginning at 1. Also dimensions, once created, cannot be deleted. Therefore, knowing the number of dimension IDs in a netCDF dataset means knowing all the dimension IDs: they are the integers 1, 2, 3, ...up to the number of dimensions. For each dimension ID, a call to the inquire function NF90_INQUIRE_DIMENSION returns the dimension name and length.

Variable IDs are also assigned from consecutive integers 1, 2, 3, ... up to the number of variables. These can be used in NF90_INQUIRE_VARIABLE calls to find out the names, types, shapes, and the number of attributes assigned to each variable.

Once the number of attributes for a variable is known, successive calls to NF90_INQ_ATTNAME return the name for each attribute given the netCDF ID, variable ID, and attribute number. Armed with the attribute name, a call to NF90_INQUIRE_ATTRIBUTE returns its type and length. Given the type and length, you can allocate enough space to hold the attribute values. Then a call to NF90_GET_ATT returns the attribute values.

Once the IDs and shapes of netCDF variables are known, data values can be accessed by calling NF90_GET_VAR.

1.4 Writing Data in an Existing NetCDF Dataset

With write access to an existing netCDF dataset, you can overwrite data values in existing variables or append more data to record variables along the unlimited (record) dimension.

To append more data to non-record variables requires changing the shape of such variables, which means creating a new netCDF dataset, defining new variables with the desired shape, and copying data. The netCDF data model was not designed to make such "schema changes" efficient or easy, so it is best to specify the shapes of variables correctly when you create a netCDF dataset, and to anticipate which variables will later grow by using the unlimited dimension in their definition.

The following code template lists a typical sequence of calls to overwrite some existing values and add some new records to record variables in an existing netCDF dataset with known variable names:

```
NF90_OPEN             ! open existing netCDF dataset
  ...
  NF90_INQ_VARID      ! get variable IDs
  ...
  NF90_PUT_VAR        ! provide new values for variables, if any
  ...
  NF90_PUT_ATT        ! provide new values for attributes, if any
    ...
NF90_CLOSE            ! close netCDF dataset
```

A netCDF dataset is first opened by the NF90_OPEN call. This call puts the open dataset in data mode, which means existing data values can be accessed and changed, existing attributes can be changed, but no new dimensions, variables, or attributes can be added.

Next, calls to NF90_INQ_VARID get the variable ID from the name, for each variable you want to write. Then each call to NF90_PUT_VAR writes data into a specified variable, either a single value at a time, or a whole set of values at a time, depending on which variant of the interface is used. The calls used to overwrite values of non-record variables are the same as are used to overwrite values of record variables or append new data to record variables. The difference is that, with record variables, the record dimension is extended by writing values that don't yet exist in the dataset. This extends all record variables at once, writing "fill values" for record variables for which the data has not yet been written (but see Section 6.11 [Fill Values], page 84 to specify different behavior).

Calls to NF90_PUT_ATT may be used to change the values of existing attributes, although data that changes after a file is created is typically stored in variables rather than attributes.

Finally, you should explicitly close any netCDF datasets into which data has been written by calling NF90_CLOSE before program termination. Otherwise, modifications to the dataset may be lost.

1.5 Adding New Dimensions, Variables, Attributes

An existing netCDF dataset can be extensively altered. New dimensions, variables, and attributes can be added or existing ones renamed, and existing attributes can be deleted. Existing dimensions, variables, and attributes can be renamed. The following code template lists a typical sequence of calls to add new netCDF components to an existing dataset:

```
NF90_OPEN             ! open existing netCDF dataset
  ...
```

```
NF90_REDEF              ! put it into define mode
    ...
  NF90_DEF_DIM          ! define additional dimensions (if any)
    ...
  NF90_DEF_VAR          ! define additional variables (if any)
    ...
  NF90_PUT_ATT          ! define other attributes (if any)
    ...
NF90_ENDDEF             ! check definitions, leave define mode
    ...
  NF90_PUT_VAR          ! provide new variable values
    ...
NF90_CLOSE              ! close netCDF dataset
```

A netCDF dataset is first opened by the NF90_OPEN call. This call puts the open dataset in data mode, which means existing data values can be accessed and changed, existing attributes can be changed (so long as they do not grow), but nothing can be added. To add new netCDF dimensions, variables, or attributes you must enter define mode, by calling NF90_REDEF. In define mode, call NF90_DEF_DIM to define new dimensions, NF90_DEF_VAR to define new variables, and NF90_PUT_ATT to assign new attributes to variables or enlarge old attributes.

You can leave define mode and reenter data mode, checking all the new definitions for consistency and committing the changes to disk, by calling NF90_ENDDEF. If you do not wish to reenter data mode, just call NF90_CLOSE, which will have the effect of first calling NF90_ENDDEF.

Until the NF90_ENDDEF call, you may back out of all the redefinitions made in define mode and restore the previous state of the netCDF dataset by calling NF90_ABORT. You may also use the NF90_ABORT call to restore the netCDF dataset to a consistent state if the call to NF90_ENDDEF fails. If you have called NF90_CLOSE from definition mode and the implied call to NF90_ENDDEF fails, NF90_ABORT will automatically be called to close the netCDF dataset and leave it in its previous consistent state (before you entered define mode).

At most one process should have a netCDF dataset open for writing at one time. The library is designed to provide limited support for multiple concurrent readers with one writer, via disciplined use of the NF90_SYNC function and the NF90_SHARE flag. If a writer makes changes in define mode, such as the addition of new variables, dimensions, or attributes, some means external to the library is necessary to prevent readers from making concurrent accesses and to inform readers to call NF90_SYNC before the next access.

1.6 Error Handling

The netCDF library provides the facilities needed to handle errors in a flexible way. Each netCDF function returns an integer status value. If the returned status value indicates an error, you may handle it in any way desired, from printing an associated error message and exiting to ignoring the error indication and proceeding (not recommended!). For simplicity, the examples in this guide check the error status and call a separate function to handle any errors.

The NF90_STRERROR function is available to convert a returned integer error status into an error message string.

Occasionally, low-level I/O errors may occur in a layer below the netCDF library. For example, if a write operation causes you to exceed disk quotas or to attempt to write to a device that is no longer available, you may get an error from a layer below the netCDF library, but the resulting write error will still be reflected in the returned status value.

1.7 Compiling and Linking with the NetCDF Library

Details of how to compile and link a program that uses the netCDF C or Fortran interfaces differ, depending on the operating system, the available compilers, and where the netCDF library and include files are installed.

Every Fortran 90 procedure or module which references netCDF constants or procedures must have access to the module information created when the netCDF module was compiled. The suffix for this file is "MOD" (or sometimes "mod").

Most F90 compilers allow the user to specify the location of .MOD files, usually with the -I flag. (Some compilers, like absoft, use -p instead).

```
f90 -c -I/usr/local/include mymodule.f90
```

Starting with version 3.6.2, another method of building the netCDF fortran libraries becomes available. With the –enable-separate-fortran option to configure, the user can specify that the C library should not contain the fortran functions. In these cases an additional library, libnetcdff.a (not the extra "f") will be built. This library contains the fortran functions.

For more information about configure options, See Section "Specifying the Environment for Building" in *The NetCDF Installation and Porting Guide*.

Building separate fortran libraries is required for shared library builds, but is not done, by default, for static library builds.

When linking fortran programs without a separate fortran library, programs must link to the netCDF library like this:

```
f90 -o myprogram myprogram.o -L/usr/local/netcdf/lib -lnetcdf
```

2 Datasets

2.1 Datasets Introduction

This chapter presents the interfaces of the netCDF functions that deal with a netCDF dataset or the whole netCDF library.

A netCDF dataset that has not yet been opened can only be referred to by its dataset name. Once a netCDF dataset is opened, it is referred to by a netCDF ID, which is a small nonnegative integer returned when you create or open the dataset. A netCDF ID is much like a file descriptor in C or a logical unit number in FORTRAN. In any single program, the netCDF IDs of distinct open netCDF datasets are distinct. A single netCDF dataset may be opened multiple times and will then have multiple distinct netCDF IDs; however at most one of the open instances of a single netCDF dataset should permit writing. When an open netCDF dataset is closed, the ID is no longer associated with a netCDF dataset.

Functions that deal with the netCDF library include:

- Get version of library.
- Get error message corresponding to a returned error code.

The operations supported on a netCDF dataset as a single object are:

- Create, given dataset name and whether to overwrite or not.
- Open for access, given dataset name and read or write intent.
- Put into define mode, to add dimensions, variables, or attributes.
- Take out of define mode, checking consistency of additions.
- Close, writing to disk if required.
- Inquire about the number of dimensions, number of variables, number of global attributes, and ID of the unlimited dimension, if any.
- Synchronize to disk to make sure it is current.
- Set and unset nofill mode for optimized sequential writes.
- After a summary of conventions used in describing the netCDF interfaces, the rest of this chapter presents a detailed description of the interfaces for these operations.

2.2 NetCDF Library Interface Descriptions

Each interface description for a particular netCDF function in this and later chapters contains:

- a description of the purpose of the function;
- a Fortran 90 interface block that presents the type and order of the formal parameters to the function;
- a description of each formal parameter in the C interface;
- a list of possible error conditions; and
- an example of a Fortran 90 program fragment calling the netCDF function (and perhaps other netCDF functions).

The examples follow a simple convention for error handling, always checking the error status returned from each netCDF function call and calling a handle_error function in case an error was detected. For an example of such a function, see Section 5.2 "Get error message corresponding to error status: nf90_strerror".

2.3 NF90_STRERROR

The function NF90_STRERROR returns a static reference to an error message string corresponding to an integer netCDF error status or to a system error number, presumably returned by a previous call to some other netCDF function. The list of netCDF error status codes is available in the appropriate include file for each language binding.

Usage

```
function nf90_strerror(ncerr)
  integer, intent( in) :: ncerr
  character(len = 80)  :: nf90_strerror
```

`NCERR` An error status that might have been returned from a previous call to some netCDF function.

Errors

If you provide an invalid integer error status that does not correspond to any netCDF error message or or to any system error message (as understood by the system strerror function), NF90_STRERROR returns a string indicating that there is no such error status.

Example

Here is an example of a simple error handling function that uses NF90_STRERROR to print the error message corresponding to the netCDF error status returned from any netCDF function call and then exit:

```
subroutine handle_err(status)
  integer, intent ( in) :: status

  if(status /= nf90_noerr) then
    print *, trim(nf90_strerror(status))
    stop "Stopped"
  end if
end subroutine handle_err
```

2.4 Get netCDF library version: NF90_INQ_LIBVERS

The function NF90_INQ_LIBVERS returns a string identifying the version of the netCDF library, and when it was built.

Usage

```
function nf90_inq_libvers()
  character(len = 80) :: nf90_inq_libvers
```

Errors

This function takes no arguments, and returns no error status.

Example

Here is an example using nf90_inq_libvers to print the version of the netCDF library with which the program is linked:

```
print *, trim(nf90_inq_libvers())
```

2.5 NF90_CREATE

This function creates a new netCDF dataset, returning a netCDF ID that can subsequently be used to refer to the netCDF dataset in other netCDF function calls. The new netCDF dataset opened for write access and placed in define mode, ready for you to add dimensions, variables, and attributes.

A creation mode flag specifies whether to overwrite any existing dataset with the same name and whether access to the dataset is shared.

Usage

```
function nf90_create(path, cmode, ncid, initialsize, bufrsize, cache_size, &
     cache_nelems, cache_preemption, comm, info)
  implicit none
  character (len = *), intent(in) :: path
  integer, intent(in) :: cmode
  integer, intent(out) :: ncid
  integer, optional, intent(in) :: initialsize
  integer, optional, intent(inout) :: bufrsize
  integer, optional, intent(in) :: cache_size, cache_nelems
  real, optional, intent(in) :: cache_preemption
  integer, optional, intent(in) :: comm, info
  integer :: nf90_create
```

`path` The file name of the new netCDF dataset.

`cmode` The creation mode flag. The following flags are available: NF90_NOCLOBBER, NF90_SHARE, NF90_64BIT_OFFSET, NF90_HDF5, and NF90_CLASSIC_MODEL.

A zero value (defined for convenience as NF90_CLOBBER) specifies the default behavior: overwrite any existing dataset with the same file name and buffer and cache accesses for efficiency. The dataset will be in netCDF classic format. See Section “NetCDF Classic Format Limitations” in *NetCDF Users' Guide*.

Setting NF90_NOCLOBBER means you do not want to clobber (overwrite) an existing dataset; an error (NF90_EEXIST) is returned if the specified dataset already exists.

The NF90_SHARE flag is appropriate when one process may be writing the dataset and one or more other processes reading the dataset concurrently; it means that dataset accesses are not buffered and caching is limited. Since the buffering scheme is optimized for sequential access, programs that do not access data sequentially may see some performance improvement by setting the NF90_SHARE flag. (This only applies to netCDF-3 classic or 64-bit offset files.)

Setting NF90_64BIT_OFFSET causes netCDF to create a 64-bit offset format file, instead of a netCDF classic format file. The 64-bit offset format imposes far fewer restrictions on very large (i.e. over 2 GB) data files. See Section "Large File Support" in *NetCDF Users' Guide*.

Setting the NF90_HDF5 flag causes netCDF to create a netCDF-4/HDF5 format output file.

Oring the NF90_CLASSIC_MODEL flag with the NF90_HDF5 flag causes the resulting netCDF-4/HDF5 file to restrict itself to the classic model - none of the new netCDF-4 data model features, such as groups or user-defined types, are allowed in such a file.

`ncid` Returned netCDF ID.

The following optional arguments allow additional performance tuning.

`initialsize`
: The initial size of the file (in bytes) at creation time. A value of 0 causes the file size to be computed when nf90_enddef is called. This is ignored for NetCDF-4/HDF5 files.

`bufrsize`
: Controls a space versus time trade-off, memory allocated in the netcdf library versus number of system calls. Because of internal requirements, the value may not be set to exactly the value requested. The actual value chosen is returned.

The library chooses a system-dependent default value if NF90_SIZEHINT_DEFAULT is supplied as input. If the "preferred I/O block size" is available from the stat() system call as member st_blksize this value is used. Lacking that, twice the system pagesize is used. Lacking a call to discover the system pagesize, the default bufrsize is set to 8192 bytes.

The bufrsize is a property of a given open netcdf descriptor ncid, it is not a persistent property of the netcdf dataset.

This is ignored for NetCDF-4/HDF5 files.

`cache_size`
: If the cache_size is provided when creating a netCDF-4/HDF5 file, it will be used instead of the default (32000000) as the size, in bytes, of the HDF5 chunk cache.

`cache_nelems`
: If cache_nelems is provided when creating a netCDF-4/HDF5 file, it will be used instead of the default (1000) as the maximum number of elements in the HDF5 chunk cache.

`cache_premtion`
: If cache_preemption is provided when creating a netCDF-4/HDF5 file, it will be used instead of the default (0.75) as the preemption value for the HDF5 chunk cache.

`comm` If the comm and info parameters are provided the file is created and opened for parallel I/O. Set the comm parameter to the MPI communicator (of type MPI_Comm). If this parameter is provided the info parameter must also be provided.

`info` If the comm and info parameters are provided the file is created and opened for parallel I/O. Set the comm parameter to the MPI information value (of type MPI_Info). If this parameter is provided the comm parameter must also be provided.

Errors

NF90_CREATE returns the value NF90_NOERR if no errors occurred. Possible causes of errors include:

- Passing a dataset name that includes a directory that does not exist.
- Specifying a dataset name of a file that exists and also specifying NF90_NOCLOBBER.
- Specifying a meaningless value for the creation mode.
- Attempting to create a netCDF dataset in a directory where you don't have permission to create files.

Example

In this example we create a netCDF dataset named foo.nc; we want the dataset to be created in the current directory only if a dataset with that name does not already exist:

```
use netcdf
implicit none
integer :: ncid, status
...
status = nf90_create(path = "foo.nc", cmode = nf90_noclobber, ncid = ncid)
if (status /= nf90_noerr) call handle_err(status)
```

2.6 NF90_OPEN

The function NF90_OPEN opens an existing netCDF dataset for access.

Usage

```
 function nf90_open(path, mode, ncid, bufrsize, cache_size, cache_nelems, &
                    cache_preemption, comm, info)
   implicit none
   character (len = *), intent(in) :: path
   integer, intent(in) :: mode
   integer, intent(out) :: ncid
   integer, optional, intent(inout) :: bufrsize
   integer, optional, intent(in) :: cache_size, cache_nelems
   real, optional, intent(in) :: cache_preemption
   integer, optional, intent(in) :: comm, info
   integer :: nf90_open
```

`path` File name for netCDF dataset to be opened. This may be an OPeNDAP URL if DAP support is enabled.

`mode` A zero value (or NF90_NOWRITE) specifies the default behavior: open the dataset with read-only access, buffering and caching accesses for efficiency

Otherwise, the open mode is NF90_WRITE, NF90_SHARE, or NF90_WRITE|NF90_SHARE. Setting the NF90_WRITE flag opens the dataset with read-write access. ("Writing" means any kind of change to the dataset, including appending or changing data, adding or renaming dimensions, variables, and attributes, or deleting attributes.) The NF90_SHARE flag is appropriate when one process may be writing the dataset and one or more other processes reading the dataset concurrently (note that this is not the same as parallel I/O); it means that dataset accesses are not buffered and caching is limited. Since the buffering scheme is optimized for sequential access, programs that do not access data sequentially may see some performance improvement by setting the NF90_SHARE flag.

`ncid` Returned netCDF ID.

The following optional argument allows additional performance tuning.

`bufrsize` This parameter applies only when opening classic format or 64-bit offset files. It is ignored for netCDF-4/HDF5 files.

It Controls a space versus time trade-off, memory allocated in the netcdf library versus number of system calls. Because of internal requirements, the value may not be set to exactly the value requested. The actual value chosen is returned.

The library chooses a system-dependent default value if NF90_SIZEHINT_DEFAULT is supplied as input. If the "preferred I/O block size" is available from the stat() system call as member st_blksize this value is used. Lacking that, twice the system pagesize is used. Lacking a call to discover the system pagesize, the default bufrsize is set to 8192 bytes.

The bufrsize is a property of a given open netcdf descriptor ncid, it is not a persistent property of the netcdf dataset.

`cache_size`
If the cache_size is provided when opening a netCDF-4/HDF5 file, it will be used instead of the default (32000000) as the size, in bytes, of the HDF5 chunk cache.

`cache_nelems`
If cache_nelems is provided when opening a netCDF-4/HDF5 file, it will be used instead of the default (1000) as the maximum number of elements in the HDF5 chunk cache.

`cache_premtion`
If cache_preemption is provided when opening a netCDF-4/HDF5 file, it will be used instead of the default (0.75) as the preemption value for the HDF5 chunk cache.

`comm` If the comm and info parameters are provided the file is opened for parallel I/O. Set the comm parameter to the MPI communicator (of type MPI_Comm). If this parameter is provided the info parameter must also be provided.

`info` If the comm and info parameters are provided the file is opened for parallel I/O. Set the comm parameter to the MPI information value (of type MPI_Info). If this parameter is provided the comm parameter must also be provided.

Errors

NF90_OPEN returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified netCDF dataset does not exist.
- A meaningless mode was specified.

