The NetCDF C Interface Guide

NetCDF Version 3.6.3
Last Updated 29 January 2008

Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies
Unidata Program Center

The Unidata Program Center is managed by the University Corporation for Atmospheric Research and sponsored by the National Science Foundation. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

This document describes the C interface to the netCDF library; it applies to netCDF version 3.6.3 and was last updated on 29 January 2008.

For a complete description of the netCDF format and utilities see section “Top” in *The NetCDF Users Guide*.

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.) 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:

```
nc_create           /* create netCDF dataset: enter define mode */
     ...
   nc_def_dim       /* define dimensions: from name and length */
     ...
   nc_def_var       /* define variables: from name, type, ... */
     ...
   nc_put_att       /* put attribute: assign attribute values */
     ...
nc_enddef           /* end definitions: leave define mode */
     ...
   nc_put_var       /* provide values for variables */
     ...
nc_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 nc_def_dim is needed for each dimension created. Similarly, one call to nc_def_var is needed for each variable creation, and one call to a member of the nc_put_att family is needed for each attribute defined and assigned a value. To leave define mode and enter data mode, call nc_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). Single values may be written to a netCDF variable with one of the members of the nc_put_var1 family, depending on what type of data you have to write. All the values of a variable may be written at once with one of the members of the nc_put_var family. Arrays or array cross-sections of a variable may be written using members of the nc_put_vara family. Subsampled array sections may be written using members of the nc_put_vars family. Mapped array sections may be written using members of the nc_put_varm family. (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 nc_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 NC_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 nc_sync or nc_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:

```
nc_open                 /* open existing netCDF dataset */
     ...
   nc_inq_dimid         /* get dimension IDs */
     ...
   nc_inq_varid         /* get variable IDs */
     ...
   nc_get_att           /* get attribute values */
     ...
   nc_get_var           /* get values of variables */
     ...
nc_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 nc_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 nc_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 a member of the nc_get_att family (typically nc_get_att_text or nc_get_att_double) for each desired attribute. Variable data values can be directly accessed from the netCDF dataset with calls to members of the nc_get_var1 family for single values, the nc_get_var family for entire variables, or various other members of the nc_get_vara, nc_get_vars, or nc_get_varm families for array, subsampled or mapped access.

Finally, the netCDF dataset is closed with nc_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:

```
     nc_open                   /* open existing netCDF dataset */
       ...
     nc_inq                    /* find out what is in it */
          ...
        nc_inq_dim             /* get dimension names, lengths */
          ...
        nc_inq_var             /* get variable names, types, shapes */
             ...
           nc_inq_attname      /* get attribute names */
             ...
           nc_inq_att          /* get attribute types and lengths */
             ...
           nc_get_att          /* get attribute values */
             ...
        nc_get_var             /* get values of variables */
          ...
     nc_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 nc_inq 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 0. 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 0, 1, 2, ...up to the number of dimensions. For each dimension ID, a call to the inquire function nc_inq_dim returns the dimension name and length.

Variable IDs are also assigned from consecutive integers 0, 1, 2, ... up to the number of variables. These can be used in nc_inq_var 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 nc_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 nc_inq_att returns its type and length. Given the type and length, you can allocate enough space to hold the attribute values. Then a call to a member of the nc_get_att family returns the attribute values.

Once the IDs and shapes of netCDF variables are known, data values can be accessed by calling a member of the nc_get_var1 family for single values, or members of the nc_get_var, nc_get_vara, nc_get_vars, or nc_get_varm for various kinds of array access.

1.4 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:

```
nc_open              /* open existing netCDF dataset */
  ...
nc_redef             /* put it into define mode */
    ...
  nc_def_dim         /* define additional dimensions (if any) */
    ...
  nc_def_var         /* define additional variables (if any) */
    ...
  nc_put_att         /* define additional attributes (if any) */
    ...
nc_enddef            /* check definitions, leave define mode */
    ...
  nc_put_var         /* provide values for new variables */
    ...
nc_close             /* close netCDF dataset */
```

A netCDF dataset is first opened by the nc_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 nc_redef. In define mode, call nc_def_dim to define new dimensions, nc_def_var to define new variables, and a member of the nc_put_attfamily 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 nc_enddef. If you do not wish to reenter data mode, just call nc_close, which will have the effect of first calling nc_enddef.

Until the nc_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 nc_abort. You may also use the nc_abort call to restore the netCDF dataset to a consistent state if the call to nc_enddef fails. If you have called nc_close from definition mode and the implied call to nc_enddef fails, nc_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 nc_sync function and the NC_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 nc_sync before the next access.

1.5 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, handle_err(), to handle any errors. One possible definition of handle_err() can be found within the documentation of nc_strerror (see Section 2.2 [nc_strerror], page 10).

The nc_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.

1.6 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. Nevertheless, we provide here examples of how to compile and link a program that uses the netCDF library on a Unix platform, so that you can adjust these examples to fit your installation.

Every C file that references netCDF functions or constants must contain an appropriate #include statement before the first such reference:

```
#include <netcdf.h>
```

Unless the netcdf.h file is installed in a standard directory where the C compiler always looks, you must use the -I option when invoking the compiler, to specify a directory where netcdf.h is installed, for example:

```
cc -c -I/usr/local/netcdf/include myprogram.c
```

Alternatively, you could specify an absolute path name in the #include statement, but then your program would not compile on another platform where netCDF is installed in a different location.

Unless the netCDF library is installed in a standard directory where the linker always looks, you must use the -L and -l options to link an object file that uses the netCDF library. For example:

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

Alternatively, you could specify an absolute path name for the library:

```
cc -o myprogram myprogram.o -l/usr/local/netcdf/lib/libnetcdf.a
```

2 Datasets

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 non-negative 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.

2.1 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 C function prototype 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 C 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 2.2 [nc_strerror], page 10.

2.2 Get error message corresponding to error status: nc_strerror

The function nc_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

```
const char * nc_strerror(int ncerr);
```

`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), nc_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 nc_strerror to print the error message corresponding to the netCDF error status returned from any netCDF function call and then exit:

```
#include <netcdf.h>
   ...
void handle_error(int status) {
if (status != NC_NOERR) {
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}
```

2.3 Get netCDF library version: nc_inq_libvers

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

Usage

```
const char * nc_inq_libvers(void);
```

Errors

This function takes no arguments, and thus no errors are possible in its invocation.

Example

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

```
#include <netcdf.h>
   ...
   printf("%s\n", nc_inq_libvers());
```

2.4 Create a NetCDF Dataset: nc_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,
- whether access to the dataset is shared,
- whether this file should be in netCDF classic format (the default), or the new 64-bit offset format.

Usage

```
int nc_create (const char* path, int cmode, int *ncidp);
```

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

`cmode` The creation mode flag. The following flags are available: NC_NOCLOBBER, NC_SHARE, and NC_64BIT_OFFSET.

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

The NC_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 NC_SHARE flag.

Setting NC_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 *The NetCDF Users Guide*.

A zero value (defined for convenience as NC_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 *The NetCDF Users Guide*.

`ncidp` Pointer to location where returned netCDF ID is to be stored.

Errors

nc_create returns the value NC_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 NC_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.

Examples

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:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);
```

In this example we create a netCDF dataset named foo_large.nc. It will be in the 64-bit offset format.

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER | NC_64BIT_OFFSET, &ncid);
if (status != NC_NOERR) handle_error(status);
```

A variant of nc_create, nc__create (note the double underscore) allows users to specify two tuning parameters for the file that it is creating. These tuning parameters are not written to the data file, they are only used for so long as the file remains open after an nc__create.

Usage