Example

Here is an example using NF90_OPEN to open an existing netCDF dataset named foo.nc for read-only, non-shared access:

```
use netcdf
implicit none
integer :: ncid, status
...
status = nf90_open(path = "foo.nc", cmode = nf90_nowrite, ncid = ncid)
if (status /= nf90_noerr) call handle_err(status)
```

Example

Here is an example using NF90_OPEN to open an existing netCDF dataset for parallel I/O access. (Note the use of the comm and info parameters). This example is from test program nf_test/f90tst_parallel.f90.

```
use netcdf
implicit none
integer :: ncid, status
...
 ! Reopen the file.
 call handle_err(nf90_open(FILE_NAME, nf90_nowrite, ncid, comm = MPI_COMM_WORLD, &
      info = MPI_INFO_NULL))
```

2.7 NF90_REDEF

The function NF90_REDEF puts an open netCDF dataset into define mode, so dimensions, variables, and attributes can be added or renamed and attributes can be deleted.

Usage

```
function nf90_redef(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_redef
```

`ncid` netCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

Errors

NF90_REDEF returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified netCDF dataset is already in define mode.
- The specified netCDF dataset was opened for read-only.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_REDEF to open an existing netCDF dataset named foo.nc and put it into define mode:

```
use netcdf
implicit none
integer :: ncid, status
...
status = nf90_open("foo.nc", nf90_write, ncid) ! Open dataset
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_redef(ncid)                       ! Put the file in define mode
if (status /= nf90_noerr) call handle_err(status)
```

2.8 NF90_ENDDEF

The function NF90_ENDDEF takes an open netCDF dataset out of define mode. The changes made to the netCDF dataset while it was in define mode are checked and committed to disk if no problems occurred. Non-record variables may be initialized to a "fill value" as well (see Section 2.13 [NF90_SET_FILL], page 20). The netCDF dataset is then placed in data mode, so variable data can be read or written.

This call may involve copying data under some circumstances. For a more extensive discussion See Section "File Structure and Performance" in *NetCDF Users Guide*.

Usage

```
function nf90_enddef(ncid, h_minfree, v_align, v_minfree, r_align)
  integer,           intent( in) :: ncid
  integer, optional, intent( in) :: h_minfree, v_align, v_minfree, r_align
  integer                        :: nf90_enddef
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

The following arguments allow additional performance tuning. Note: these arguments expose internals of the netcdf version 1 file format, and may not be available in future netcdf implementations.

The current netcdf file format has three sections: the "header" section, the data section for fixed size variables, and the data section for variables which have an unlimited dimension (record variables). The header begins at the beginning of the file. The index (offset) of the beginning of the other two sections is contained in the header. Typically, there is no space between the sections. This causes copying overhead to accrue if one wishes to change the size of the sections, as may happen when changing the names of things, text attribute values, adding attributes or adding variables. Also, for buffered i/o, there may be advantages to aligning sections in certain ways.

The minfree parameters allow one to control costs of future calls to nf90_redef or nf90_enddef by requesting that some space be available at the end of the section. The default value for both h_minfree and v_minfree is 0.

The align parameters allow one to set the alignment of the beginning of the corresponding sections. The beginning of the section is rounded up to an index which is a multiple of the align parameter. The flag value NF90_ALIGN_CHUNK tells the library to use the bufrsize (see above) as the align parameter. The default value for both v_align and r_align is 4 bytes.

`h_minfree`
: Size of the pad (in bytes) at the end of the "header" section.

`v_minfree`
: Size of the pad (in bytes) at the end of the data section for fixed size variables.

`v_align`
: The alignment of the beginning of the data section for fixed size variables.

`r_align`
: The alignment of the beginning of the data section for variables which have an unlimited dimension (record variables).

Errors

NF90_ENDDEF returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified netCDF dataset is not in define mode.
- The specified netCDF ID does not refer to an open netCDF dataset.
- The size of one or more variables exceed the size constraints for whichever variant of the file format is in use). See Section "Large File Support" in *The NetCDF Users Guide*.

Example

Here is an example using NF90_ENDDEF to finish the definitions of a new netCDF dataset named foo.nc and put it into data mode:

```
use netcdf
implicit none
integer :: ncid, status
...
status = nf90_create("foo.nc", nf90_noclobber, ncid)
if (status /= nf90_noerr) call handle_err(status)
...  !  create dimensions, variables, attributes
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

2.9 NF90_CLOSE

The function NF90_CLOSE closes an open netCDF dataset. If the dataset is in define mode, NF90_ENDDEF will be called before closing. (In this case, if NF90_ENDDEF returns an error, NF90_ABORT will automatically be called to restore the dataset to the consistent state before define mode was last entered.) After an open netCDF dataset is closed, its netCDF ID may be reassigned to the next netCDF dataset that is opened or created.

Usage

```
function nf90_close(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_close
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

Errors

NF90_CLOSE returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- Define mode was entered and the automatic call made to NF90_ENDDEF failed.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_CLOSE to finish the definitions of a new netCDF dataset named foo.nc and release its netCDF ID:

```
use netcdf
implicit none
integer :: ncid, status
...
status = nf90_create("foo.nc", nf90_noclobber, ncid)
if (status /= nf90_noerr) call handle_err(status)
... !  create dimensions, variables, attributes
status = nf90_close(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

2.10 NF90_INQUIRE Family

The NF90_INQUIRE subroutine returns information about an open netCDF dataset, given its netCDF ID. The subroutine can be called from either define mode or data mode, and returns values for any or all of the following: the number of dimensions, the number of variables, the number of global attributes, and the dimension ID of the dimension defined with unlimited length, if any. An additional function, NF90_INQ_FORMAT, returns the (rarely needed) format version.

No I/O is performed when NF90_INQUIRE is called, since the required information is available in memory for each open netCDF dataset.

Usage

```
function nf90_inquire(ncid, nDimensions, nVariables, nAttributes, &
                      unlimitedDimId, formatNum)
  integer,           intent( in) :: ncid
  integer, optional, intent(out) :: nDimensions, nVariables, &
                                    nAttributes, unlimitedDimId, &
                                    formatNum
  integer                        :: nf90_inquire
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`nDimensions`
Returned number of dimensions defined for this netCDF dataset.

`nVariables`
Returned number of variables defined for this netCDF dataset.

`nAttributes`
Returned number of global attributes defined for this netCDF dataset.

`unlimitedDimID`
Returned ID of the unlimited dimension, if there is one for this netCDF dataset. If no unlimited length dimension has been defined, -1 is returned.

`format` Returned integer indicating format version for this dataset, one of nf90_format_classic, nf90_format_64bit, nf90_format_netcdf4, or nf90_format_netcdf4_classic. These are rarely needed by users or applications, since thhe library recognizes the format of a file it is accessing and handles it accordingly.

Errors

Function NF90_INQUIRE returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_INQUIRE to find out about a netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, nDims, nVars, nGlobalAtts, unlimDimID
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_inquire(ncid, nDims, nVars, nGlobalAtts, unlimdimid)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_inquire(ncid, nDimensions = nDims, &
```

```
                            unlimitedDimID = unlimdimid)
      if (status /= nf90_noerr) call handle_err(status)
```

2.11 NF90_SYNC

The function NF90_SYNC offers a way to synchronize the disk copy of a netCDF dataset with in-memory buffers. There are two reasons you might want to synchronize after writes:

- To minimize data loss in case of abnormal termination, or
- To make data available to other processes for reading immediately after it is written. But note that a process that already had the dataset open for reading would not see the number of records increase when the writing process calls NF90_SYNC; to accomplish this, the reading process must call NF90_SYNC.

This function is backward-compatible with previous versions of the netCDF library. The intent was to allow sharing of a netCDF dataset among multiple readers and one writer, by having the writer call NF90_SYNC after writing and the readers call NF90_SYNC before each read. For a writer, this flushes buffers to disk. For a reader, it makes sure that the next read will be from disk rather than from previously cached buffers, so that the reader will see changes made by the writing process (e.g., the number of records written) without having to close and reopen the dataset. If you are only accessing a small amount of data, it can be expensive in computer resources to always synchronize to disk after every write, since you are giving up the benefits of buffering.

An easier way to accomplish sharing (and what is now recommended) is to have the writer and readers open the dataset with the NF90_SHARE flag, and then it will not be necessary to call NF90_SYNC at all. However, the NF90_SYNC function still provides finer granularity than the NF90_SHARE flag, if only a few netCDF accesses need to be synchronized among processes.

It is important to note that changes to the ancillary data, such as attribute values, are not propagated automatically by use of the NF90_SHARE flag. Use of the NF90_SYNC function is still required for this purpose.

Sharing datasets when the writer enters define mode to change the data schema requires extra care. In previous releases, after the writer left define mode, the readers were left looking at an old copy of the dataset, since the changes were made to a new copy. The only way readers could see the changes was by closing and reopening the dataset. Now the changes are made in place, but readers have no knowledge that their internal tables are now inconsistent with the new dataset schema. If netCDF datasets are shared across redefinition, some mechanism external to the netCDF library must be provided that prevents access by readers during redefinition and causes the readers to call NF90_SYNC before any subsequent access.

When calling NF90_SYNC, the netCDF dataset must be in data mode. A netCDF dataset in define mode is synchronized to disk only when NF90_ENDDEF is called. A process that is reading a netCDF dataset that another process is writing may call NF90_SYNC to get updated with the changes made to the data by the writing process (e.g., the number of records written), without having to close and reopen the dataset.

Data is automatically synchronized to disk when a netCDF dataset is closed, or whenever you leave define mode.

Usage

```
function nf90_sync(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_sync
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

Errors

NF90_SYNC returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The netCDF dataset is in define mode.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_SYNC to synchronize the disk writes of a netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status
...
status = nf90_open("foo.nc", nf90_write, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! write data or change attributes
...
status = NF90_SYNC(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

2.12 NF90_ABORT

You no longer need to call this function, since it is called automatically by NF90_CLOSE in case the dataset is in define mode and something goes wrong with committing the changes. The function NF90_ABORT just closes the netCDF dataset, if not in define mode. If the dataset is being created and is still in define mode, the dataset is deleted. If define mode was entered by a call to NF90_REDEF, the netCDF dataset is restored to its state before definition mode was entered and the dataset is closed.

Usage

```
function nf90_abort(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_abort
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

Errors

NF90_ABORT returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- When called from define mode while creating a netCDF dataset, deletion of the dataset failed.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_ABORT to back out of redefinitions of a dataset named foo.nc:

```
 use netcdf
 implicit none
 integer :: ncid, status, LatDimID
 ...
 status = nf90_open("foo.nc", nf90_write, ncid)
 if (status /= nf90_noerr) call handle_err(status)
 ...
 status = nf90_redef(ncid)
 if (status /= nf90_noerr) call handle_err(status)
 ...
 status = nf90_def_dim(ncid, "Lat", 18, LatDimID)
 if (status /= nf90_noerr) then ! Dimension definition failed
   call handle_err(status)
   status = nf90_abort(ncid) ! Abort redefinitions
   if (status /= nf90_noerr) call handle_err(status)
 end if
...
```

2.13 NF90_SET_FILL

This function is intended for advanced usage, to optimize writes under some circumstances described below. The function NF90_SET_FILL sets the fill mode for a netCDF dataset open for writing and returns the current fill mode in a return parameter. The fill mode can be specified as either NF90_FILL or NF90_NOFILL. The default behavior corresponding to NF90_FILL is that data is pre-filled with fill values, that is fill values are written when you create non-record variables or when you write a value beyond data that has not yet been written. This makes it possible to detect attempts to read data before it was written. See Section 6.11 [Fill Values], page 84, for more information on the use of fill values. See Section "Attribute Conventions" in *The NetCDF Users Guide*, for information about how to define your own fill values.

The behavior corresponding to NF90_NOFILL overrides the default behavior of prefilling data with fill values. This can be used to enhance performance, because it avoids the duplicate writes that occur when the netCDF library writes fill values that are later overwritten with data.

A value indicating which mode the netCDF dataset was already in is returned. You can use this value to temporarily change the fill mode of an open netCDF dataset and then restore it to the previous mode.

After you turn on NF90_NOFILL mode for an open netCDF dataset, you must be certain to write valid data in all the positions that will later be read. Note that nofill mode is only a transient property of a netCDF dataset open for writing: if you close and reopen the dataset, it will revert to the default behavior. You can also revert to the default behavior by calling NF90_SET_FILL again to explicitly set the fill mode to NF90_FILL.

There are three situations where it is advantageous to set nofill mode:

1. Creating and initializing a netCDF dataset. In this case, you should set nofill mode before calling NF90_ENDDEF and then write completely all non-record variables and the initial records of all the record variables you want to initialize.
2. Extending an existing record-oriented netCDF dataset. Set nofill mode after opening the dataset for writing, then append the additional records to the dataset completely, leaving no intervening unwritten records.
3. Adding new variables that you are going to initialize to an existing netCDF dataset. Set nofill mode before calling NF90_ENDDEF then write all the new variables completely.

If the netCDF dataset has an unlimited dimension and the last record was written while in nofill mode, then the dataset may be shorter than if nofill mode was not set, but this will be completely transparent if you access the data only through the netCDF interfaces.

The use of this feature may not be available (or even needed) in future releases. Programmers are cautioned against heavy reliance upon this feature.

Usage

```
function nf90_set_fill(ncid, fillmode, old_mode)
  integer, intent( in) :: ncid, fillmode
  integer, intent(out) :: old_mode
  integer              :: nf90_set_fill
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`fillmode` Desired fill mode for the dataset, either NF90_NOFILL or NF90_FILL.

`old_mode` Returned current fill mode of the dataset before this call, either NF90_NOFILL or NF90_FILL.

Errors

NF90_SET_FILL returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified netCDF ID does not refer to an open netCDF dataset.
- The specified netCDF ID refers to a dataset open for read-only access.
- The fill mode argument is neither NF90_NOFILL nor NF90_FILL..

Example

Here is an example using NF90_SET_FILL to set nofill mode for subsequent writes of a netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, oldMode
...
status = nf90_open("foo.nc", nf90_write, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Write data with prefilling behavior
...
status = nf90_set_fill(ncid, nf90_nofill, oldMode)
if (status /= nf90_noerr) call handle_err(status)
...
!  Write data with no prefilling
...
```

3 Groups

NetCDF-4 added support for hierarchical groups within netCDF datasets.

Groups are identified with a ncid, which identifies both the open file, and the group within that file. When a file is opened with NF90_OPEN or NF90_CREATE, the ncid for the root group of that file is provided. Using that as a starting point, users can add new groups, or list and navigate existing groups.

All netCDF calls take a ncid which determines where the call will take its action. For example, the NF90_DEF_VAR function takes a ncid as its first parameter. It will create a variable in whichever group its ncid refers to. Use the root ncid provided by NF90_CREATE or NF90_OPEN to create a variable in the root group. Or use NF90_DEF_GRP to create a group and use its ncid to define a variable in the new group.

Variable are only visible in the group in which they are defined. The same applies to attributes. "Global" attributes are defined in whichever group is refered to by the ncid.

Dimensions are visible in their groups, and all child groups.

Group operations are only permitted on netCDF-4 files - that is, files created with the HDF5 flag in nf90_create. (see Section 2.5 [NF90_CREATE], page 9). Groups are not compatible with the netCDF classic data model, so files created with the NF90_CLASSIC_MODEL file cannot contain groups (except the root group).

3.1 Find a Group ID: NF90_INQ_NCID

Given an ncid and group name (NULL or "" gets root group), return ncid of the named group.

Usage

```
function nf90_inq_ncid(ncid, name, grp_ncid)
  integer, intent(in) :: ncid
  character (len = *), intent(in) :: name
  integer, intent(out) :: grp_ncid
  integer :: nf90_inq_ncid
```

`NCID` The group id for this operation.

`NAME` A character array that holds the name of the desired group. Must be less then NF90_MAX_NAME.

`GRPID` The ID of the group will go here.

Errors

`NF90_NOERR`
No error.

`NF90_EBADID`
Bad group id.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

This example is from nf90_test/ftst_groups.F.

3.2 Get a List of Groups in a Group: NF90_INQ_GRPS

Given a location id, return the number of groups it contains, and an array of their ncids.

Usage

```
function nf90_inq_grps(ncid, numgrps, ncids)
  integer, intent(in) :: ncid
  integer, intent(out) :: numgrps
  integer, intent(out) :: ncids
  integer :: nf90_inq_grps
```

`NCID` The group id for this operation.

`NUMGRPS` An integer which will get number of groups in this group.

`NCIDS` An array of ints which will receive the IDs of all the groups in this group.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

3.3 Find all the Variables in a Group: NF90_INQ_VARIDS

Find all varids for a location.

Usage

```
function nf90_inq_varids(ncid, nvars, varids)
  integer, intent(in) :: ncid
  integer, intent(out) :: nvars
  integer, intent(out) :: varids
  integer :: nf90_inq_varids
```

`NCID` The group id for this operation.

`VARIDS` An already allocated array to store the list of varids. Use nf90_inq_nvars to find out how many variables there are. (see Section 6.6 [NF90_INQUIRE_VARIABLE], page 70).

Errors

`NF90_NOERR`
No error.

`NF90_EBADID`
Bad group id.

`NF90_ENOTNC4`
Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

3.4 Find all Dimensions Visible in a Group: NF90_INQ_DIMIDS

Find all dimids for a location. This finds all dimensions in a group, or any of its parents.

Usage

```
function nf90_inq_dimids(ncid, ndims, dimids, include_parents)
  integer, intent(in) :: ncid
```

```
    integer, intent(out) :: ndims
    integer, intent(out) :: dimids
    integer, intent(out) :: include_parents
    integer :: nf90_inq_dimids
```

`NCID` The group id for this operation.

`DIMIDS` An array of ints when the dimids of the visible dimensions will be stashed. Use nf90_inq_ndims to find out how many dims are visible from this group. (see Section 6.6 [NF90_INQUIRE_VARIABLE], page 70).

`INCLUDE_PARENTS`
If zero, only the group specified by NCID will be searched for dimensions. Otherwise parent groups will be searched too.

Errors

`NF90_NOERR`
No error.

`NF90_EBADID`
Bad group id.

`NF90_ENOTNC4`
Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

3.5 Find the Length of a Group's Full Name: NF90_INQ_GRPNAME_LEN

Given ncid, find length of the full name. (Root group is named "/", with length 1.)

Usage

```
  function nf90_inq_grpname_len(ncid, len)
    integer, intent(in) :: ncid
    integer, intent(out) :: len
    integer :: nf90_inq_grpname_len
  end function nf90_inq_grpname_len
```

`NCID` The group id for this operation.

`LEN` An integer where the length will be placed.

Errors

NF90_NOERR
: No error.

NF90_EBADID
: Bad group id.

NF90_ENOTNC4
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

NF90_ESTRICTNC3
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

NF90_EHDFERR
: An error was reported by the HDF5 layer.

Example

3.6 Find a Group's Name: NF90_INQ_GRPNAME

Given ncid, find relative name of group. (Root group is named "/").

The name provided by this function is relative to the parent group. For a full path name for the group is, with all parent groups included, separated with a forward slash (as in Unix directory names) See Section 3.7 [NF90_INQ_GRPNAME_FULL], page 28.