```
int nc__create(const char *path, int cmode, size_t initialsz,
               size_t *chunksizehintp, int *ncidp);
```

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

`cmode` The creation mode flag. The following flags are available: NC_NOCLOBBER, NC_SHARE, and NC_64BIT_OFFSET.

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

The NC_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 NC_SHARE flag.

Setting NC_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 *The NetCDF Users Guide*.

A zero value (defined for convenience as NC_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 *The NetCDF Users Guide*.

`initialsz`
: On some systems, and with custom I/O layers, it may be advantageous to set the size of the output file at creation time. This parameter sets the initial size of the file at creation time.

`chunksizehintp`
: The argument referenced by chunksizehintp controls a space versus time tradeoff, 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 by reference.

Using the value NC_SIZEHINT_DEFAULT causes the library to choose a default. How the system chooses the default depends on the system. On many systems, the "preferred I/O block size" is available from the stat() system call, struct stat member st_blksize. If this is available it is used. Lacking that, twice the system pagesize is used.

Lacking a call to discover the system pagesize, we just set default chunksize to 8192.

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

`ncidp`
: Pointer to location where returned netCDF ID is to be stored.

Errors

nc__create returns the value NC_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 NC_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.

Examples

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:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);
```

In this example we create a netCDF dataset named foo_large.nc; we want the dataset to be created in the current directory only if a dataset with that name does not already exist. We also specify that chunksize and initial size for the file.

```
#include <netcdf.h>
   ...
int status;
int ncid;
int intialsz = 2048;
int *chunksize;
   ...
*chunksize = 1024;
status = nc__create("foo.nc", NC_NOCLOBBER, initialsz, chunksize, &ncid);
if (status != NC_NOERR) handle_error(status);
```

2.5 Create a NetCDF Dataset With Performance Options: nc__create

This function is a variant of nc_create, nc__create (note the double underscore) allows users to specify two tuning parameters for the file that it is creating. These tuning parameters are not written to the data file, they are only used for so long as the file remains open after an nc__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, and whether this file should be in netCDF classic format (the default), or the new 64-bit offset format.

Usage

```
int nc__create(const char *path, int cmode, size_t initialsz,
               size_t *chunksizehintp, int *ncidp);
```

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

`cmode` The creation mode flag. The following flags are available: NC_NOCLOBBER, NC_SHARE, and NC_64BIT_OFFSET.

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

The NC_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 NC_SHARE flag.

Setting NC_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 *The NetCDF Users Guide*.

A zero value (defined for convenience as NC_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 *The NetCDF Users Guide*.

`initialsz`
This parameter sets the initial size of the file at creation time.

`chunksizehintp`
The argument referenced by chunksizehintp controls a space versus time tradeoff, 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 by reference.

Using the value NC_SIZEHINT_DEFAULT causes the library to choose a default. How the system chooses the default depends on the system. On many systems, the "preferred I/O block size" is available from the stat() system call, struct stat member st_blksize. If this is available it is used. Lacking that, twice the system pagesize is used.

Lacking a call to discover the system pagesize, we just set default chunksize to 8192.

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

`ncidp` Pointer to location where returned netCDF ID is to be stored.

Errors

nc_create returns the value NC_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 NC_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.

Examples

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. We also specify that chunksize and initial size for the file.

```
#include <netcdf.h>
   ...
int status;
int ncid;
int intialsz = 2048;
int *chunksize;
   ...
*chunksize = 1024;
status = nc__create("foo.nc", NC_NOCLOBBER, initialsz, chunksize, &ncid);
if (status != NC_NOERR) handle_error(status);
```

2.6 Open a NetCDF Dataset for Access: nc_open

The function nc_open opens an existing netCDF dataset for access.

Usage

```
int nc_open (const char *path, int omode, int *ncidp);
```

`path` File name for netCDF dataset to be opened.

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

Otherwise, the creation mode is NC_WRITE, NC_SHARE, or NC_WRITE|NC_SHARE. Setting the NC_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 NC_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 NC_SHARE flag.

`ncidp` Pointer to location where returned netCDF ID is to be stored.

Errors

nc_open returns the value NC_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 nc_open to open an existing netCDF dataset named foo.nc for read-only, non-shared access:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_open("foo.nc", 0, &ncid);
if (status != NC_NOERR) handle_error(status);
```

2.7 Open a NetCDF Dataset for Access with Performance Tuning: nc__open

A function opens a netCDF dataset for access with an additional performance tuning parameter.

Usage

```
int nc__open(const char *path, int mode, size_t *chunksizehintp, int *ncidp);
```

`path` File name for netCDF dataset to be opened.

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

Otherwise, the creation mode is NC_WRITE, NC_SHARE, or NC_WRITE|NC_SHARE. Setting the NC_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 NC_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 NC_SHARE flag.

`chunksizehintp`

The argument referenced by chunksizehintp controls a space versus time tradeoff, 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 by reference.

Using the value NC_SIZEHINT_DEFAULT causes the library to choose a default. How the system chooses the default depends on the system. On many systems, the "preferred I/O block size" is available from the stat() system call, struct stat member st_blksize. If this is available it is used. Lacking that, twice the system pagesize is used.

Lacking a call to discover the system pagesize, we just set default chunksize to 8192.

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

`ncidp` Pointer to location where returned netCDF ID is to be stored.

Errors

nc__open returns the value NC_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 nc__open to open an existing netCDF dataset named foo.nc for read-only, non-shared access:

```
#include <netcdf.h>
   ...
int status;
int ncid;
int *chunksize;
   ...
*chunksize = 1024;
status = nc_open("foo.nc", 0, chunksize, &ncid);
if (status != NC_NOERR) handle_error(status);
```

2.8 Put Open NetCDF Dataset into Define Mode: nc_redef

The function nc_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

```
int nc_redef(int ncid);
```

`ncid` netCDF ID, from a previous call to nc_open or nc_create.

Errors

nc_redef returns the value NC_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 nc_redef to open an existing netCDF dataset named foo.nc and put it into define mode:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);  /* open dataset */
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_redef(ncid);                      /* put in define mode */
if (status != NC_NOERR) handle_error(status);
```

2.9 Leave Define Mode: nc_enddef

The function nc_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 2.15 [nc_set_fill], page 26. 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 *The NetCDF Users Guide*.

Usage

```
int nc_enddef(int ncid);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

Errors

nc_enddef returns the value NC_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 nc_enddef to finish the definitions of a new netCDF dataset named foo.nc and put it into data mode:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);
```

```
    ...       /* create dimensions, variables, attributes */

status = nc_enddef(ncid);  /*leave define mode*/
if (status != NC_NOERR) handle_error(status);
```

2.10 Leave Define Mode with Performance Tuning: nc__enddef

The function nc__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 2.15 [nc_set_fill], page 26. 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 *The NetCDF Users Guide*.

Caution: this function exposes internals of the netcdf version 1 file format. Users should use nc_enddef in most circumstances. This function may not be available on 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 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 nc_redef, nc_enddef by requesting that minfree bytes be available at the end of the section.

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 ALIGN_CHUNK tells the library to use the chunksize (see above) as the align parameter.

The file format requires mod 4 alignment, so the align parameters are silently rounded up to multiples of 4. The usual call,

```
nc_enddef(ncid);
```

is equivalent to

```
nc_enddef(ncid, 0, 4, 0, 4);
```

The file format does not contain a "record size" value, this is calculated from the sizes of the record variables. This unfortunate fact prevents us from providing minfree and alignment control of the "records" in a netcdf file. If you add a variable which has an unlimited dimension, the third section will always be copied with the new variable added.

Usage

```
int nc__enddef(int ncid, size_t h_minfree, size_t v_align,
               size_t v_minfree, size_t r_align);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`h_minfree`
: Sets the pad at the end of the "header" section.

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

`v_minfree`
: Sets the pad at the end of the data section for fixed size variables.

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

Errors

nc__enddef returns the value NC_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 nc_enddef to finish the definitions of a new netCDF dataset named foo.nc and put it into data mode:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);

   ...       /* create dimensions, variables, attributes */

status = nc_enddef(ncid);  /*leave define mode*/
if (status != NC_NOERR) handle_error(status);
```

2.11 Close an Open NetCDF Dataset: nc_close

The function nc_close closes an open netCDF dataset. If the dataset is in define mode, nc_enddef will be called before closing. (In this case, if nc_enddef returns an error, nc_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

```
int nc_close(int ncid);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

Errors

nc_close returns the value NC_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 nc_enddef failed.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

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

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);

   ...       /* create dimensions, variables, attributes */

status = nc_close(ncid);       /* close netCDF dataset */
if (status != NC_NOERR) handle_error(status);
```

2.12 Inquire about an Open NetCDF Dataset: nc_inq Family

Members of the nc_inq family of functions return information about an open netCDF dataset, given its netCDF ID. Dataset inquire functions may be called from either define mode or data mode. The first function, nc_inq, returns values for 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. The other functions in the family each return just one of these items of information.