Usage

```
function nf90_inq_grpname(ncid, name)
  integer, intent(in) :: ncid
  character (len = *), intent(out) :: name
  integer :: nf90_inq_grpname
```

NCID
: The group id for this operation.

NAME
: The name of the group will be copied to this character array. The name will be less than NF90_MAX_NAME in length.

Errors

NF90_NOERR
: No error.

NF90_EBADID
: Bad group id.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

3.7 Find a Group's Full Name: NF90_INQ_GRPNAME_FULL

Given ncid, find complete name of group. (Root group is named "/").

The name provided by this function is a full path name for the group is, with all parent groups included, separated with a forward slash (as in Unix directory names). For a name relative to the parent group See Section 3.6 [NF90_INQ_GRPNAME], page 27.

To find the length of the full name See Section 3.5 [NF90_INQ_GRPNAME_LEN], page 26.

Usage

```
function nf90_inq_grpname_full(ncid, len, name)
  integer, intent(in) :: ncid
  integer, intent(out) :: len
  character (len = *), intent(out) :: name
  integer :: nf90_inq_grpname_full
```

`NCID` The group id for this operation.

`LEN` The length of the full group name will go here.

`NAME` The name of the group will be copied to this character array.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

This example is from test program nf_test/f90tst_grps.f90.

```
call check(nf90_inq_grpname_full(grpid1, len, name_in))
if (name_in .ne. grp1_full_name) stop 62
```

3.8 Find a Group's Parent: NF90_INQ_GRP_PARENT

Given ncid, find the ncid of the parent group.

When used with the root group, this function returns the NF90_ENOGRP error (since the root group has no parent.)

Usage

```
function nf90_inq_grp_parent(ncid, parent_ncid)
  integer, intent(in) :: ncid
  integer, intent(out) :: parent_ncid
  integer :: nf90_inq_grp_parent
```

`NCID` The group id.

`PARENT_NCID`
: The ncid of the parent group will be copied here.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_ENOGRP`
: No parent group found (i.e. this is the root group).

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

3.9 Find a Group by Name: NF90_INQ_GRP_NCID

Given a group name an an ncid, find the ncid of the group id.

Usage

```
function nf90_inq_grp_ncid(ncid, name, grpid)
  integer, intent(in) :: ncid
  character (len = *), intent(in) :: name
  integer, intent(out) :: grpid
  integer :: nf90_inq_grp_ncid

  nf90_inq_grp_ncid = nf_inq_grp_ncid(ncid, name, grpid)
end function nf90_inq_grp_ncid
```

`NCID` The group id to look in.

`GRP_NAME` The name of the group that should be found.

`GRP_NCID` This will get the group id, if it is found.

Return Codes

The following return codes may be returned by this function.

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_EINVAL`
: No name provided or name longer than NF90_MAX_NAME.

`NF90_ENOGRP`
: Named group not found.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

This example is from test program nf_test/f90tst_grps.f90.

```
! Get the group ids for the newly reopened file.
call check(nf90_inq_grp_ncid(ncid, GRP1_NAME, grpid1))
call check(nf90_inq_grp_ncid(grpid1, GRP2_NAME, grpid2))
call check(nf90_inq_grp_ncid(grpid2, GRP3_NAME, grpid3))
call check(nf90_inq_grp_ncid(grpid3, GRP4_NAME, grpid4))
```

3.10 Find a Group by its Fully-qualified Name: NF90_INQ_GRP_FULL_NCID

Given a fully qualified group name an an ncid, find the ncid of the group id.

Usage

```
function nf90_inq_grpname_full(ncid, len, name)
  integer, intent(in) :: ncid
  integer, intent(out) :: len
  character (len = *), intent(out) :: name
  integer :: nf90_inq_grpname_full

  nf90_inq_grpname_full = nf_inq_grpname_full(ncid, len, name)
end function nf90_inq_grpname_full
```

`NCID` The group id to look in.

`FULL_NAME`
The fully-qualified group name.

`GRP_NCID` This will get the group id, if it is found.

Return Codes

The following return codes may be returned by this function.

`NF90_NOERR`
No error.

`NF90_EBADID`
Bad group id.

`NF90_EINVAL`
No name provided or name longer than NF90_MAX_NAME.

`NF90_ENOGRP`
Named group not found.

`NF90_ENOTNC4`
Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

This example is from test program nf_test/tstf90_grps.f90.

```
! Check for the groups with full group names.
write(grp1_full_name, '(AA)') '/', GRP1_NAME
call check(nf90_inq_grp_full_ncid(ncid, grp1_full_name, grpid1))
```

3.11 Create a New Group: NF90_DEF_GRP

Create a group. Its location id is returned in new_ncid.

Usage

```
function nf90_def_grp(parent_ncid, name, new_ncid)
  integer, intent(in) :: parent_ncid
  character (len = *), intent(in) :: name
  integer, intent(out) :: new_ncid
  integer :: nf90_def_grp
```

`PARENT_NCID`
: The group id of the parent group.

`NAME`
: The name of the new group, which must be different from the name of any variable within the same parent group.

`NEW_NCID`
: The ncid of the new group will be placed there.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_ENAMEINUSE`
: That name is in use. Group names must be unique within a group.

`NF90_EMAXNAME`
: Name exceed max length NF90_MAX_NAME.

`NF90_EBADNAME`
: Name contains illegal characters.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

`NF90_EPERM`
: Attempt to write to a read-only file.

`NF90_ENOTINDEFINE`
: Not in define mode.

Example

```
C     Create the netCDF file.
      retval = nf90_create(file_name, NF90_NETCDF4, ncid)
      if (retval .ne. nf90_noerr) call handle_err(retval)

C     Create a group and a subgroup.
      retval = nf90_def_grp(ncid, group_name, grpid)
      if (retval .ne. nf90_noerr) call handle_err(retval)
      retval = nf90_def_grp(grpid, sub_group_name, sub_grpid)
      if (retval .ne. nf90_noerr) call handle_err(retval)
```

4 Dimensions

4.1 Dimensions Introduction

Dimensions for a netCDF dataset are defined when it is created, while the netCDF dataset is in define mode. Additional dimensions may be added later by reentering define mode. A netCDF dimension has a name and a length. At most one dimension in a netCDF dataset can have the unlimited length, which means variables using this dimension can grow along this dimension.

There is a suggested limit (512) to the number of dimensions that can be defined in a single netCDF dataset. The limit is the value of the constant NF90_MAX_DIMS. The purpose of the limit is to make writing generic applications simpler. They need only provide an array of NF90_MAX_DIMS dimensions to handle any netCDF dataset. The implementation of the netCDF library does not enforce this advisory maximum, so it is possible to use more dimensions, if necessary, but netCDF utilities that assume the advisory maximums may not be able to handle the resulting netCDF datasets.

Ordinarily, the name and length of a dimension are fixed when the dimension is first defined. The name may be changed later, but the length of a dimension (other than the unlimited dimension) cannot be changed without copying all the data to a new netCDF dataset with a redefined dimension length.

A netCDF dimension in an open netCDF dataset is referred to by a small integer called a dimension ID. In the Fortran 90 interface, dimension IDs are 1, 2, 3, ..., in the order in which the dimensions were defined.

Operations supported on dimensions are:

- Create a dimension, given its name and length.
- Get a dimension ID from its name.
- Get a dimension's name and length from its ID.
- Rename a dimension.

4.2 NF90_DEF_DIM

The function NF90_DEF_DIM adds a new dimension to an open netCDF dataset in define mode. It returns (as an argument) a dimension ID, given the netCDF ID, the dimension name, and the dimension length. At most one unlimited length dimension, called the record dimension, may be defined for each netCDF dataset.

Usage

```
function nf90_def_dim(ncid, name, len, dimid)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent( in) :: len
  integer,             intent(out) :: dimid
  integer                          :: nf90_def_dim
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`name` Dimension name.

`len` Length of dimension; that is, number of values for this dimension as an index to variables that use it. This should be either a positive integer or the predefined constant NF90_UNLIMITED.

`dimid` Returned dimension ID.

Errors

NF90_DEF_DIM returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The netCDF dataset is not in definition mode.
- The specified dimension name is the name of another existing dimension.
- The specified length is not greater than zero.
- The specified length is unlimited, but there is already an unlimited length dimension defined for this netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_DEF_DIM to create a dimension named lat of length 18 and a unlimited dimension named rec in a new netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, LatDimID, RecordDimID
...
status = nf90_create("foo.nc", nf90_noclobber, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_def_dim(ncid, "Lat", 18, LatDimID)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_def_dim(ncid, "Record", nf90_unlimited, RecordDimID)
if (status /= nf90_noerr) call handle_err(status)
```

4.3 NF90_INQ_DIMID

The function NF90_INQ_DIMID returns (as an argument) the ID of a netCDF dimension, given the name of the dimension. If ndims is the number of dimensions defined for a netCDF dataset, each dimension has an ID between 1 and ndims.

Usage

```
function nf90_inq_dimid(ncid, name, dimid)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent(out) :: dimid
  integer                          :: nf90_inq_dimid
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`name` Dimension name.

`dimid` Returned dimension ID.

Errors

NF90_INQ_DIMID returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The name that was specified is not the name of a dimension in the netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_INQ_DIMID to determine the dimension ID of a dimension named lat, assumed to have been defined previously in an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, LatDimID
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_inq_dimid(ncid, "Lat", LatDimID)
if (status /= nf90_noerr) call handle_err(status)
```

4.4 NF90_INQUIRE_DIMENSION

This function information about a netCDF dimension. Information about a dimension includes its name and its length. The length for the unlimited dimension, if any, is the number of records written so far.

Usage

```
function nf90_inquire_dimension(ncid, dimid, name, len)
  integer,                       intent( in) :: ncid, dimid
  character (len = *), optional, intent(out) :: name
  integer,             optional, intent(out) :: len
  integer                                    :: nf90_inquire_dimension
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`dimid` Dimension ID, from a previous call to NF90_INQ_DIMID or NF90_DEF_DIM.

`name` Returned dimension name. The caller must allocate space for the returned name. The maximum possible length, in characters, of a dimension name is given by the predefined constant NF90_MAX_NAME.

`len` Returned length of dimension. For the unlimited dimension, this is the current maximum value used for writing any variables with this dimension, that is the maximum record number.

Errors

These functions return the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The dimension ID is invalid for the specified netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_INQ_DIM to determine the length of a dimension named lat, and the name and current maximum length of the unlimited dimension for an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, LatDimID, RecordDimID
integer :: nLats, nRecords
character(len = nf90_max_name) :: RecordDimName
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
! Get ID of unlimited dimension
status = nf90_inquire(ncid, unlimitedDimId = RecordDimID)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_inq_dimid(ncid, "Lat", LatDimID)
if (status /= nf90_noerr) call handle_err(status)
! How many values of "lat" are there?
status = nf90_inquire_dimension(ncid, LatDimID, len = nLats)
if (status /= nf90_noerr) call handle_err(status)
! What is the name of the unlimited dimension, how many records are there?
status = nf90_inquire_dimension(ncid, RecordDimID, &
                                name = RecordDimName, len = Records)
if (status /= nf90_noerr) call handle_err(status)
```

4.5 NF90_RENAME_DIM

The function NF90_RENAME_DIM renames an existing dimension in a netCDF dataset open for writing. If the new name is longer than the old name, the netCDF dataset must be in define mode. You cannot rename a dimension to have the same name as another dimension.

Usage

```
function nf90_rename_dim(ncid, dimid, name)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent( in) :: dimid
  integer                          :: nf90_rename_dim
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`dimid` Dimension ID, from a previous call to NF90_INQ_DIMID or NF90_DEF_DIM.

`name` New dimension name.

Errors

NF90_RENAME_DIM returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The new name is the name of another dimension.
- The dimension ID is invalid for the specified netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.
- The new name is longer than the old name and the netCDF dataset is not in define mode.

Example

Here is an example using NF90_RENAME_DIM to rename the dimension lat to latitude in an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, LatDimID
...
status = nf90_open("foo.nc", nf90_write, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Put in define mode so we can rename the dimension
status = nf90_redef(ncid)
if (status /= nf90_noerr) call handle_err(status)
! Get the dimension ID for "Lat"...
status = nf90_inq_dimid(ncid, "Lat", LatDimID)
if (status /= nf90_noerr) call handle_err(status)
! ... and change the name to "Latitude".
status = nf90_rename_dim(ncid, LatDimID, "Latitude")
if (status /= nf90_noerr) call handle_err(status)
! Leave define mode
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

5 User Defined Data Types

5.1 User Defined Types Introduction

NetCDF-4 has added support for four different user defined data types.

`compound type`
: Like a C struct, a compound type is a collection of types, including other user defined types, in one package.

`variable length array type`
: The variable length array may be used to store ragged arrays.

`opaque type`
: This type has only a size per element, and no other type information.

`enum type`
: Like an enumeration in C, this type lets you assign text values to integer values, and store the integer values.

Users may construct user defined type with the various NF90_DEF_* functions described in this section. They may learn about user defined types by using the NF90_INQ_ functions defined in this section.

Once types are constructed, define variables of the new type with NF90_DEF_VAR (see Section 6.3 [NF90_DEF_VAR], page 64). Write to them with NF90_PUT_VAR (see Section 6.8 [NF90_PUT_VAR], page 72). Read data of user-defined type with NF90_GET_VAR (see Section 6.9 [NF90_GET_VAR], page 77).

Create attributes of the new type with NF90_PUT_ATT (see Section 7.2 [NF90_PUT_ATT], page 89). Read attributes of the new type with NF90_GET_ATT (see Section 7.4 [NF90_GET_ATT], page 93).

5.2 Learn the IDs of All Types in Group: NF90_INQ_TYPEIDS

Learn the number of types defined in a group, and their IDs.

Usage

```
function nf90_inq_typeids(ncid, ntypes, typeids)
  integer, intent(in) :: ncid
  integer, intent(out) :: ntypes
  integer, intent(out) :: typeids
  integer :: nf90_inq_typeids
```

`NCID`
: The group id.

`NTYPES`
: A pointer to int which will get the number of types defined in the group. If NULL, ignored.

`TYPEIDS`
: A pointer to an int array which will get the typeids. If NULL, ignored.

Errors

`NF90_NOERR`
: No error.

`NF90_BADID`
: Bad ncid.

Example

5.3 Find a Typeid from Group and Name: nf90_inq_typeid

Given a group ID and a type name, find the ID of the type. If the type is not found in the group, then the parents are searched. If still not found, the entire file is searched.

Usage

```
int nf90_inq_typeid(int ncid, char *name, nf90_type *typeidp);
```

`ncid` The group id.

`name` The name of a type.

`typeidp` The typeid, if found.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad ncid.

`NF90_EBADTYPE`
: Can't find type.

Example

5.4 Learn About a User Defined Type: NF90_INQ_TYPE

Given an ncid and a typeid, get the information about a type. This function will work on any type, including atomic and any user defined type, whether compound, opaque, enumeration, or variable length array.

For even more information about a user defined type Section 5.5 [NF90_INQ_USER_TYPE], page 43.

Usage

```
function nf90_inq_type(ncid, xtype, name, size, nfields)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(out) :: name
```

```
integer, intent(out) :: size
integer, intent(out) :: nfields
integer :: nf90_inq_type
```

`NCID` The ncid for the group containing the type (ignored for atomic types).

`XTYPE` The typeid for this type, as returned by NF90_DEF_COMPOUND, NF90_DEF_OPAQUE, NF90_DEF_ENUM, NF90_DEF_VLEN, or NF90_INQ_VAR, or as found in netcdf.inc in the list of atomic types (NF90_CHAR, NF90_INT, etc.).

`NAME` The name of the user defined type will be copied here. It will be NF90_MAX_NAME bytes or less. For atomic types, the type name from CDL will be given.

`SIZEP` The (in-memory) size of the type (in bytes) will be copied here. VLEN type size is the size of one element of the VLEN. String size is returned as the size of one char.

Return Codes

`NF90_NOERR`
No error.

`NF90_EBADTYPEID`
Bad typeid.

`NF90_ENOTNC4`
Seeking a user-defined type in a netCDF-3 file.

`NF90_ESTRICTNC3`
Seeking a user-defined type in a netCDF-4 file for which classic model has been turned on.

`NF90_EBADGRPID`
Bad group ID in ncid.

`NF90_EBADID`
Type ID not found.

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

5.5 Learn About a User Defined Type: NF90_INQ_USER_TYPE

Given an ncid and a typeid, get the information about a user defined type. This function will work on any user defined type, whether compound, opaque, enumeration, or variable length array.

Usage

```
function nf90_inq_user_type(ncid, xtype, name, size, base_typeid, nfields, class)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(out) :: name
  integer, intent(out) :: size
  integer, intent(out) :: base_typeid
  integer, intent(out) :: nfields
  integer, intent(out) :: class
  integer :: nf90_inq_user_type
```

`NCID` The ncid for the group containing the user defined type.

`XTYPE` The typeid for this type, as returned by NF90_DEF_COMPOUND, NF90_DEF_OPAQUE, NF90_DEF_ENUM, NF90_DEF_VLEN, or NF90_INQ_VAR.

`NAME` The name of the user defined type will be copied here. It will be NF90_MAX_NAME bytes or less.

`SIZE` The (in-memory) size of the user defined type will be copied here.

`BASE_NF90_TYPE`
The base typeid will be copied here for vlen and enum types.

`NFIELDS` The number of fields will be copied here for enum and compound types.

`CLASS` The class of the user defined type, NF90_VLEN, NF90_OPAQUE, NF90_ENUM, or NF90_COMPOUND, will be copied here.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPEID`
Bad typeid.

`NF90_EBADFIELDID`
Bad fieldid.

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

5.5.1 Set a Variable Length Array with NF90_PUT_VLEN_ELEMENT

Use this to set the element of the (potentially) n-dimensional array of VLEN. That is, this sets the data in one variable length array.

Usage

```
INTEGER FUNCTION NF90_PUT_VLEN_ELEMENT(INTEGER NCID, INTEGER XTYPE,
        CHARACTER*(*) VLEN_ELEMENT, INTEGER LEN, DATA)
```

`NCID` The ncid of the file that contains the VLEN type.

`XTYPE` The type of the VLEN.

`VLEN_ELEMENT`
The VLEN element to be set.

`LEN` The number of entries in this array.

`DATA` The data to be stored. Must match the base type of this VLEN.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPE`
Can't find the typeid.

`NF90_EBADID`
ncid invalid.

`NF90_EBADGRPID`
Group ID part of ncid was invalid.

Example

This example is from nf90_test/ftst_vars4.F.

```
C     Set up the vlen with this helper function, since F77 can't deal
C     with pointers.
      retval = nf90_put_vlen_element(ncid, vlen_typeid, vlen,
     &     vlen_len, data1)
      if (retval .ne. nf90_noerr) call handle_err(retval)
```

5.5.2 Set a Variable Length Array with NF90_GET_VLEN_ELEMENT

Use this to set the element of the (potentially) n-dimensional array of VLEN. That is, this sets the data in one variable length array.

Usage

```
INTEGER FUNCTION NF90_GET_VLEN_ELEMENT(INTEGER NCID, INTEGER XTYPE,
        CHARACTER*(*) VLEN_ELEMENT, INTEGER LEN, DATA)
```

`NCID` The ncid of the file that contains the VLEN type.

`XTYPE` The type of the VLEN.

`VLEN_ELEMENT`
The VLEN element to be set.

`LEN` This will be set to the number of entries in this array.

`DATA` The data will be copied here. Sufficient storage must be available or bad things will happen to you.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPE`
Can't find the typeid.

`NF90_EBADID`
ncid invalid.

`NF90_EBADGRPID`
Group ID part of ncid was invalid.

Example

5.6 Compound Types Introduction

NetCDF-4 added support for compound types, which allow users to construct a new type - a combination of other types, like a C struct.

Compound types are not supported in classic or 64-bit offset format files.

To write data in a compound type, first use nf90_def_compound to create the type, multiple calls to nf90_insert_compound to add to the compound type, and then write data with the appropriate nf90_put_var1, nf90_put_vara, nf90_put_vars, or nf90_put_varm call.

To read data written in a compound type, you must know its structure. Use the NF90_INQ_COMPOUND functions to learn about the compound type.

In Fortran a character buffer must be used for the compound data. The user must read the data from within that buffer in the same way that the C compiler which compiled netCDF would store the structure.

The use of compound types introduces challenges and portability issues for Fortran users.

5.6.1 Creating a Compound Type: NF90_DEF_COMPOUND

Create a compound type. Provide an ncid, a name, and a total size (in bytes) of one element of the completed compound type.

After calling this function, fill out the type with repeated calls to NF90_INSERT_COMPOUND (see Section 5.6.2 [NF90_INSERT_COMPOUND], page 48). Call NF90_INSERT_COMPOUND once for each field you wish to insert into the compound type.

Note that there does not seem to be a fully portable way to read such types into structures in Fortran 90 (and there are no structures in Fortran 77). Dozens of top-notch programmers are swarming over this problem in a sub-basement of Unidata's giant underground bunker in Wyoming.

Fortran users may use character buffers to read and write compound types. User are invited to try classic Fortran features such as the equivilence and the common block statment.

Usage

```
function nf90_def_compound(ncid, size, name, typeid)
  integer, intent(in) :: ncid
  integer, intent(in) :: size
  character (len = *), intent(in) :: name
  integer, intent(out) :: typeid
  integer :: nf90_def_compound
```

`NCID` The groupid where this compound type will be created.

`SIZE` The size, in bytes, of the compound type.

`NAME` The name of the new compound type.

`TYPEIDP` The typeid of the new type will be placed here.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_ENAMEINUSE`
: That name is in use. Compound type names must be unique in the data file.

`NF90_EMAXNAME`
: Name exceeds max length NF90_MAX_NAME.

`NF90_EBADNAME`
: Name contains illegal characters.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NF90_NETCDF4. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

`NF90_EPERM`
: Attempt to write to a read-only file.

`NF90_ENOTINDEFINE`
: Not in define mode.

Example

5.6.2 Inserting a Field into a Compound Type: NF90_INSERT_COMPOUND

Insert a named field into a compound type.

Usage

```
function nf90_insert_compound(ncid, xtype, name, offset, field_typeid)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(in) :: name
  integer, intent(in) :: offset
  integer, intent(in) :: field_typeid
  integer :: nf90_insert_compound
```

`TYPEID` The typeid for this compound type, as returned by NF90_DEF_COMPOUND, or NF90_INQ_VAR.

`NAME` The name of the new field.

`OFFSET` Offset in byte from the beginning of the compound type for this field.

`FIELD_TYPEID`
The type of the field to be inserted.

Errors

`NF90_NOERR`
No error.

`NF90_EBADID`
Bad group id.

`NF90_ENAMEINUSE`
That name is in use. Field names must be unique within a compound type.

`NF90_EMAXNAME`
Name exceed max length NF90_MAX_NAME.

`NF90_EBADNAME`
Name contains illegal characters.

`NF90_ENOTNC4`
Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NF90_NETCDF4. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

`NF90_ENOTINDEFINE`
Not in define mode.

Example

5.6.3 Inserting an Array Field into a Compound Type: NF90_INSERT_ARRAY_COMPOUND

Insert a named array field into a compound type.

Usage

```
function nf90_insert_array_compound(ncid, xtype, name, offset, field_typeid, &
     ndims, dim_sizes)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(in) :: name
  integer, intent(in) :: offset
  integer, intent(in) :: field_typeid
  integer, intent(in) :: ndims
  integer, intent(in) :: dim_sizes
  integer :: nf90_insert_array_compound
```

`NCID` The ID of the file that contains the array type and the compound type.

`XTYPE` The typeid for this compound type, as returned by nf90_def_compound, or nf90_inq_var.

`NAME` The name of the new field.

`OFFSET` Offset in byte from the beginning of the compound type for this field.

`FIELD_TYPEID`
The base type of the array to be inserted.

`NDIMS` The number of dimensions for the array to be inserted.

`DIM_SIZES`
An array containing the sizes of each dimension.

Errors

`NF90_NOERR`
No error.

`NF90_EBADID`
Bad group id.

`NF90_ENAMEINUSE`
That name is in use. Field names must be unique within a compound type.

`NF90_EMAXNAME`
Name exceed max length NF90_MAX_NAME.

`NF90_EBADNAME`
: Name contains illegal characters.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NF90_NETCDF4. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

`NF90_ENOTINDEFINE`
: Not in define mode.

`NF90_ETYPEDEFINED`
: Attempt to change type that has already been committed. The first time the file leaves define mode, all defined types are committed, and can't be changed. If you wish to add an array to a compound type, you must do so before the compound type is committed.

Example

5.6.4 Learn About a Compound Type: NF90_INQ_COMPOUND

Get the number of fields, length in bytes, and name of a compound type.

In addtion to the NF90_INQ_COMPOUND function, three additional functions are provided which get only the name, size, and number of fields.

Usage

```
function nf90_inq_compound(ncid, xtype, name, size, nfields)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(out) :: name
  integer, intent(out) :: size
  integer, intent(out) :: nfields
  integer :: nf90_inq_compound

function nf90_inq_compound_name(ncid, xtype, name)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(out) :: name
  integer :: nf90_inq_compound_name

function nf90_inq_compound_size(ncid, xtype, size)
  integer, intent(in) :: ncid
```

```
    integer, intent(in) :: xtype
    integer, intent(out) :: size
    integer :: nf90_inq_compound_size

  function nf90_inq_compound_nfields(ncid, xtype, nfields)
    integer, intent(in) :: ncid
    integer, intent(in) :: xtype
    integer, intent(out) :: nfields
    integer :: nf90_inq_compound_nfields
```

`NCID` The ID of any group in the file that contains the compound type.

`XTYPE` The typeid for this compound type, as returned by NF90_DEF_COMPOUND, or NF90_INQ_VAR.

`NAME` Character array which will get the name of the compound type. It will have a maximum length of NF90_MAX_NAME.

`SIZEP` The size of the compound type in bytes will be put here.

`NFIELDSP` The number of fields in the compound type will be placed here.

Return Codes

`NF90_NOERR`
No error.

`NF90_EBADID`
Couldn't find this ncid.

`NF90_ENOTNC4`
Not a netCDF-4/HDF5 file.

`NF90_ESTRICTNC3`
A netCDF-4/HDF5 file, but with CLASSIC_MODEL. No user defined types are allowed in the classic model.

`NF90_EBADTYPE`
This type not a compound type.

`NF90_EBADTYPEID`
Bad type id.

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

5.6.5 Learn About a Field of a Compound Type: NF90_INQ_COMPOUND_FIELD

Get information about one of the fields of a compound type.

Usage

```
function nf90_inq_compound_field(ncid, xtype, fieldid, name, offset, &
     field_typeid, ndims, dim_sizes)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: fieldid
  character (len = *), intent(out) :: name
  integer, intent(out) :: offset
  integer, intent(out) :: field_typeid
  integer, intent(out) :: ndims
  integer, intent(out) :: dim_sizes
  integer :: nf90_inq_compound_field

function nf90_inq_compound_fieldname(ncid, xtype, fieldid, name)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: fieldid
  character (len = *), intent(out) :: name
  integer :: nf90_inq_compound_fieldname

function nf90_inq_compound_fieldindex(ncid, xtype, name, fieldid)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(in) :: name
  integer, intent(out) :: fieldid
  integer :: nf90_inq_compound_fieldindex

function nf90_inq_compound_fieldoffset(ncid, xtype, fieldid, offset)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: fieldid
  integer, intent(out) :: offset
  integer :: nf90_inq_compound_fieldoffset

function nf90_inq_compound_fieldtype(ncid, xtype, fieldid, field_typeid)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: fieldid
  integer, intent(out) :: field_typeid
  integer :: nf90_inq_compound_fieldtype

function nf90_inq_compound_fieldndims(ncid, xtype, fieldid, ndims)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: fieldid
  integer, intent(out) :: ndims
```

```
    integer :: nf90_inq_compound_fieldndims

  function nf90_inq_cmp_fielddim_sizes(ncid, xtype, fieldid, dim_sizes)
    integer, intent(in) :: ncid
    integer, intent(in) :: xtype
    integer, intent(in) :: fieldid
    integer, intent(out) :: dim_sizes
    integer :: nf90_inq_cmp_fielddim_sizes
```

`NCID` The groupid where this compound type exists.

`XTYPE` The typeid for this compound type, as returned by NF90_DEF_COMPOUND, or NF90_INQ_VAR.

`FIELDID` A one-based index number specifying a field in the compound type.

`NAME` A character array which will get the name of the field. The name will be NF90_MAX_NAME characters, at most.

`OFFSETP` An integer which will get the offset of the field.

`FIELD_TYPEID`
An integer which will get the typeid of the field.

`NDIMSP` An integer which will get the number of dimensions of the field.

`DIM_SIZESP`
An integer array which will get the dimension sizes of the field.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPEID`
Bad type id.

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

5.7 Variable Length Array Introduction

NetCDF-4 added support for a variable length array type. This is not supported in classic or 64-bit offset files, or in netCDF-4 files which were created with the NF90_CLASSIC_MODEL flag.

A variable length array is represented in C as a structure from HDF5, the nf90_vlen_t structure. It contains a len member, which contains the length of that array, and a pointer to the array.

So an array of VLEN in C is an array of nc_vlen_t structures. The only way to handle this in Fortran is with a character buffer sized correctly for the platform.

VLEN arrays are handled differently with respect to allocation of memory. Generally, when reading data, it is up to the user to malloc (and subsequently free) the memory needed to hold the data. It is up to the user to ensure that enough memory is allocated.

With VLENs, this is impossible. The user cannot know the size of an array of VLEN until after reading the array. Therefore when reading VLEN arrays, the netCDF library will allocate the memory for the data within each VLEN.

It is up to the user, however, to eventually free this memory. This is not just a matter of one call to free, with the pointer to the array of VLENs; each VLEN contains a pointer which must be freed.

Compression is permitted but may not be effective for VLEN data, because the compression is applied to the nc_vlen_t structures, rather than the actual data.

5.7.1 Define a Variable Length Array (VLEN): NF90_DEF_VLEN

Use this function to define a variable length array type.

Usage

```
function nf90_def_vlen(ncid, name, base_typeid, xtypeid)
  integer, intent(in) :: ncid
  character (len = *), intent(in) :: name
  integer, intent(in) :: base_typeid
  integer, intent(out) :: xtypeid
  integer :: nf90_def_vlen
```

`NCID` The ncid of the file to create the VLEN type in.

`NAME` A name for the VLEN type.

`BASE_TYPEID`
: The typeid of the base type of the VLEN. For example, for a VLEN of shorts, the base type is NF90_SHORT. This can be a user defined type.

`XTYPEP` The typeid of the new VLEN type will be set here.

Errors

`NF90_NOERR`
: No error.

`NF90_EMAXNAME`
: NF90_MAX_NAME exceeded.

`NF90_ENAMEINUSE`
: Name is already in use.

`NF90_EBADNAME`
: Attribute or variable name contains illegal characters.

`NF90_EBADID`
: ncid invalid.

`NF90_EBADGRPID`
: Group ID part of ncid was invalid.

`NF90_EINVAL`
: Size is invalid.

`NF90_ENOMEM`
: Out of memory.

Example

5.7.2 Learning about a Variable Length Array (VLEN) Type: NF90_INQ_VLEN

Use this type to learn about a vlen.

Usage

```
function nf90_inq_vlen(ncid, xtype, name, datum_size, base_nc_type)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(out) :: name
  integer, intent(out) :: datum_size
  integer, intent(out) :: base_nc_type
  integer :: nf90_inq_vlen
```

`NCID` The ncid of the file that contains the VLEN type.

`XTYPE` The type of the VLEN to inquire about.

`NAME` The name of the VLEN type. The name will be NF90_MAX_NAME characters or less.

`DATUM_SIZEP`
: A pointer to a size_t, this will get the size of one element of this vlen.

`BASE_NF90_TYPEP`
: An integer that will get the type of the VLEN base type. (In other words, what type is this a VLEN of?)

Errors

`NF90_NOERR`
: No error.

`NF90_EBADTYPE`
: Can't find the typeid.

`NF90_EBADID`
: ncid invalid.

`NF90_EBADGRPID`
: Group ID part of ncid was invalid.

Example

5.7.3 Releasing Memory for a Variable Length Array (VLEN) Type: NF90_FREE_VLEN

When a VLEN is read into user memory from the file, the HDF5 library performs memory allocations for each of the variable length arrays contained within the VLEN structure. This memory must be freed by the user to avoid memory leaks.

This violates the normal netCDF expectation that the user is responsible for all memory allocation. But, with VLEN arrays, the underlying HDF5 library allocates the memory for the user, and the user is responsible for deallocating that memory.

Usage

```
function nf90_free_vlen(vl)
  character (len = *), intent(in) :: vlen
  integer :: nf90_free_vlen
end function nf90_free_vlen
```

`VL` The variable length array structure which is to be freed.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPE`
Can't find the typeid.

Example

5.8 Opaque Type Introduction

NetCDF-4 added support for the opaque type. This is not supported in classic or 64-bit offset files.

The opaque type is a type which is a collection of objects of a known size. (And each object is the same size). Nothing is known to netCDF about the contents of these blobs of data, except their size in bytes, and the name of the type.

To use an opaque type, first define it with Section 5.8.1 [NF90_DEF_OPAQUE], page 56. If encountering an enum type in a new data file, use Section 5.8.2 [NF90_INQ_OPAQUE], page 57 to learn its name and size.

5.8.1 Creating Opaque Types: NF90_DEF_OPAQUE

Create an opaque type. Provide a size and a name.

Usage

```
function nf90_def_opaque(ncid, size, name, xtype)
  integer, intent(in) :: ncid
```

```
    integer, intent(in) :: size
    character (len = *), intent(in) :: name
    integer, intent(out) :: xtype
    integer :: nf90_def_opaque
```

`NCID` The groupid where the type will be created. The type may be used anywhere in the file, no matter what group it is in.

`NAME` The name for this type. Must be shorter than NF90_MAX_NAME.

`SIZE` The size of each opaque object.

`TYPEIDP` Pointer where the new typeid for this type is returned. Use this typeid when defining variables of this type with Section 6.3 [NF90_DEF_VAR], page 64.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPEID`
Bad typeid.

`NF90_EBADFIELDID`
Bad fieldid.

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

5.8.2 Learn About an Opaque Type: NF90_INQ_OPAQUE

Given a typeid, get the information about an opaque type.

Usage

```
  function nf90_inq_opaque(ncid, xtype, name, size)
    integer, intent(in) :: ncid
    integer, intent(in) :: xtype
    character (len = *), intent(out) :: name
    integer, intent(out) :: size
    integer :: nf90_inq_opaque
```

`NCID` The ncid for the group containing the opaque type.

`XTYPE` The typeid for this opaque type, as returned by NF90_DEF_COMPOUND, or NF90_INQ_VAR.

`NAME` The name of the opaque type will be copied here. It will be NF90_MAX_NAME bytes or less.

`SIZEP` The size of the opaque type will be copied here.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADTYPEID`
: Bad typeid.

`NF90_EBADFIELDID`
: Bad fieldid.

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

5.9 Enum Type Introduction

NetCDF-4 added support for the enum type. This is not supported in classic or 64-bit offset files.

5.9.1 Creating a Enum Type: NF90_DEF_ENUM

Create an enum type. Provide an ncid, a name, and a base integer type.

After calling this function, fill out the type with repeated calls to NF90_INSERT_ENUM (see Section 5.9.2 [NF90_INSERT_ENUM], page 59). Call NF90_INSERT_ENUM once for each value you wish to make part of the enumeration.

Usage

```
function nf90_def_enum(ncid, base_typeid, name, typeid)
  integer, intent(in) :: ncid
  integer, intent(in) :: base_typeid
  character (len = *), intent(in) :: name
  integer, intent(out) :: typeid
  integer :: nf90_def_enum
```

`NCID`
: The groupid where this compound type will be created.

`BASE_TYPEID`
: The base integer type for this enum. Must be one of: NF90_BYTE, NF90_UBYTE, NF90_SHORT, NF90_USHORT, NF90_INT, NF90_UINT, NF90_INT64, NF90_UINT64.

`NAME`
: The name of the new enum type.

`TYPEIDP`
: The typeid of the new type will be placed here.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
Bad group id.

`NF90_ENAMEINUSE`
That name is in use. Compound type names must be unique in the data file.

`NF90_EMAXNAME`
Name exceeds max length NF90_MAX_NAME.

`NF90_EBADNAME`
Name contains illegal characters.

`NF90_ENOTNC4`
Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NF90_NETCDF4. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

`NF90_EPERM`
Attempt to write to a read-only file.

`NF90_ENOTINDEFINE`
Not in define mode.

Example

5.9.2 Inserting a Field into a Enum Type: NF90_INSERT_ENUM

Insert a named member into a enum type.

Usage

```
function nf90_insert_enum(ncid, xtype, name, value)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(in) :: name
  integer, intent(in) :: value
  integer :: nf90_insert_enum
```

`NCID` The ncid of the group which contains the type.

`TYPEID` The typeid for this enum type, as returned by nf90_def_enum, or nf90_inq_var.

`IDENTIFIER`
The identifier of the new member.

`VALUE` The value that is to be associated with this member.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADID`
: Bad group id.

`NF90_ENAMEINUSE`
: That name is in use. Field names must be unique within a enum type.

`NF90_EMAXNAME`
: Name exceed max length NF90_MAX_NAME.

`NF90_EBADNAME`
: Name contains illegal characters.

`NF90_ENOTNC4`
: Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NF90_NETCDF4. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_ESTRICTNC3`
: This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see Section 2.6 [NF90_OPEN], page 11).

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

`NF90_ENOTINDEFINE`
: Not in define mode.

Example

5.9.3 Learn About a Enum Type: NF90_INQ_ENUM

Get information about a user-defined enumeration type.

Usage