For C, these functions include nc_inq, nc_inq_ndims, nc_inq_nvars, nc_inq_natts, and nc_inq_unlimdim. An additional function, nc_inq_format, returns the (rarely needed) format version.

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

Usage

```
int nc_inq          (int ncid, int *ndimsp, int *nvarsp, int *ngattsp,
                     int *unlimdimidp);
```

```
int nc_inq_ndims    (int ncid, int *ndimsp);
int nc_inq_nvars    (int ncid, int *nvarsp);
int nc_inq_natts    (int ncid, int *ngattsp);
int nc_inq_unlimdim (int ncid, int *unlimdimidp);
int nc_inq_format   (int ncid, int *formatp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`ndimsp` Pointer to location for returned number of dimensions defined for this netCDF dataset.

`nvarsp` Pointer to location for returned number of variables defined for this netCDF dataset.

`ngattsp` Pointer to location for returned number of global attributes defined for this netCDF dataset.

`unlimdimidp`
Pointer to location for 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.

`formatp` Pointer to location for returned format version, one of NC_FORMAT_CLASSIC, NC_FORMAT_64BIT, NC_FORMAT_NETCDF4, NC_FORMAT_NETCDF4_CLASSIC.

Errors

All members of the nc_inq family return the value NC_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 nc_inq to find out about a netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int status, ncid, ndims, nvars, ngatts, unlimdimid;
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq(ncid, &ndims, &nvars, &ngatts, &unlimdimid);
if (status != NC_NOERR) handle_error(status);
```

2.13 Synchronize an Open NetCDF Dataset to Disk: nc_sync

The function nc_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 nc_sync; to accomplish this, the reading process must call nc_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 nc_sync after writing and the readers call nc_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 NC_SHARE flag, and then it will not be necessary to call nc_sync at all. However, the nc_sync function still provides finer granularity than the NC_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 NC_SHARE flag. Use of the nc_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 nc_sync before any subsequent access.

When calling nc_sync, the netCDF dataset must be in data mode. A netCDF dataset in define mode is synchronized to disk only when nc_enddef is called. A process that is reading a netCDF dataset that another process is writing may call nc_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

```
int nc_sync(int ncid);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

Errors

nc_sync returns the value NC_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 nc_sync to synchronize the disk writes of a netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int status;
int ncid;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);  /* open for writing */
if (status != NC_NOERR) handle_error(status);

   ...           /* write data or change attributes */

status = nc_sync(ncid);      /* synchronize to disk */
if (status != NC_NOERR) handle_error(status);
```

2.14 Back Out of Recent Definitions: nc_abort

You no longer need to call this function, since it is called automatically by nc_close in case the dataset is in define mode and something goes wrong with committing the changes. The function nc_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 nc_redef, the netCDF dataset is restored to its state before definition mode was entered and the dataset is closed.

Usage

```
int nc_abort(int ncid);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

Errors

nc_abort returns the value NC_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 nc_abort to back out of redefinitions of a dataset named foo.nc:

```
#include <netcdf.h>
   ...
```

```
int ncid, status, latid;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);/* open for writing */
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_redef(ncid);                    /* enter define mode */
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_def_dim(ncid, "lat", 18L, &latid);
if (status != NC_NOERR) {
   handle_error(status);
   status = nc_abort(ncid);                 /* define failed, abort */
   if (status != NC_NOERR) handle_error(status);
}
```

2.15 Set Fill Mode for Writes: nc_set_fill

This function is intended for advanced usage, to optimize writes under some circumstances described below. The function nc_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 NC_FILL or NC_NOFILL. The default behavior corresponding to NC_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. For more information on the use of fill values see Section 4.17 [Fill Values], page 66. For information about how to define your own fill values see section "Attribute Conventions" in *NetCDF Users' Guide*.

The behavior corresponding to NC_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 NC_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 nc_set_fill again to explicitly set the fill mode to NC_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 nc_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 nc_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

```
int nc_set_fill (int ncid, int fillmode, int *old_modep);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`fillmode` Desired fill mode for the dataset, either NC_NOFILL or NC_FILL.