```
function nf90_inq_enum(ncid, xtype, name, base_nc_type, base_size, num_members)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  character (len = *), intent(out) :: name
  integer, intent(out) :: base_nc_type
  integer, intent(out) :: base_size
  integer, intent(out) :: num_members
  integer :: nf90_inq_enum
```

`NCID` The group ID of the group which holds the enum type.

`XTYPE` The typeid for this enum type, as returned by NF90_DEF_ENUM, or NF90_INQ_VAR.

`NAME` Character array which will get the name. It will have a maximum length of NF90_MAX_NAME.

`BASE_NF90_TYPE`
An integer which will get the base integer type of this enum.

`BASE_SIZE`
An integer which will get the size (in bytes) of the base integer type of this enum.

`NUM_MEMBERS`
An integer which will get the number of members defined for this enumeration type.

Errors

`NF90_NOERR`
No error.

`NF90_EBADTYPEID`
Bad type id.

`NF90_EHDFERR`
An error was reported by the HDF5 layer.

Example

5.9.4 Learn the Name of a Enum Type: nf90_inq_enum_member

Get information about a member of an enum type.

Usage

```
function nf90_inq_enum_member(ncid, xtype, idx, name, value)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: idx
  character (len = *), intent(out) :: name
  integer, intent(in) :: value
  integer :: nf90_inq_enum_member
```

`NCID` The groupid where this enum type exists.

`XTYPE` The typeid for this enum type.

`IDX` The one-based index number for the member of interest.

`NAME` A character array which will get the name of the member. It will have a maximum length of NF90_MAX_NAME.

`VALUE` An integer that will get the value associated with this member.

Errors

`NF90_NOERR`
: No error.

`NF90_EBADTYPEID`
: Bad type id.

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

Example

5.9.5 Learn the Name of a Enum Type: NF90_INQ_ENUM_IDENT

Get the name which is associated with an enum member value.

This is similar to NF90_INQ_ENUM_MEMBER, but instead of using the index of the member, you use the value of the member.

Usage

```
function nf90_inq_enum_ident(ncid, xtype, value, idx)
  integer, intent(in) :: ncid
  integer, intent(in) :: xtype
  integer, intent(in) :: value
  integer, intent(out) :: idx
  integer :: nf90_inq_enum_ident
```

`NCID` The groupid where this enum type exists.

`XTYPE` The typeid for this enum type.

`VALUE` The value for which an identifier is sought.

`IDENTIFIER`
: A character array that will get the identifier. It will have a maximum length of NF90_MAX_NAME.

Return Code

`NF90_NOERR`
: No error.

`NF90_EBADTYPEID`
: Bad type id, or not an enum type.

`NF90_EHDFERR`
: An error was reported by the HDF5 layer.

`NF90_EINVAL`
: The value was not found in the enum.

Example

6 Variables

6.1 Variables Introduction

Variables for a netCDF dataset are defined when the dataset is created, while the netCDF dataset is in define mode. Other variables may be added later by reentering define mode. A netCDF variable has a name, a type, and a shape, which are specified when it is defined. A variable may also have values, which are established later in data mode.

Ordinarily, the name, type, and shape are fixed when the variable is first defined. The name may be changed, but the type and shape of a variable cannot be changed. However, a variable defined in terms of the unlimited dimension can grow without bound in that dimension.

A netCDF variable in an open netCDF dataset is referred to by a small integer called a variable ID.

Variable IDs reflect the order in which variables were defined within a netCDF dataset. Variable IDs are 1, 2, 3,..., in the order in which the variables were defined. A function is available for getting the variable ID from the variable name and vice-versa.

Attributes (see Chapter 7 [Attributes], page 89) may be associated with a variable to specify such properties as units.

Operations supported on variables are:

- Create a variable, given its name, data type, and shape.
- Get a variable ID from its name.
- Get a variable's name, data type, shape, and number of attributes from its ID.
- Put a data value into a variable, given variable ID, indices, and value.
- Put an array of values into a variable, given variable ID, corner indices, edge lengths, and a block of values.
- Put a subsampled or mapped array-section of values into a variable, given variable ID, corner indices, edge lengths, stride vector, index mapping vector, and a block of values.
- Get a data value from a variable, given variable ID and indices.
- Get an array of values from a variable, given variable ID, corner indices, and edge lengths.
- Get a subsampled or mapped array-section of values from a variable, given variable ID, corner indices, edge lengths, stride vector, and index mapping vector.
- Rename a variable.

6.2 Language Types Corresponding to netCDF external data types

The following table gives the netCDF external data types and the corresponding type constants for defining variables in the FORTRAN interface:

Type	FORTRAN API Mnemonic	Bits
byte	NF90_BYTE	8

char	NF90_CHAR	8
short	NF90_SHORT	16
int	NF90_INT	32
float	NF90_FLOAT	32
double	NF90_DOUBLE	64

The first column gives the netCDF external data type, which is the same as the CDL data type. The next column gives the corresponding Fortran 90 parameter for use in netCDF functions (the parameters are defined in the netCDF Fortran 90 module netcdf.f90). The last column gives the number of bits used in the external representation of values of the corresponding type.

Note that there are no netCDF types corresponding to 64-bit integers or to characters wider than 8 bits in the current version of the netCDF library.

6.3 Create a Variable: `NF90_DEF_VAR`

The function NF90_DEF_VAR adds a new variable to an open netCDF dataset in define mode. It returns (as an argument) a variable ID, given the netCDF ID, the variable name, the variable type, the number of dimensions, and a list of the dimension IDs.

Optional arguments allow additional settings for variables in netCDF-4/HDF5 files. These parameters allow data compression and control of the layout of the data on disk for performance tuning. These parameters may also be used to set the chunk sizes to get chunked storage, or to set the contiguous flag to get contiguous storage.

Variables that make use of one or more unlimited dimensions, compression, or checksums must use chunking. Such variables are created with default chunk sizes of 1 for each unlimited dimension and the dimension length for other dimensions, except that if the resulting chunks are too large, the default chunk sizes for non-record dimensions are reduced.

All parameters after the varid are optional, and only supported if netCDF was built with netCDF-4 features enabled, and if the variable is in a netCDF-4/HDF5 file.

Usage

```
function nf90_def_var(ncid, name, xtype, dimids, varid, contiguous, &
      chunksizes, deflate_level, shuffle, fletcher32, endianness, &
      cache_size, cache_nelems, cache_preemption)
  integer, intent(in) :: ncid
  character (len = *), intent(in) :: name
  integer, intent( in) :: xtype
  integer, dimension(:), intent(in) :: dimids
  integer, intent(out) :: varid
  logical, optional, intent(in) :: contiguous
  integer, optional, dimension(:), intent(in) :: chunksizes
  integer, optional, intent(in) :: deflate_level
  logical, optional, intent(in) :: shuffle, fletcher32
```

```
integer, optional, intent(in) :: endianness
 integer, optional, intent(in) :: cache_size, cache_nelems, cache_preemption
integer                                    :: nf90_def_var
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`name` Variable name.

`xtype` One of the set of predefined netCDF external data types. The type of this parameter, NF90_TYPE, is defined in the netCDF header file. The valid netCDF external data types are NF90_BYTE, NF90_CHAR, NF90_SHORT, NF90_INT, NF90_FLOAT, and NF90_DOUBLE. If the file is a NetCDF-4/HDF5 file, the additional types NF90_UBYTE, NF90_USHORT, NF90_UINT, NF90_INT64, NF90_UINT64, and NF90_STRING may be used, as well as a user defined type ID.

`dimids` Vector of dimension IDs corresponding to the variable dimensions. For example, a vector of 2 dimension IDs specifies a 2-dimensional matrix.

If an integer is passed for this parameter, a 1-D variable is created.

If this parameter is not passed (or is a 1D array of size zero) it means the variable is a scalar with no dimensions.

For classic data model files, if the ID of the unlimited dimension is included, it must be first. In expanded model netCDF4/HDF5 files, there may be any number of unlimited dimensions, and they may be used in any element of the dimids array.

This argument is optional, and if absent specifies a scalar with no dimensions.

`varid` Returned variable ID.

`storage` If NF90_CONTIGUOUS, then contiguous storage is used for this variable. Variables that use deflation, shuffle filter, or checksums, or that have one or more unlimited dimensions cannot use contiguous storage.

If NF90_CHUNKED, then chunked storage is used for this variable. Chunk sizes may be specified with the chunksizes parameter. Default sizes will be used if chunking is required and this function is not called.

By default contiguous storage is used for fix-sized variables when conpression, chunking, shuffle, and checksums are not used.

`chunksizes`

An array of chunk number of elements. This array has the number of elements along each dimension of the data chunk. The array must have the one chunksize for each dimension in the variable.

The total size of a chunk must be less than 4 GiB. That is, the product of all chunksizes and the size of the data (or the size of nc_vlen_t for VLEN types) must be less than 4 GiB. (This is a very large chunk size in any case.)

If not provided, but chunked data are needed, then default chunksizes will be chosen. For more information see Section “Chunking” in *The NetCDF Users Guide*.

`shuffle` If non-zero, turn on the shuffle filter.

`deflate_level`
: If the deflate parameter is non-zero, set the deflate level to this value. Must be between 1 and 9.

`fletcher32`
: Set to true to turn on fletcher32 checksums for this variable.

`endianness`
: Set to NF90_ENDIAN_LITTLE for little-endian format, NF90_ENDIAN_BIG for big-endian format, and NF90_ENDIAN_NATIVE (the default) for the native endianness of the platform.

`cache_size`
: The size of the per-variable cache in MegaBytes.

`cache_nelems`
: The number slots in the per-variable chunk cache (should be a prime number larger than the number of chunks in the cache).

`cache_preemption`
: The preemtion value must be between 0 and 100 inclusive and indicates how much chunks that have been fully read are favored for preemption. A value of zero means fully read chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of 100 means fully read chunks are always preempted before other chunks.

Return Codes

NF90_DEF_VAR returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error.

- NF90_EBADNAME The specified variable name is the name of another existing variable.
- NF90_EBADTYPE The specified type is not a valid netCDF type.
- NF90_EMAXDIMS The specified number of dimensions is negative or more than the constant NF90_MAX_VAR_DIMS, the maximum number of dimensions permitted for a netCDF variable. (Does not apply to netCDF-4/HDF5 files unless they were created with the CLASSIC_MODE flag.)
- NF90_EBADDIM One or more of the dimension IDs in the list of dimensions is not a valid dimension ID for the netCDF dataset.
- NF90_EMAXVARS The number of variables would exceed the constant NF90_MAX_VARS, the maximum number of variables permitted in a classic netCDF dataset. (Does not apply to netCDF-4/HDF5 files unless they were created with the CLASSIC_MODE flag.)
- NF90_BADID The specified netCDF ID does not refer to an open netCDF dataset.
- NF90_ENOTNC4 NetCDF-4 operation attempted on a files that is not a netCDF-4/HDF5 file. Only variables in NetCDF-4/HDF5 files may use compression, chunking, and endianness control.
- NF90_ENOTVAR Can't find this variable.

- NF90_EINVAL Invalid input. This may be because contiguous storage is requested for a variable that has compression, checksums, chunking, or one or more unlimited dimensions.
- NF90_ELATEDEF This variable has already been the subject of a NF90_ENDDEF call. Once enddef has been called, it is impossible to set the chunking for a variable. (In netCDF-4/HDF5 files NF90_ENDDEF will be called automatically for any data read or write.)
- NF90_ENOTINDEFINE Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with NF90_STRICT_NC3 flag. (see Section 2.5 [NF90_CREATE], page 9).
- NF90_ESTRICTNC3 Trying to create a var some place other than the root group in a netCDF file with NF90_STRICT_NC3 turned on.

Example

Here is an example using NF90_DEF_VAR to create a variable named rh of type double with three dimensions, time, lat, and lon in a new netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: status, ncid
integer :: LonDimId, LatDimId, TimeDimId
integer :: RhVarId
...
status = nf90_create("foo.nc", nf90_NoClobber, ncid)
if(status /= nf90_NoErr) call handle_error(status)
...
! Define the dimensions
status = nf90_def_dim(ncid, "lat", 5, LatDimId)
if(status /= nf90_NoErr) call handle_error(status)
status = nf90_def_dim(ncid, "lon", 10, LonDimId)
if(status /= nf90_NoErr) call handle_error(status)
status = nf90_def_dim(ncid, "time", nf90_unlimited, TimeDimId)
if(status /= nf90_NoErr) call handle_error(status)
...
! Define the variable
status = nf90_def_var(ncid, "rh", nf90_double, &
                      (/ LonDimId, LatDimID, TimeDimID /), RhVarId)
if(status /= nf90_NoErr) call handle_error(status)
```

In the following example, from nf_test/f90tst_vars2.f90, chunking, checksums, and endianness control are all used in a netCDF-4/HDF5 file.

```
! Create the netCDF file.
call check(nf90_create(FILE_NAME, nf90_netcdf4, ncid, cache_nelems = CACHE_NELEMS, &
     cache_size = CACHE_SIZE))

! Define the dimensions.
call check(nf90_def_dim(ncid, "x", NX, x_dimid))
```

```
call check(nf90_def_dim(ncid, "y", NY, y_dimid))
dimids =  (/ y_dimid, x_dimid /)

! Define some variables.
chunksizes = (/ NY, NX /)
call check(nf90_def_var(ncid, VAR1_NAME, NF90_INT, dimids, varid1, chunksizes = chunksizes, &
     shuffle = .TRUE., fletcher32 = .TRUE., endianness = nf90_endian_big, deflate_level = DEFLATE_LEVEL))
call check(nf90_def_var(ncid, VAR2_NAME, NF90_INT, dimids, varid2, contiguous = .TRUE.))
call check(nf90_def_var(ncid, VAR3_NAME, NF90_INT64, varid3))
call check(nf90_def_var(ncid, VAR4_NAME, NF90_INT, x_dimid, varid4, contiguous = .TRUE.))
```

6.4 Define Fill Parameters for a Variable: `nf90_def_var_fill`

The function NF90_DEF_VAR_FILL sets the fill parameters for a variable in a netCDF-4 file.

This function must be called after the variable is defined, but before NF90_ENDDEF is called.

Usage

```
NF90_DEF_VAR_FILL(INTEGER NCID, INTEGER VARID, INTEGER NO_FILL, FILL_VALUE);
```

`NCID` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`VARID` Variable ID.

`NO_FILL` Set to non-zero value to set no_fill mode on a variable. When this mode is on, fill values will not be written for the variable. This is helpful in high performance applications. For netCDF-4/HDF5 files (whether classic model or not), this may only be changed after the variable is defined, but before it is committed to disk (i.e. before the first NF90_ENDDEF after the NF90_DEF_VAR.) For classic and 64-bit offset file, the no_fill mode may be turned on and off at any time.

`FILL_VALUE`
A value which will be used as the fill value for the variable. Must be the same type as the variable. This will be written to a _FillValue attribute, created for this purpose. If NULL, this argument will be ignored.

Return Codes

`NF90_NOERR`
No error.

`NF90_BADID`
Bad ncid.

`NF90_ENOTNC4`
Not a netCDF-4 file.

`NF90_ENOTVAR`
Can't find this variable.

NF90_ELATEDEF
: This variable has already been the subject of a NF90_ENDDEF call. In netCDF-4 files NF90_ENDDEF will be called automatically for any data read or write. Once enddef has been called, it is impossible to set the fill for a variable.

NF90_ENOTINDEFINE
: Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with NF90_STRICT_NC3 flag. (see Section 2.5 [NF90_CREATE], page 9).

NF90_EPERM
: Attempt to create object in read-only file.

Example

6.5 Learn About Fill Parameters for a Variable: NF90_INQ_VAR_FILL

The function NF90_INQ_VAR_FILL returns the fill settings for a variable in a netCDF-4 file.

Usage

```
NF90_INQ_VAR_FILL(INTEGER NCID, INTEGER VARID, INTEGER NO_FILL, FILL_VALUE)
```

NCID
: NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

VARID
: Variable ID.

NO_FILL
: An integer which will get a 1 if no_fill mode is set for this variable, and a zero if it is not set

FILL_VALUE
: This will get the fill value for this variable. This parameter will be ignored if it is NULL.

Return Codes

NF90_NOERR
: No error.

NF90_BADID
: Bad ncid.

NF90_ENOTNC4
: Not a netCDF-4 file.

NF90_ENOTVAR
: Can't find this variable.

Example

6.6 Get Information about a Variable from Its ID: NF90_INQUIRE_VARIABLE

NF90_INQUIRE_VARIABLE returns information about a netCDF variable given its ID. Information about a variable includes its name, type, number of dimensions, a list of dimension IDs describing the shape of the variable, and the number of variable attributes that have been assigned to the variable.

All parameters after nAtts are optional, and only supported if netCDF was built with netCDF-4 features enabled, and if the variable is in a netCDF-4/HDF5 file.

Usage

```
function nf90_inquire_variable(ncid, varid, name, xtype, ndims, dimids, nAtts, &
     contiguous, chunksizes, deflate_level, shuffle, fletcher32, endianness)
  integer, intent(in) :: ncid, varid
  character (len = *), optional, intent(out) :: name
  integer, optional, intent(out) :: xtype, ndims
  integer, dimension(:), optional, intent(out) :: dimids
  integer, optional, intent(out) :: nAtts
  logical, optional, intent(out) :: contiguous
  integer, optional, dimension(:), intent(out) :: chunksizes
  integer, optional, intent(out) :: deflate_level
  logical, optional, intent(out) :: shuffle, fletcher32
  integer, optional, intent(out) :: endianness
  integer :: nf90_inquire_variable
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID.

`name` Returned variable name. The caller must allocate space for the returned name. The maximum possible length, in characters, of a variable name is given by the predefined constant NF90_MAX_NAME.

`xtype` Returned variable type, one of the set of predefined netCDF external data types. The type of this parameter, NF90_TYPE, is defined in the netCDF header file. The valid netCDF external data types are NF90_BYTE, NF90_CHAR, NF90_SHORT, NF90_INT, NF90_FLOAT, AND NF90_DOUBLE.

`ndims` Returned number of dimensions the variable was defined as using. For example, 2 indicates a matrix, 1 indicates a vector, and 0 means the variable is a scalar with no dimensions.

`dimids` Returned vector of *ndimsp dimension IDs corresponding to the variable dimensions. The caller must allocate enough space for a vector of at least *ndimsp integers to be returned. The maximum possible number of dimensions for a variable is given by the predefined constant NF90_MAX_VAR_DIMS.

`natts` Returned number of variable attributes assigned to this variable.

`contiguous`
On return, set to NF90_CONTIGUOUS if this variable uses contiguous storage, NF90_CHUNKED if it uses chunked storage.

`chunksizes`
: An array of chunk sizes. The array must have the one element for each dimension in the variable.

`shuffle`
: True if the shuffle filter is turned on for this variable.

`deflate_level`
: The deflate_level from 0 to 9. A value of zero indicates no deflation is in use.

`fletcher32`
: Set to true if the fletcher32 checksum filter is turned on for this variable.

`endianness`
: Will be set to NF90_ENDIAN_LITTLE if this variable is stored in little-endian format, NF90_ENDIAN_BIG if it is stored in big-endian format, and NF90_ENDIAN_NATIVE if the endianness is not set, and the variable is not created yet.

These functions return the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The variable ID is invalid for the specified netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_INQ_VAR to find out about a variable named rh in an existing netCDF dataset named foo.nc:

```
    use netcdf
    implicit none
    integer                            :: status, ncid, &
                                          RhVarId       &
                                          numDims, numAtts
 integer, dimension(nf90_max_var_dims) :: rhDimIds
 ...
 status = nf90_open("foo.nc", nf90_NoWrite, ncid)
 if(status /= nf90_NoErr) call handle_error(status)
 ...
 status = nf90_inq_varid(ncid, "rh", RhVarId)
 if(status /= nf90_NoErr) call handle_err(status)
 status = nf90_inquire_variable(ncid, RhVarId, ndims = numDims, natts = numAtts)
 if(status /= nf90_NoErr) call handle_err(status)
 status = nf90_inquire_variable(ncid, RhVarId, dimids = rhDimIds(:numDims))
 if(status /= nf90_NoErr) call handle_err(status)
```

6.7 Get the ID of a variable from the name: NF90_INQ_VARID

Given the name of a varaible, nf90_inq_varid finds the variable ID.

Usage

```
function nf90_inq_varid(ncid, name, varid)
  integer, intent(in) :: ncid
  character (len = *), intent( in) :: name
  integer, intent(out) :: varid
  integer :: nf90_inq_varid
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`name` The variable name. The maximum possible length, in characters, of a variable name is given by the predefined constant NF90_MAX_NAME.

`varid` Variable ID.

These functions return the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- Variable not found.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_INQ_VARID to find out about a variable named rh in an existing netCDF dataset named foo.nc:

```
    use netcdf
    implicit none
    integer                             :: status, ncid, &
                                           RhVarId       &
                                           numDims, numAtts
 integer, dimension(nf90_max_var_dims) :: rhDimIds
 ...
 status = nf90_open("foo.nc", nf90_NoWrite, ncid)
 if(status /= nf90_NoErr) call handle_error(status)
 ...
 status = nf90_inq_varid(ncid, "rh", RhVarId)
 if(status /= nf90_NoErr) call handle_err(status)
 status = nf90_inquire_variable(ncid, RhVarId, ndims = numDims, natts = numAtts)
 if(status /= nf90_NoErr) call handle_err(status)
 status = nf90_inquire_variable(ncid, RhVarId, dimids = rhDimIds(:numDims))
 if(status /= nf90_NoErr) call handle_err(status)
```

6.8 Writing Data Values: NF90_PUT_VAR

The function NF90_PUT_VAR puts one or more data values into the variable of an open netCDF dataset that is in data mode. Required inputs are the netCDF ID, the variable ID, and one or more data values. Optional inputs may indicate the starting position of the data values in the netCDF variable (argument start), the sampling frequency with which data values are written into the netCDF variable (argument stride), and a mapping between the dimensions of the data array and the netCDF variable (argument map). The values to be written are associated with the netCDF variable by assuming that the first dimension of the netCDF variable varies fastest in the Fortran 90 interface. Data values are converted to the external type of the variable, if necessary.

Take care when using the simplest forms of this interface with record variables (variables that use the NF90_UNLIMITED dimension) when you don't specify how many records are to be written. If you try to write all the values of a record variable into a netCDF file that has no record data yet (hence has 0 records), nothing will be written. Similarly, if you try to write all the values of a record variable from an array but there are more records in the file than you assume, more in-memory data will be accessed than you expect, which may cause a segmentation violation. To avoid such problems, it is better to specify start and count arguments for variables that use the NF90_UNLIMITED dimension.

Usage

```
function nf90_put_var(ncid, varid, values, start, count, stride, map)
  integer,                         intent( in) :: ncid, varid
  any valid type, scalar or array of any rank, &
                                   intent( in) :: values
  integer, dimension(:), optional, intent( in) :: start, count, stride, map
  integer                                      :: nf90_put_var
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID.

`values` The data value(s) to be written. The data may be of any type, and may be a scalar or an array of any rank. You cannot put CHARACTER data into a numeric variable or numeric data into a text variable. For numeric data, if the type of data differs from the netCDF variable type, type conversion will occur. See Section "Type Conversion" in *NetCDF Users Guide*.

`start` A vector of integers specifying the index in the variable where the first (or only) of the data values will be written. The indices are relative to 1, so for example, the first data value of a variable would have index (1, 1, ..., 1). The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the last index would correspond to the starting record number for writing the data values.

By default, start(:) = 1.

`count` A vector of integers specifying the number of indices selected along each dimension. To write a single value, for example, specify count as (1, 1, ..., 1). The elements of count correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the last element of count corresponds to a count of the number of records to write.

By default, count(:numDims) = shape(values) and count(numDims + 1:) = 1, where numDims = size(shape(values)).

`stride` A vector of integers that specifies the sampling interval along each dimension of the netCDF variable. The elements of the stride vector correspond, in order, to the netCDF variable's dimensions (stride(1) gives the sampling interval along the most rapidly varying dimension of the netCDF variable). Sampling intervals are specified in type-independent units of elements (a value of 1 selects consecutive elements of the netCDF variable along the corresponding dimension, a value of 2 selects every other element, etc.).

By default, stride(:) = 1.

`imap`
: A vector of integers that specifies the mapping between the dimensions of a netCDF variable and the in-memory structure of the internal data array. The elements of the index mapping vector correspond, in order, to the netCDF variable's dimensions (map(1) gives the distance between elements of the internal array corresponding to the most rapidly varying dimension of the netCDF variable). Distances between elements are specified in units of elements.

  By default, edgeLengths = shape(values), and map = (/ 1, (product(edgeLengths(:i)), i = 1, size(edgeLengths) - 1) /), that is, there is no mapping.

  Use of Fortran 90 intrinsic functions (including reshape, transpose, and spread) may let you avoid using this argument.

Errors

NF90_PUT_VAR1_ type returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The variable ID is invalid for the specified netCDF dataset.
- The specified indices were out of range for the rank of the specified variable. For example, a negative index or an index that is larger than the corresponding dimension length will cause an error.
- The specified value is out of the range of values representable by the external data type of the variable.
- The specified netCDF is in define mode rather than data mode.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_PUT_VAR to set the (4,3,2) element of the variable named rh to 0.5 in an existing netCDF dataset named foo.nc. For simplicity in this example, we assume that we know that rh is dimensioned with lon, lat, and time, so we want to set the value of rh that corresponds to the fourth lon value, the third lat value, and the second time value:

```
use netcdf
implicit none
integer :: ncId, rhVarId, status
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
```

```
status = nf90_put_var(ncid, rhVarId, 0.5, start = (/ 4, 3, 2 /) )
if(status /= nf90_NoErr) call handle_err(status)
```

In this example we use NF90_PUT_VAR to add or change all the values of the variable named rh to 0.5 in an existing netCDF dataset named foo.nc. We assume that we know that rh is dimensioned with lon, lat, and time. In this example we query the netCDF file to discover the lengths of the dimensions, then use the Fortran 90 intrinsic function reshape to create a temporary array of data values which is the same shape as the netCDF variable.

```
use netcdf
implicit none
integer                               :: ncId, rhVarId,status,          &
                                         lonDimID, latDimId, timeDimId, &
                                         numLons, numLats, numTimes,    &
                                         i
integer, dimension(nf90_max_var_dims) :: dimIDs
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
! How big is the netCDF variable, that is, what are the lengths of
!   its constituent dimensions?
status = nf90_inquire_variable(ncid, rhVarId, dimids = dimIDs)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_inquire_dimension(ncid, dimIDs(1), len = numLons)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_inquire_dimension(ncid, dimIDs(2), len = numLats)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_inquire_dimension(ncid, dimIDs(3), len = numTimes)
if(status /= nf90_NoErr) call handle_err(status)
...
! Make a temporary array the same shape as the netCDF variable.
status = nf90_put_var(ncid, rhVarId, &
                      reshape( &
                        (/ (0.5, i = 1, numLons * numLats * numTimes) /) , &
                       shape = (/ numLons, numLats, numTimes /) )
if(status /= nf90_NoErr) call handle_err(status)
```

Here is an example using NF90_PUT_VAR to add or change a section of the variable named rh to 0.5 in an existing netCDF dataset named foo.nc. For simplicity in this example, we assume that we know that rh is dimensioned with lon, lat, and time, that there are ten lon values, five lat values, and three time values, and that we want to replace all the values at the last time.

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 10, numLats = 5, numTimes = 3
```

```
real, dimension(numLons, numLats) &
                   :: rhValues
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
! Fill in all values at the last time
rhValues(:, :) = 0.5
status = nf90_put_var(ncid, rhVarId,rhvalues,       &
                      start = (/ 1, 1, numTimes /), &
                      count = (/ numLats, numLons, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

Here is an example of using NF90_PUT_VAR to write every other point of a netCDF variable named rh having dimensions (6, 4).

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) &
                   :: rhValues = 0.5
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
...
! Fill in every other value using an array section
status = nf90_put_var(ncid, rhVarId, rhValues(::2, ::2), &
                      stride = (/ 2, 2 /))
if(status /= nf90_NoErr) call handle_err(status)
```

The following map vector shows the default mapping between a 2x3x4 netCDF variable and an internal array of the same shape:

```
real,    dimension(2, 3, 4):: a  ! same shape as netCDF variable
integer, dimension(3)      :: map  = (/ 1, 2, 6 /)
                    ! netCDF dimension inter-element distance
                    ! ---------------- ----------------------
                    ! most rapidly varying       1
                    ! intermediate               2 (= map(1)*2)
                    ! most slowly varying        6 (= map(2)*3)
```

Using the map vector above obtains the same result as simply not passing a map vector at all.

Here is an example of using nf90_put_var to write a netCDF variable named rh whose dimensions are the transpose of the Fortran 90 array:

```
use netcdf
implicit none
integer                            :: ncId, rhVarId, status
integer, parameter                 :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) :: rhValues
! netCDF variable has dimensions (numLats, numLons)
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
...
!Write transposed values: map vector would be (/ 1, numLats /) for
!   no transposition
status = nf90_put_var(ncid, rhVarId,rhValues, map = (/ numLons, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

The same effect can be obtained more simply using Fortran 90 intrinsic functions:

```
use netcdf
implicit none
integer                            :: ncId, rhVarId, status
integer, parameter                 :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) :: rhValues
! netCDF variable has dimensions (numLats, numLons)
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_put_var(ncid, rhVarId, transpose(rhValues))
if(status /= nf90_NoErr) call handle_err(status)
```

6.9 Reading Data Values: NF90_GET_VAR

The function NF90_GET_VAR gets one or more data values from a netCDF variable of an open netCDF dataset that is in data mode. Required inputs are the netCDF ID, the variable ID, and a specification for the data values into which the data will be read. Optional inputs may indicate the starting position of the data values in the netCDF variable (argument start), the sampling frequency with which data values are read from the netCDF variable (argument stride), and a mapping between the dimensions of the data array and the netCDF variable (argument map). The values to be read are associated with the netCDF variable by assuming that the first dimension of the netCDF variable varies fastest in the Fortran 90 interface. Data values are converted from the external type of the variable, if necessary.

Take care when using the simplest forms of this interface with record variables (variables that use the NF90_UNLIMITED dimension) when you don't specify how many records are to be read. If you try to read all the values of a record variable into an array but there are more records in the file than you assume, more data will be read than you expect, which may cause a segmentation violation. To avoid such problems, it is better to specify the optional start and count arguments for variables that use the NF90_UNLIMITED dimension.

In netCDF classic model the maximum integer size is NF90_INT, the 4-byte signed integer. Reading variables into an eight-byte integer array from a classic model file will read from an NF90_INT. Reading variables into an eight-byte integer in a netCDF-4/HDF5 (without classic model flag) will read from an NF90_INT64

Usage

```
function nf90_get_var(ncid, varid, values, start, count, stride, map)
  integer,                         intent( in) :: ncid, varid
  any valid type, scalar or array of any rank, &
                                   intent(out) :: values
  integer, dimension(:), optional, intent( in) :: start, count, stride, map
  integer                                      :: nf90_get_var
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID.

`values` The data value(s) to be read. The data may be of any type, and may be a scalar or an array of any rank. You cannot read CHARACTER data from a numeric variable or numeric data from a text variable. For numeric data, if the type of data differs from the netCDF variable type, type conversion will occur. See Section "Type Conversion" in *NetCDF Users Guide*.

`start` A vector of integers specifying the index in the variable from which the first (or only) of the data values will be read. The indices are relative to 1, so for example, the first data value of a variable would have index (1, 1, ..., 1). The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the last index would correspond to the starting record number for writing the data values.

By default, start(:) = 1.

`count` A vector of integers specifying the number of indices selected along each dimension. To read a single value, for example, specify count as (1, 1, ..., 1). The elements of count correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the last element of count corresponds to a count of the number of records to read.

By default, count(:numDims) = shape(values) and count(numDims + 1:) = 1, where numDims = size(shape(values)).

`stride` A vector of integers that specifies the sampling interval along each dimension of the netCDF variable. The elements of the stride vector correspond, in order, to the netCDF variable's dimensions (stride(1) gives the sampling interval along the most rapidly varying dimension of the netCDF variable). Sampling intervals are specified in type-independent units of elements (a value of 1 selects consecutive elements of the netCDF variable along the corresponding dimension, a value of 2 selects every other element, etc.).

By default, stride(:) = 1.

`map` A vector of integers that specifies the mapping between the dimensions of a netCDF variable and the in-memory structure of the internal data array. The elements of the index mapping vector correspond, in order, to the netCDF variable's dimensions (map(1) gives the distance between elements of the internal array corresponding to the most rapidly varying dimension of the netCDF variable). Distances between elements are specified in units of elements.

By default, edgeLengths = shape(values), and map = (/ 1, (product(edgeLengths(:i)), i = 1, size(edgeLengths) - 1) /), that is, there is no mapping.

Use of Fortran 90 intrinsic functions (including reshape, transpose, and spread) may let you avoid using this argument.

Errors

NF90_GET_VAR returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The variable ID is invalid for the specified netCDF dataset.
- The assumed or specified start, count, and stride generate an index which is out of range. Note that no error checking is possible on the map vector.
- One or more of the specified values are out of the range of values representable by the desired type.
- The specified netCDF is in define mode rather than data mode.
- The specified netCDF ID does not refer to an open netCDF dataset.

(As noted above, another possible source of error is using this interface to read all the values of a record variable without specifying the number of records. If there are more records in the file than you assume, more data will be read than you expect!)

Example

Here is an example using NF90_GET_VAR to read the (4,3,2) element of the variable named rh from an existing netCDF dataset named foo.nc. For simplicity in this example, we assume that we know that rh is dimensioned with lon, lat, and time, so we want to read the value of rh that corresponds to the fourth lon value, the third lat value, and the second time value:

```
use netcdf
implicit none
integer :: ncId, rhVarId, status
real    :: rhValue
...
status = nf90_open("foo.nc", nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
-
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
```

```
status = nf90_get_var(ncid, rhVarId, rhValue, start = (/ 4, 3, 2 /) )
if(status /= nf90_NoErr) call handle_err(status)
```

In this example we use NF90_GET_VAR to read all the values of the variable named rh from an existing netCDF dataset named foo.nc. We assume that we know that rh is dimensioned with lon, lat, and time. In this example we query the netCDF file to discover the lengths of the dimensions, then allocate a Fortran 90 array the same shape as the netCDF variable.

```
use netcdf
implicit none
integer                               :: ncId, rhVarId, &
                                         lonDimID, latDimId, timeDimId, &
                                         numLons, numLats, numTimes,    &
                                         status
integer, dimension(nf90_max_var_dims) :: dimIDs
real, dimension(:, :, :), allocatable :: rhValues
...
status = nf90_open("foo.nc", nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
! How big is the netCDF variable, that is, what are the lengths of
!   its constituent dimensions?
status = nf90_inquire_variable(ncid, rhVarId, dimids = dimIDs)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_inquire_dimension(ncid, dimIDs(1), len = numLons)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_inquire_dimension(ncid, dimIDs(2), len = numLats)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_inquire_dimension(ncid, dimIDs(3), len = numTimes)
if(status /= nf90_NoErr) call handle_err(status)
allocate(rhValues(numLons, numLats, numTimes))
...
status = nf90_get_var(ncid, rhVarId, rhValues)
if(status /= nf90_NoErr) call handle_err(status)
```

Here is an example using NF90_GET_VAR to read a section of the variable named rh from an existing netCDF dataset named foo.nc. For simplicity in this example, we assume that we know that rh is dimensioned with lon, lat, and time, that there are ten lon values, five lat values, and three time values, and that we want to replace all the values at the last time.

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 10, numLats = 5, numTimes = 3
real, dimension(numLons, numLats, numTimes) &
                   :: rhValues
```

```
...
status = nf90_open("foo.nc", nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
!Read the values at the last time by passing an array section
status = nf90_get_var(ncid, rhVarId, rhValues(:, :, 3), &
                      start = (/ 1, 1, numTimes /),     &
                      count = (/ numLons, numLats, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

Here is an example of using NF90_GET_VAR to read every other point of a netCDF variable named rh having dimensions (6, 4).

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) &
                   :: rhValues
...
status = nf90_open("foo.nc", nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
...
! Read every other value into an array section
status = nf90_get_var(ncid, rhVarId, rhValues(::2, ::2) &
                      stride = (/ 2, 2 /))
if(status /= nf90_NoErr) call handle_err(status)
```

The following map vector shows the default mapping between a 2x3x4 netCDF variable and an internal array of the same shape:

```
real,    dimension(2, 3, 4):: a  ! same shape as netCDF variable
integer, dimension(3)      :: map  = (/ 1, 2, 6 /)
                    ! netCDF dimension inter-element distance
                    ! ---------------- ----------------------
                    ! most rapidly varying       1
                    ! intermediate               2 (= map(1)*2)
                    ! most slowly varying        6 (= map(2)*3)
```

Using the map vector above obtains the same result as simply not passing a map vector at all.