`old_modep`
Pointer to location for returned current fill mode of the dataset before this call, either NC_NOFILL or NC_FILL.

Errors

nc_set_fill returns the value NC_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 NC_NOFILL nor NC_FILL..

Example

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

```
#include <netcdf.h>
   ...
int ncid, status, old_fill_mode;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);  /* open for writing */
if (status != NC_NOERR) handle_error(status);

   ...           /* write data with default prefilling behavior */

status = nc_set_fill(ncid, NC_NOFILL, &old_fill_mode); /* set nofill */
if (status != NC_NOERR) handle_error(status);

   ...           /* write data with no prefilling */
```

2.16 Set Default Creation Format: nc_set_default_format

This function is intended for advanced users.

Starting in version 3.6, netCDF introduced a new data format, the first change in the underlying binary data format since the netCDF interface was released. The new format, 64-bit offset format, was introduced to greatly relax the limitations on creating very large files.

Users are warned that creating files in the 64-bit offset format makes them unreadable by the netCDF library prior to version 3.6.0. For reasons of compatibility, users should continue to create files in netCDF classic format.

Users who do want to use 64-bit offset format files can create them directory from nc_create, using the proper cmode flag. (see Section 2.4 [nc_create], page 11).

The function nc_set_default_format allows the user to change the format of the netCDF file to be created by future calls to nc_create (or nc__create) without changing the cmode flag.

This allows the user to convert a program to use 64-bit offset formation without changing all calls the nc_create. See section "Large File Support" in *The NetCDF Users Guide*.

Once the default format is set, all future created files will be in the desired format.

Two constants are provided in the netcdf.h file to be used with this function, NC_FORMAT_64BIT and NC_FORMAT_CLASSIC.

If a non-NULL pointer is provided, it is assumed to point to an int, where the existing default format will be written.

Using nc_create with a cmode including NC_64BIT_OFFSET overrides the default format, and creates a 64-bit offset file.

Usage

```
int nc_set_default_format(int format, int *old_formatp);
```

`format` Either NC_FORMAT_CLASSIC (the default setting) or NC_FORMAT_64BIT.

`old_formatp`
Either NULL (in which case it will be ignored), or a pointer to an int where the existing default format (i.e. before being changed to the new format) will be written. This allows you to get the existing default format while setting a new default format.

Errors

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

- Invalid format. The only valid formats are NC_FORMAT_CLASSIC and NC_FORMAT_64BIT. Trying to set the default format to something else will result in an invalid argument error. (NC_EINVAL)

Example

Here is an example using nc_set_default_format to create the same file in both formats with the same nc_create call:

```
#include <netcdf.h>
   ...
int ncid, status, old_fill_mode;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);  /* open for writing */
if (status != NC_NOERR) handle_error(status);

   ...           /* write data with default prefilling behavior */

status = nc_set_fill(ncid, NC_NOFILL, &old_fill_mode); /* set nofill */
if (status != NC_NOERR) handle_error(status);

   ...           /* write data with no prefilling */
```

3 Dimensions

3.1 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 (1024) to the number of dimensions that can be defined in a single netCDF dataset. The limit is the value of the predefined macro NC_MAX_DIMS. The purpose of the limit is to make writing generic applications simpler. They need only provide an array of NC_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.

Dimension lengths in the C interface are type size_t rather than type int to make it possible to access all the data in a netCDF dataset on a platform that only supports a 16-bit int data type, for example MSDOS. If dimension lengths were type int instead, it would not be possible to access data from variables with a dimension length greater than a 16-bit int can accommodate.

A netCDF dimension in an open netCDF dataset is referred to by a small integer called a dimension ID. In the C interface, dimension IDs are 0, 1, 2, ..., 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.