Here is an example of using nf90_get_var to read a netCDF variable named rh whose dimensions are the transpose of the Fortran 90 array:

```
use netcdf
implicit none
```

```
integer                          :: ncId, rhVarId, status
integer, parameter               :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) :: rhValues
! netCDF variable has dimensions (numLats, numLons)
...
status = nf90_open("foo.nc", nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
...
! Read transposed values: map vector would be (/ 1, numLats /) for
!   no transposition
status = nf90_get_var(ncid, rhVarId,rhValues, map = (/ numLons, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

The same effect can be obtained more simply, though using more memory, using Fortran 90 intrinsic functions:

```
use netcdf
implicit none
integer                          :: ncId, rhVarId, status
integer, parameter               :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) :: rhValues
! netCDF variable has dimensions (numLats, numLons)
real, dimension(numLons, numLats) :: tempValues
...
status = nf90_open("foo.nc", nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_get_var(ncid, rhVarId, tempValues))
if(status /= nf90_NoErr) call handle_err(status)
rhValues(:, :) = transpose(tempValues)
```

6.10 Reading and Writing Character String Values

Character strings are not a primitive netCDF external data type under the classic netCDF data model, in part because FORTRAN does not support the abstraction of variable-length character strings (the FORTRAN LEN function returns the static length of a character string, not its dynamic length). As a result, a character string cannot be written or read as a single object in the netCDF interface. Instead, a character string must be treated as an array of characters, and array access must be used to read and write character strings as variable data in netCDF datasets. Furthermore, variable-length strings are not supported by the netCDF classic interface except by convention; for example, you may treat a zero byte as terminating a character string, but you must explicitly specify the length of strings to be read from and written to netCDF variables.

Character strings as attribute values are easier to use, since the strings are treated as a single unit for access. However, the value of a character-string attribute in the classic netCDF interface is still an array of characters with an explicit length that must be specified when the attribute is defined.

When you define a variable that will have character-string values, use a character-position dimension as the most quickly varying dimension for the variable (the first dimension for the variable in Fortran 90). The length of the character-position dimension will be the maximum string length of any value to be stored in the character-string variable. Space for maximum-length strings will be allocated in the disk representation of character-string variables whether you use the space or not. If two or more variables have the same maximum length, the same character-position dimension may be used in defining the variable shapes.

To write a character-string value into a character-string variable, use either entire variable access or array access. The latter requires that you specify both a corner and a vector of edge lengths. The character-position dimension at the corner should be one for Fortran 90. If the length of the string to be written is n, then the vector of edge lengths will specify n in the character-position dimension, and one for all the other dimensions: (n, 1, 1, ..., 1).

In Fortran 90, fixed-length strings may be written to a netCDF dataset without a terminating character, to save space. Variable-length strings should follow the C convention of writing strings with a terminating zero byte so that the intended length of the string can be determined when it is later read by either C or Fortran 90 programs. It is the users responsibility to provide such null termination.

If you are writing data in the default prefill mode (see next section), you can ensure that simple strings represented as 1-dimensional character arrays are null terminated in the netCDF file by writing fewer characters than the length declared when the variable was defined. That way, the extra unwritten characters will be filled with the default character fill value, which is a null byte. The Fortran intrinsic TRIM function can be used to trim trailing blanks from the character string argument to NF90_PUT_VAR to make the argument shorter than the declared length. If prefill is not on, the data writer must explicitly provide a null terminating byte.

Here is an example illustrating this way of writing strings to character array variables:

```
use netcdf
implicit none
integer status
integer                           :: ncid, oceanStrLenID, oceanId
integer, parameter                :: MaxOceanNameLen = 20
character, (len = MaxOceanNameLen):: ocean
...
status = nf90_create("foo.nc", nf90_NoClobber, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_def_dim(ncid, "oceanStrLen", MaxOceanNameLen, oceanStrLenId)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_def_var(ncid, "ocean", nf90_char, (/ oceanStrLenId /), oceanId)
if(status /= nf90_NoErr) call handle_err(status)
```

```
...
! Leave define mode, which prefills netCDF variables with fill values
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Note that this assignment adds blank fill
ocean = "Pacific"
! Using trim removes trailing blanks, prefill provides null
! termination, so C programs can later get intended string.
status = nf90_put_var(ncid, oceanId, trim(ocean))
if(status /= nf90_NoErr) call handle_err(status)
```

6.11 Fill Values

What happens when you try to read a value that was never written in an open netCDF dataset? You might expect that this should always be an error, and that you should get an error message or an error status returned. You do get an error if you try to read data from a netCDF dataset that is not open for reading, if the variable ID is invalid for the specified netCDF dataset, or if the specified indices are not properly within the range defined by the dimension lengths of the specified variable. Otherwise, reading a value that was not written returns a special fill value used to fill in any undefined values when a netCDF variable is first written.

You may ignore fill values and use the entire range of a netCDF external data type, but in this case you should make sure you write all data values before reading them. If you know you will be writing all the data before reading it, you can specify that no prefilling of variables with fill values will occur by calling writing. This may provide a significant performance gain for netCDF writes.

The variable attribute _FillValue may be used to specify the fill value for a variable. There are default fill values for each type, defined in module netcdf: NF90_FILL_CHAR, NF90_FILL_INT1 (same as NF90_FILL_BYTE), NF90_FILL_INT2 (same as NF90_FILL_SHORT), NF90_FILL_INT, NF90_FILL_REAL (same as NF90_FILL_FLOAT), and NF90_FILL_DOUBLE

The netCDF byte and character types have different default fill values. The default fill value for characters is the zero byte, a useful value for detecting the end of variable-length C character strings. If you need a fill value for a byte variable, it is recommended that you explicitly define an appropriate _FillValue attribute, as generic utilities such as ncdump will not assume a default fill value for byte variables.

Type conversion for fill values is identical to type conversion for other values: attempting to convert a value from one type to another type that can't represent the value results in a range error. Such errors may occur on writing or reading values from a larger type (such as double) to a smaller type (such as float), if the fill value for the larger type cannot be represented in the smaller type.

6.12 NF90_RENAME_VAR

The function NF90_RENAME_VAR changes the name of a netCDF variable in an open netCDF dataset. If the new name is longer than the old name, the netCDF dataset must be in define mode. You cannot rename a variable to have the name of any existing variable.

Usage

```
function nf90_rename_var(ncid, varid, newname)
  integer,             intent( in) :: ncid, varid
  character (len = *), intent( in) :: newname
  integer                          :: nf90_rename_var
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID.

`newname` New name for the specified variable.

Errors

NF90_RENAME_VAR returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The new name is in use as the name of another variable.
- The variable ID is invalid for the specified netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_RENAME_VAR to rename the variable rh to rel_hum in an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncId, rhVarId, status
...
status = nf90_open("foo.nc", nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
...
status = nf90_inq_varid(ncid, "rh", rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_redef(ncid)  ! Enter define mode to change variable name
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_rename_var(ncid, rhVarId, "rel_hum")
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_enddef(ncid) ! Leave define mode
if(status /= nf90_NoErr) call handle_err(status)
```

6.13 Change between Collective and Independent Parallel Access: NF90_VAR_PAR_ACCESS

The function NF90_VAR_PAR_ACCESS changes whether read/write operations on a parallel file system are performed collectively or independently (the default) on the variable. This function can only be called if the file was created (see Section 2.5 [NF90_CREATE], page 9) or opened (see Section 2.6 [NF90_OPEN], page 11) for parallel I/O.

This function is only available if the netCDF library was built with parallel I/O enabled.

Calling this function affects only the open file - information about whether a variable is to be accessed collectively or independently is not written to the data file. Every time you open a file on a parallel file system, all variables default to independent operations. The change of a variable to collective access lasts only as long as that file is open.

The variable can be changed from collective to independent, and back, as often as desired.

Classic and 64-bit offset files, when opened for parallel access, use the parallel-netcdf (a.k.a. pnetcdf) library, which does not allow per-variable changes of access mode - the entire file must be access independently or collectively. For classic and 64-bit offset files, the nf90_var_par_access function changes the access for all variables in the file.

Usage

```
function nf90_var_par_access(ncid, varid, access)
  integer, intent(in) :: ncid
  integer, intent(in) :: varid
  integer, intent(in) :: access
  integer :: nf90_var_par_access
end function nf90_var_par_access
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN (see Section 2.6 [NF90_OPEN], page 11) or NF90_CREATE (see Section 2.5 [NF90_CREATE], page 9).

`varid` Variable ID.

`access` NF90_INDEPENDENT to set this variable to independent operations. NF90_COLLECTIVE to set it to collective operations.

Return Values

`NF90_NOERR`
No error.

`NF90_ENOTVAR`
No variable found.

`NF90_NOPAR`
File not opened for parallel access.

Example

This example comes from test program nf_test/f90tst_parallel.f90. For this test to be run, netCDF must have been built with a parallel-enabled HDF5, and –enable-parallel-tests must have been used when configuring netcdf.

```
! Reopen the file.
call handle_err(nf90_open(FILE_NAME, nf90_nowrite, ncid, comm = MPI_COMM_WORLD, &
     info = MPI_INFO_NULL))

! Set collective access on this variable. This will cause all
! reads/writes to happen together on every processor.
call handle_err(nf90_var_par_access(ncid, varid, nf90_collective))

! Read this processor's data.
call handle_err(nf90_get_var(ncid, varid, data_in, start = start, count = count))
```

7 Attributes

7.1 Attributes Introduction

Attributes may be associated with each netCDF variable to specify such properties as units, special values, maximum and minimum valid values, scaling factors, and offsets. Attributes for a netCDF dataset are defined when the dataset is first created, while the netCDF dataset is in define mode. Additional attributes may be added later by reentering define mode. A netCDF attribute has a netCDF variable to which it is assigned, a name, a type, a length, and a sequence of one or more values. An attribute is designated by its variable ID and name. When an attribute name is not known, it may be designated by its variable ID and number in order to determine its name, using the function NF90_INQ_ATTNAME.

The attributes associated with a variable are typically defined immediately after the variable is created, while still in define mode. The data type, length, and value of an attribute may be changed even when in data mode, as long as the changed attribute requires no more space than the attribute as originally defined.

It is also possible to have attributes that are not associated with any variable. These are called global attributes and are identified by using NF90_GLOBAL as a variable pseudo-ID. Global attributes are usually related to the netCDF dataset as a whole and may be used for purposes such as providing a title or processing history for a netCDF dataset.

Attributes are much more useful when they follow established community conventions. See Section “Attribute Conventions” in *The NetCDF Users Guide*.

Operations supported on attributes are:

- Create an attribute, given its variable ID, name, data type, length, and value.
- Get attribute's data type and length from its variable ID and name.
- Get attribute's value from its variable ID and name.
- Copy attribute from one netCDF variable to another.
- Get name of attribute from its number.
- Rename an attribute.
- Delete an attribute.

7.2 Create an Attribute: NF90_PUT_ATT

The function NF90_PUT_ATTadds or changes a variable attribute or global attribute of an open netCDF dataset. If this attribute is new, or if the space required to store the attribute is greater than before, the netCDF dataset must be in define mode.

Usage

Although it's possible to create attributes of all types, text and double attributes are adequate for most purposes.

```
function nf90_put_att(ncid, varid, name, values)
  integer,            intent( in) :: ncid, varid
  character(len = *), intent( in) :: name
  any valid type, scalar or array of rank 1, &
```

```
                            intent( in) :: values
     integer                            :: nf90_put_att
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID of the variable to which the attribute will be assigned or NF90_GLOBAL for a global attribute.

`name` Attribute name. Attribute name conventions are assumed by some netCDF generic applications, e.g., '`units`' as the name for a string attribute that gives the units for a netCDF variable. See Section "Attribute Conventions" in *The NetCDF Users Guide*.

`values` An array of attribute values. Values may be supplied as scalars or as arrays of rank one (one dimensional vectors). The external data type of the attribute is set to match the internal representation of the argument, that is if values is a two byte integer array, the attribute will be of type NF90_INT2. Fortran 90 intrinsic functions can be used to convert attributes to the desired type.

Errors

NF90_PUT_ATT returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The variable ID is invalid for the specified netCDF dataset.
- The specified netCDF type is invalid.
- The specified length is negative.
- The specified open netCDF dataset is in data mode and the specified attribute would expand.
- The specified open netCDF dataset is in data mode and the specified attribute does not already exist.
- The specified netCDF ID does not refer to an open netCDF dataset.
- The number of attributes for this variable exceeds NF90_MAX_ATTRS.

Example

Here is an example using NF90_PUT_ATT to add a variable attribute named valid_range for a netCDF variable named rh and a global attribute named title to an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status, RHVarID
...
status = nf90_open("foo.nc", nf90_write, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Enter define mode so we can add the attribute
status = nf90_redef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

```
! Get the variable ID for "rh"...
status = nf90_inq_varid(ncid, "rh", RHVarID)
if (status /= nf90_noerr) call handle_err(status)
! ...  put the range attribute, setting it to eight byte reals...
status = nf90_put_att(ncid, RHVarID, "valid_range", real((/ 0, 100 /))
! ... and the title attribute.
if (status /= nf90_noerr) call handle_err(status)
status = nf90_put_att(ncid, RHVarID, "title", "example netCDF dataset") )
if (status /= nf90_noerr) call handle_err(status)
! Leave define mode
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

7.3 Get Information about an Attribute: NF90_INQUIRE_ATTRIBUTE and NF90_INQ_ATTNAME

The function NF90_INQUIRE_ATTRIBUTE returns information about a netCDF attribute given the variable ID and attribute name. Information about an attribute includes its type, length, name, and number. See NF90_GET_ATT for getting attribute values.

The function NF90_INQ_ATTNAME gets the name of an attribute, given its variable ID and number. This function is useful in generic applications that need to get the names of all the attributes associated with a variable, since attributes are accessed by name rather than number in all other attribute functions. The number of an attribute is more volatile than the name, since it can change when other attributes of the same variable are deleted. This is why an attribute number is not called an attribute ID.

Usage

```
function nf90_inquire_attribute(ncid, varid, name, xtype, len, attnum)
  integer,             intent( in)           :: ncid, varid
  character (len = *), intent( in)           :: name
  integer,             intent(out), optional :: xtype, len, attnum
  integer                                    :: nf90_inquire_attribute
function nf90_inq_attname(ncid, varid, attnum, name)
  integer,             intent( in) :: ncid, varid, attnum
  character (len = *), intent(out) :: name
  integer                          :: nf90_inq_attname
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID of the attribute's variable, or NF90_GLOBAL for a global attribute.

`name` Attribute name. For NF90_INQ_ATTNAME, this is a pointer to the location for the returned attribute name.

`xtype` Returned attribute type, one of the set of predefined netCDF external data types. The valid netCDF external data types are NF90_BYTE, NF90_CHAR, NF90_SHORT, NF90_INT, NF90_FLOAT, and NF90_DOUBLE.

`len` Returned number of values currently stored in the attribute. For a string-valued attribute, this is the number of characters in the string.

`attnum` For NF90_INQ_ATTNAME, the input attribute number; for NF90_INQ_ATTID, the returned attribute number. The attributes for each variable are numbered from 1 (the first attribute) to NATTS, where NATTS is the number of attributes for the variable, as returned from a call to NF90_INQ_VARNATTS.

(If you already know an attribute name, knowing its number is not very useful, because accessing information about an attribute requires its name.)

Errors

Each function returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The variable ID is invalid for the specified netCDF dataset.
- The specified attribute does not exist.
- The specified netCDF ID does not refer to an open netCDF dataset.
- For NF90_INQ_ATTNAME, the specified attribute number is negative or more than the number of attributes defined for the specified variable.

Example

Here is an example using NF90_INQUIRE_ATTRIBUTE to inquire about the lengths of an attribute named valid_range for a netCDF variable named rh and a global attribute named title in an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid, status
integer :: RHVarID                       ! Variable ID
integer :: validRangeLength, titleLength ! Attribute lengths
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Get the variable ID for "rh"...
status = nf90_inq_varid(ncid, "rh", RHVarID)
if (status /= nf90_noerr) call handle_err(status)
! ...  get the length of the "valid_range" attribute...
status = nf90_inquire_attribute(ncid, RHVarID, "valid_range", &
                          len = validRangeLength)
if (status /= nf90_noerr) call handle_err(status)
! ... and the global title attribute.
status = nf90_inquire_attribute(ncid, nf90_global, "title", len = titleLength)
if (status /= nf90_noerr) call handle_err(status)
```

7.4 Get Attribute's Values: NF90_GET_ATT

Function nf90_get_att gets the value(s) of a netCDF attribute, given its variable ID and name.

Usage

```
function nf90_get_att(ncid, varid, name, values)
  integer,            intent( in) :: ncid, varid
  character(len = *), intent( in) :: name
  any valid type, scalar or array of rank 1, &
                      intent(out) :: values
  integer                         :: nf90_get_att
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` Variable ID of the attribute's variable, or NF90_GLOBAL for a global attribute.

`name` Attribute name.

`values` Returned attribute values. All elements of the vector of attribute values are returned, so you must provide enough space to hold them. If you don't know how much space to reserve, call NF90_INQUIRE_ATTRIBUTE first to find out the length of the attribute. If there is only a single attribute values may be a scalar. If the attribute is of type character values should be a variable of type character with the len Fortran 90 attribute set to an appropriate value (i.e. character (len = 80) :: values). You cannot read character data from a numeric variable or numeric data from a text variable. For numeric data, if the type of data differs from the netCDF variable type, type conversion will occur. See Section "Type Conversion" in *NetCDF Users Guide*.

Errors

NF90_GET_ATT_ type returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The variable ID is invalid for the specified netCDF dataset.
- The specified attribute does not exist.
- The specified netCDF ID does not refer to an open netCDF dataset.
- One or more of the attribute values are out of the range of values representable by the desired type.

Example

Here is an example using NF90_GET_ATT to determine the values of an attribute named valid_range for a netCDF variable named rh and a global attribute named title in an existing netCDF dataset named foo.nc. In this example, it is assumed that we don't know how many values will be returned, so we first inquire about the length of the attributes to make sure we have enough space to store them:

```
use netcdf
implicit none
```

```
integer              :: ncid, status
integer              :: RHVarID                       ! Variable ID
integer              :: validRangeLength, titleLength ! Attribute lengths
real, dimension(:), allocatable, &
                     :: validRange
character (len = 80) :: title
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Find the lengths of the attributes
status = nf90_inq_varid(ncid, "rh", RHVarID)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_inquire_attribute(ncid, RHVarID, "valid_range", &
                          len = validRangeLength)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_inquire_attribute(ncid, nf90_global, "title", len = titleLength)
if (status /= nf90_noerr) call handle_err(status)
...
!Allocate space to hold attribute values, check string lengths
allocate(validRange(validRangeLength), stat = status)
if(status /= 0 .or. len(title) < titleLength)
  print *, "Not enough space to put attribute values."
  exit
end if
! Read the attributes.
status = nf90_get_att(ncid, RHVarID, "valid_range", validRange)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_get_att(ncid, nf90_global, "title", title)
if (status /= nf90_noerr) call handle_err(status)
```

7.5 Copy Attribute from One NetCDF to Another: NF90_COPY_ATT

The function NF90_COPY_ATT copies an attribute from one open netCDF dataset to another. It can also be used to copy an attribute from one variable to another within the same netCDF dataset.

If used to copy an attribute of user-defined type, then that user-defined type must already be defined in the target file. In the case of user-defined attributes, enddef/redef is called for ncid_in and ncid_out if they are in define mode. (This is the ensure that all user-defined types are committed to the file(s) before the copy is attempted.)

Usage

```
function nf90_copy_att(ncid_in, varid_in, name, ncid_out, varid_out)
  integer,             intent( in) :: ncid_in,  varid_in
  character (len = *), intent( in) :: name
  integer,             intent( in) :: ncid_out, varid_out
```

```
integer                            :: nf90_copy_att
```

`ncid_in` The netCDF ID of an input netCDF dataset from which the attribute will be copied, from a previous call to NF90_OPEN or NF90_CREATE.

`varid_in` ID of the variable in the input netCDF dataset from which the attribute will be copied, or NF90_GLOBAL for a global attribute.

`name` Name of the attribute in the input netCDF dataset to be copied.

`ncid_out` The netCDF ID of the output netCDF dataset to which the attribute will be copied, from a previous call to NF90_OPEN or NF90_CREATE. It is permissible for the input and output netCDF IDs to be the same. The output netCDF dataset should be in define mode if the attribute to be copied does not already exist for the target variable, or if it would cause an existing target attribute to grow.

`varid_out`
ID of the variable in the output netCDF dataset to which the attribute will be copied, or NF90_GLOBAL to copy to a global attribute.

Errors

NF90_COPY_ATT returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The input or output variable ID is invalid for the specified netCDF dataset.
- The specified attribute does not exist.
- The output netCDF is not in define mode and the attribute is new for the output dataset is larger than the existing attribute.
- The input or output netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_COPY_ATT to copy the variable attribute units from the variable rh in an existing netCDF dataset named foo.nc to the variable avgrh in another existing netCDF dataset named bar.nc, assuming that the variable avgrh already exists, but does not yet have a units attribute:

```
use netcdf
implicit none
integer :: ncid1, ncid2, status
integer :: RHVarID, avgRHVarID    ! Variable ID
...
status = nf90_open("foo.nc", nf90_nowrite, ncid1)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_open("bar.nc", nf90_write, ncid2)
if (status /= nf90_noerr) call handle_err(status)
...
! Find the IDs of the variables
status = nf90_inq_varid(ncid1, "rh", RHVarID)
```

```
if (status /= nf90_noerr) call handle_err(status)
status = nf90_inq_varid(ncid1, "avgrh", avgRHVarID)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_redef(ncid2)   ! Enter define mode
if (status /= nf90_noerr) call handle_err(status)
! Copy variable attribute from "rh" in file 1 to "avgrh" in file 1
status = nf90_copy_att(ncid1, RHVarID, "units", ncid2, avgRHVarID)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_enddef(ncid2)
if (status /= nf90_noerr) call handle_err(status)
```

7.6 Rename an Attribute: NF90_RENAME_ATT

The function NF90_RENAME_ATT changes the name of an attribute. If the new name is longer than the original name, the netCDF dataset must be in define mode. You cannot rename an attribute to have the same name as another attribute of the same variable.

Usage

```
function nf90_rename_att(ncid, varid, curname, newname)
  integer,             intent( in) :: ncid,  varid
  character (len = *), intent( in) :: curname, newname
  integer                          :: nf90_rename_att
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE

`varid` ID of the attribute's variable, or NF90_GLOBAL for a global attribute

`curname` The current attribute name.

`newname` The new name to be assigned to the specified attribute. If the new name is longer than the current name, the netCDF dataset must be in define mode.

Errors

NF90_RENAME_ATT returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified variable ID is not valid.
- The new attribute name is already in use for another attribute of the specified variable.
- The specified netCDF dataset is in data mode and the new name is longer than the old name.
- The specified attribute does not exist.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_RENAME_ATT to rename the variable attribute units to Units for a variable rh in an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid1, status
integer :: RHVarID         ! Variable ID
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Find the IDs of the variables
status = nf90_inq_varid(ncid, "rh", RHVarID)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_rename_att(ncid, RHVarID, "units", "Units")
if (status /= nf90_noerr) call handle_err(status)
```

7.7 NF90_DEL_ATT

The function NF90_DEL_ATT deletes a netCDF attribute from an open netCDF dataset. The netCDF dataset must be in define mode.

Usage

```
function nf90_del_att(ncid, varid, name)
  integer,             intent( in) :: ncid, varid
  character (len = *), intent( in) :: name
  integer                          :: nf90_del_att
```

`ncid` NetCDF ID, from a previous call to NF90_OPEN or NF90_CREATE.

`varid` ID of the attribute's variable, or NF90_GLOBAL for a global attribute.

`name` The name of the attribute to be deleted.

Errors

NF90_DEL_ATT returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The specified variable ID is not valid.
- The specified netCDF dataset is in data mode.
- The specified attribute does not exist.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using NF90_DEL_ATT to delete the variable attribute Units for a variable rh in an existing netCDF dataset named foo.nc:

```
use netcdf
implicit none
integer :: ncid1, status
```

```
integer :: RHVarID         ! Variable ID
...
status = nf90_open("foo.nc", nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
...
! Find the IDs of the variables
status = nf90_inq_varid(ncid, "rh", RHVarID)
if (status /= nf90_noerr) call handle_err(status)
...
status = nf90_redef(ncid)   ! Enter define mode
if (status /= nf90_noerr) call handle_err(status)
status = nf90_del_att(ncid, RHVarID, "Units")
if (status /= nf90_noerr) call handle_err(status)
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

Appendix A Appendix A - Summary of Fortran 90 Interface

Dataset Functions

```
function nf90_inq_libvers()
  character(len = 80) :: nf90_inq_libvers
function nf90_strerror(ncerr)
  integer, intent( in) :: ncerr
  character(len = 80)  :: nf90_strerror
function nf90_create(path, cmode, ncid)
  character (len = *), intent(in   ) :: path
  integer,             intent(in   ) :: cmode
  integer, optional,   intent(in   ) :: initialsize
  integer, optional,   intent(inout) :: chunksize
  integer,             intent(  out) :: ncid
  integer                            :: nf90_create
function nf90_open(path, mode, ncid, chunksize)
  character (len = *), intent(in   ) :: path
  integer,             intent(in   ) :: mode
  integer,             intent(  out) :: ncid
  integer, optional,   intent(inout) :: chunksize
  integer                            :: nf90_open
function nf90_set_fill(ncid, fillmode, old_mode)
  integer, intent( in) :: ncid, fillmode
  integer, intent(out) :: old_mode
  integer              :: nf90_set_fill
function nf90_redef(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_redef
function nf90_enddef(ncid, h_minfree, v_align, v_minfree, r_align)
  integer,           intent( in) :: ncid
  integer, optional, intent( in) :: h_minfree, v_align, v_minfree, r_align
  integer                        :: nf90_enddef
function nf90_sync(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_sync
function nf90_abort(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_abort
function nf90_close(ncid)
  integer, intent( in) :: ncid
  integer              :: nf90_close
function nf90_Inquire(ncid, nDimensions, nVariables, nAttributes, &
                   unlimitedDimId)
  integer,           intent( in) :: ncid
  integer, optional, intent(out) :: nDimensions, nVariables, nAttributes, &
                           unlimitedDimId
  integer                        :: nf90_Inquire
```

Dimension functions

```
function nf90_def_dim(ncid, name, len, dimid)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent( in) :: len
  integer,             intent(out) :: dimid
  integer                          :: nf90_def_dim
function nf90_inq_dimid(ncid, name, dimid)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent(out) :: dimid
  integer                          :: nf90_inq_dimid
function nf90_inquire_dimension(ncid, dimid, name, len)
  integer,                       intent( in) :: ncid, dimid
  character (len = *), optional, intent(out) :: name
  integer,             optional, intent(out) :: len
  integer                                    :: nf90_inquire_dimension
function nf90_rename_dim(ncid, dimid, name)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent( in) :: dimid
  integer                          :: nf90_rename_dim
```

Variable functions

```
function nf90_def_var(ncid, name, xtype, dimids, varid)
  integer,               intent( in) :: ncid
  character (len = *),   intent( in) :: name
  integer,               intent( in) :: xtype
  integer, dimension(:), intent( in) :: dimids ! May be omitted, scalar,
                                            ! vector
  integer                            :: nf90_def_var
function nf90_inq_varid(ncid, name, varid)
  integer,             intent( in) :: ncid
  character (len = *), intent( in) :: name
  integer,             intent(out) :: varid
  integer                          :: nf90_inq_varid
function nf90_inquire_variable(ncid, varid, name, xtype, ndims, &
                               dimids, nAtts)
  integer,                         intent( in) :: ncid, varid
  character (len = *),   optional, intent(out) :: name
  integer,               optional, intent(out) :: xtype, ndims
  integer, dimension(*), optional, intent(out) :: dimids
  integer,               optional, intent(out) :: nAtts
  integer                                      :: nf90_inquire_variable
function nf90_put_var(ncid, varid, values, start, stride, map)
  integer,                         intent( in) :: ncid, varid
  any valid type, scalar or array of any rank, &
```

```
                                 intent( in) :: values
    integer, dimension(:), optional, intent( in) :: start, count, stride, map
    integer                                      :: nf90_put_var
  function nf90_get_var(ncid, varid, values, start, stride, map)
    integer,                         intent( in) :: ncid, varid
    any valid type, scalar or array of any rank, &
                                     intent(out) :: values
    integer, dimension(:), optional, intent( in) :: start, count, stride, map
    integer                                      :: nf90_get_var
  function nf90_rename_var(ncid, varid, newname)
    integer,             intent( in) :: ncid, varid
    character (len = *), intent( in) :: newname
    integer                          :: nf90_rename_var
```

Attribute functions

```
  function nf90_inquire_attribute(ncid, varid, name, xtype, len, attnum)
    integer,             intent( in)           :: ncid, varid
    character (len = *), intent( in)           :: name
    integer,             intent(out), optional :: xtype, len, attnum
    integer                                    :: nf90_inquire_attribute
  function nf90_inq_attname(ncid, varid, attnum, name)
    integer,             intent( in) :: ncid, varid, attnum
    character (len = *), intent(out) :: name
    integer                          :: nf90_inq_attname
  function nf90_put_att(ncid, varid, name, values)
    integer,            intent( in) :: ncid, varid
    character(len = *), intent( in) :: name
    any valid type, scalar or array of rank 1, &
                        intent( in) :: values
    integer                         :: nf90_put_att
  function nf90_get_att(ncid, varid, name, values)
    integer,            intent( in) :: ncid, varid
    character(len = *), intent( in) :: name
    any valid type, scalar or array of rank 1, &
                        intent(out) :: values
    integer                          :: nf90_get_att
  function nf90_copy_att(ncid_in, varid_in, name, ncid_out, varid_out)
    integer,             intent( in) :: ncid_in,  varid_in
    character (len = *), intent( in) :: name
    integer,             intent( in) :: ncid_out, varid_out
    integer                          :: nf90_copy_att
  function nf90_rename_att(ncid, varid, curname, newname)
    integer,             intent( in) :: ncid,  varid
    character (len = *), intent( in) :: curname, newname
    integer                          :: nf90_rename_att
  function nf90_del_att(ncid, varid, name)
    integer,             intent( in) :: ncid, varid
```

```
character (len = *), intent( in) :: name
integer                          :: nf90_del_att
```

Appendix B Appendix B - FORTRAN 77 to Fortran 90 Transition Guide

The new Fortran 90 interface

The Fortran 90 interface to the netCDF library closely follows the FORTRAN 77 interface. In most cases, function and constant names and argument lists are the same, except that nf90_ replaces nf_ in names. The Fortran 90 interface is much smaller than the FORTRAN 77 interface, however. This has been accomplished by using optional arguments and overloaded functions wherever possible.

Because FORTRAN 77 is a subset of Fortran 90, there is no reason to modify working FORTRAN code to use the Fortran 90 interface. New code, however, can easily be patterned after existing FORTRAN while taking advantage of the simpler interface. Some compilers may provide additional support when using Fortran 90. For example, compilers may issue warnings if arguments with intent(in) are not set before they are passed to a procedure.

The Fortran 90 interface is currently implemented as a set of wrappers around the base FORTRAN subroutines in the netCDF distribution. Future versions may be implemented entirely in Fortran 90, adding additional error checking possibilities.

Changes to Inquiry functions

In the Fortran 90 interface there are two inquiry functions each for dimensions, variables, and attributes, and a single inquiry function for datasets. These functions take optional arguments, allowing users to request only the information they need. These functions replace the many-argument and single-argument inquiry functions in the FORTRAN interface.

As an example, compare the attribute inquiry functions in the Fortran 90 interface

```
function nf90_inquire_attribute(ncid, varid, name, xtype, len, attnum)
  integer,             intent( in)           :: ncid, varid
  character (len = *), intent( in)           :: name
  integer,             intent(out), optional :: xtype, len, attnum
  integer                                    :: nf90_inquire_attribute
function nf90_inq_attname(ncid, varid, attnum, name)
  integer,             intent( in) :: ncid, varid, attnum
  character (len = *), intent(out) :: name
  integer                          :: nf90_inq_attname
```

with those in the FORTRAN interface

```
INTEGER FUNCTION  NF_INQ_ATT       (NCID, VARID, NAME, xtype, len)
INTEGER FUNCTION  NF_INQ_ATTID     (NCID, VARID, NAME, attnum)
INTEGER FUNCTION  NF_INQ_ATTTYPE   (NCID, VARID, NAME, xtype)
INTEGER FUNCTION  NF_INQ_ATTLEN    (NCID, VARID, NAME, len)
INTEGER FUNCTION  NF_INQ_ATTNAME   (NCID, VARID, ATTNUM, name)
```

Changes to put and get function

The biggest simplification in the Fortran 90 is in the nf90_put_var and nf90_get_var functions. Both functions are overloaded: the values argument can be a scalar or an array any rank (7 is the maximum rank allowed by Fortran 90), and may be of any numeric type or the default character type. The netCDF library provides transparent conversion between the external representation of the data and the desired internal representation.

The start, count, stride, and map arguments to nf90_put_var and nf90_get_var are optional. By default, data is read from or written to consecutive values of starting at the origin of the netCDF variable; the shape of the argument determines how many values are read from or written to each dimension. Any or all of these arguments may be supplied to override the default behavior.

Note also that Fortran 90 allows arbitrary array sections to be passed to any procedure, which may greatly simplify programming. For examples see Section 6.8 [NF90_PUT_VAR], page 72 and Section 6.9 [NF90_GET_VAR], page 77.

Index

A

attributes, adding ... 4

C

common netcdf commands ... 1
compiling with netCDF library ... 6
compound types, overview ... 46

D

dataset, creating ... 1
datasets, overview ... 7
dimensions, adding ... 4

E

enum type ... 58
error handling ... 5

F

fill ... 68

G

groups, overview ... 23

I

interface descriptions ... 7

L

linking to netCDF library ... 6

N

NF90_ABORT ... 19
NF90_ABORT , example ... 19
NF90_CLOSE ... 16
NF90_CLOSE , example ... 16
NF90_CLOSE, typical use ... 1
NF90_COPY_ATT ... 94
NF90_COPY_ATT, example ... 94
NF90_CREATE ... 9
NF90_CREATE , example ... 9
NF90_CREATE, typical use ... 1
NF90_DEF_COMPOUND ... 46
NF90_DEF_DIM ... 35
NF90_DEF_DIM, example ... 35
NF90_DEF_DIM, typical use ... 1
NF90_DEF_ENUM ... 58
NF90_DEF_GRP ... 32
NF90_DEF_OPAQUE ... 56
NF90_DEF_VAR ... 64
NF90_DEF_VAR, example ... 64
NF90_DEF_VAR, typical use ... 1
NF90_DEF_VAR_FILL ... 68
NF90_DEF_VLEN ... 54
NF90_DEL_ATT ... 97
NF90_DEL_ATT , example ... 97
NF90_ENDDEF ... 14
NF90_ENDDEF , example ... 14
NF90_ENDDEF, typical use ... 1
NF90_FREE_VLEN ... 56
NF90_GET_ATT ... 93
NF90_GET_ATT, example ... 93
NF90_GET_ATT, typical use ... 2
NF90_GET_VAR ... 77
NF90_GET_VAR, example ... 77
NF90_GET_VAR, typical use ... 2
NF90_GET_VLEN_ELEMENT ... 45
NF90_INQ_ATTNAME ... 91
NF90_INQ_ATTNAME, example ... 91
NF90_INQ_ATTNAME, typical use ... 2
NF90_INQ_CMP_FIELDDIM_SIZES ... 51
NF90_INQ_COMPOUND ... 50
NF90_INQ_COMPOUND_FIELD ... 51
NF90_INQ_COMPOUND_FIELDINDEX ... 51
NF90_INQ_COMPOUND_FIELDNAME ... 51
NF90_INQ_COMPOUND_FIELDNDIMS ... 51
NF90_INQ_COMPOUND_FIELDOFFSET ... 51
NF90_INQ_COMPOUND_FIELDTYPE ... 51
NF90_INQ_COMPOUND_NAME ... 50
NF90_INQ_COMPOUND_NFIELDS ... 50
NF90_INQ_COMPOUND_SIZE ... 50
NF90_INQ_DIMID ... 36
NF90_INQ_DIMID , example ... 36
NF90_INQ_DIMID, typical use ... 2
NF90_INQ_DIMIDS ... 25
NF90_INQ_ENUM ... 60
NF90_INQ_ENUM_IDENT ... 62
nf90_inq_enum_member ... 61
NF90_INQ_GRP_PARENT ... 29, 30, 31
NF90_INQ_GRPNAME ... 27
NF90_INQ_GRPNAME_FULL ... 28
NF90_INQ_GRPNAME_LEN ... 26
NF90_INQ_GRPS ... 24
NF90_INQ_LIBVERS ... 8
NF90_INQ_LIBVERS, example ... 8
NF90_INQ_NCID ... 23
NF90_INQ_OPAQUE ... 57
NF90_INQ_TYPE ... 42
nf90_inq_typeid ... 42
NF90_INQ_TYPEIDS ... 41
NF90_INQ_USER_TYPE ... 43
NF90_INQ_VAR_FILL ... 69

NF90_INQ_VARID ... 71
NF90_INQ_VARID, example ... 71
NF90_INQ_VARID, typical use ... 2, 3
NF90_INQ_VARIDS ... 25
NF90_INQ_VLEN ... 55
NF90_INQUIRE, typical use ... 2
NF90_INQUIRE_ATTRIBUTE ... 91
NF90_INQUIRE_ATTRIBUTE, example ... 91
NF90_INQUIRE_ATTRIBUTE, typical use ... 2
NF90_INQUIRE_DIMENSION ... 37
NF90_INQUIRE_DIMENSION , example ... 37
NF90_INQUIRE_DIMENSION, typical use ... 2
NF90_INQUIRE_VARIABLE ... 70
NF90_INQUIRE_VARIABLE , example ... 70
NF90_INQUIRE_VARIABLE, typical use ... 2
NF90_INSERT_ARRAY_COMPOUND ... 49
NF90_INSERT_COMPOUND ... 48
NF90_INSERT_ENUM ... 59
NF90_OPEN ... 11
NF90_OPEN , example ... 11
NF90_OPEN, typical use ... 2
NF90_PUT_ATT ... 89
NF90_PUT_ATT, example ... 89
NF90_PUT_ATT, typical use ... 1, 3
NF90_PUT_VAR ... 72
NF90_PUT_VAR, example ... 72
NF90_PUT_VAR, typical use ... 1, 3
NF90_PUT_VLEN_ELEMENT ... 44
NF90_REDEF ... 13
NF90_REDEF , example ... 13
NF90_REDEF, typical use ... 4
NF90_RENAME_ATT ... 96
NF90_RENAME_ATT, example ... 96
NF90_RENAME_DIM ... 38
NF90_RENAME_DIM , example ... 38
NF90_RENAME_VAR ... 85
NF90_RENAME_VAR , example ... 85
NF90_SET_FILL ... 20
NF90_SET_FILL , example ... 20
NF90_STRERROR ... 8
NF90_STRERROR, example ... 8
NF90_STRERROR, introduction ... 5
NF90_SYNC ... 18
NF90_SYNC , example ... 18
NF90_VAR_PAR_ACCESS ... 86
NF90_VAR_PAR_ACCESS, example ... 86

O

opaque type ... 56

R

reading dataset with unknown names ... 2

U

user defined types ... 41
user defined types, overview ... 41
users' guide, netcdf ... 1

V

variable length array type, overview ... 41
variable length arrays ... 53
variables, adding ... 4
variables, fill ... 68
VLEN ... 53
VLEN, defining ... 54, 55, 56

W

writing to existing dataset ... 3