3.2 Create a Dimension: nc_def_dim

The function nc_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

```
int nc_def_dim (int ncid, const char *name, size_t len, int *dimidp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`name` Dimension name. Must be a legal netCDF identifier. A legal identifier is any sequence of one or more alphabetic characters, digits, and the following special characters: '_', '.', '-', '@', and '+'. The identifier must, however, start with an alphabetic character or underscore. Case is significant and names commencing with underscore are reserved for system use.

`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 (of type size_t) or the predefined constant NC_UNLIMITED.

`dimidp` Pointer to location for returned dimension ID.

Errors

nc_def_dim returns the value NC_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 nc_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:

```
#include <netcdf.h>
   ...
int status, ncid, latid, recid;
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_def_dim(ncid, "lat", 18L, &latid);
if (status != NC_NOERR) handle_error(status);
status = nc_def_dim(ncid, "rec", NC_UNLIMITED, &recid);
if (status != NC_NOERR) handle_error(status);
```

3.3 Get a Dimension ID from Its Name: nc_inq_dimid

The function nc_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 0 and ndims-1.

Usage

int nc_inq_dimid (int ncid, const char *name, int *dimidp);

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`name` Dimension name, a character string beginning with a letter and followed by any sequence of letters, digits, or underscore ('_') characters. Case is significant in dimension names.

`dimidp` Pointer to location for the returned dimension ID.

Errors

nc_inq_dimid returns the value NC_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 nc_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:

```
#include <netcdf.h>
   ...
int status, ncid, latid;
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);  /* open for reading */
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_dimid(ncid, "lat", &latid);
if (status != NC_NOERR) handle_error(status);
```

3.4 Inquire about a Dimension: nc_inq_dim Family

This family of functions returns 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.

The functions in this family include nc_inq_dim, nc_inq_dimname, and nc_inq_dimlen. The function nc_inq_dim returns all the information about a dimension; the other functions each return just one item of information.

Usage

```
int nc_inq_dim     (int ncid, int dimid, char* name, size_t* lengthp);
int nc_inq_dimname (int ncid, int dimid, char *name);
int nc_inq_dimlen  (int ncid, int dimid, size_t *lengthp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`dimid` Dimension ID, from a previous call to nc_inq_dimid or nc_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 NC_MAX_NAME. (This doesn't include the null terminator, so declare your array to be size NC_MAX_NAME+1). The returned character array will be null-terminated.

`lengthp` Pointer to location for returned length of dimension. For the unlimited dimension, this is the number of records written so far.

Errors

These functions return the value NC_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 nc_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:

```
#include <netcdf.h>
   ...
int status, ncid, latid, recid;
size_t latlength, recs;
char recname[NC_MAX_NAME+1];
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);  /* open for reading */
if (status != NC_NOERR) handle_error(status);
status = nc_inq_unlimdim(ncid, &recid); /* get ID of unlimited dimension */
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_dimid(ncid, "lat", &latid);  /* get ID for lat dimension */
if (status != NC_NOERR) handle_error(status);
status = nc_inq_dimlen(ncid, latid, &latlength); /* get lat length */
if (status != NC_NOERR) handle_error(status);
/* get unlimited dimension name and current length */
status = nc_inq_dim(ncid, recid, recname, &recs);
if (status != NC_NOERR) handle_error(status);
```

3.5 Rename a Dimension: nc_rename_dim

The function nc_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

int nc_rename_dim(int ncid, int dimid, const char* name);

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`dimid` Dimension ID, from a previous call to nc_inq_dimid or nc_def_dim.

`name` New dimension name.

Errors

nc_rename_dim returns the value NC_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 nc_rename_dim to rename the dimension lat to latitude in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int status, ncid, latid;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);  /* open for writing */
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_redef(ncid);  /* put in define mode to rename dimension */
if (status != NC_NOERR) handle_error(status);
status = nc_inq_dimid(ncid, "lat", &latid);
if (status != NC_NOERR) handle_error(status);
status = nc_rename_dim(ncid, latid, "latitude");
if (status != NC_NOERR) handle_error(status);
status = nc_enddef(ncid); /* leave define mode */
if (status != NC_NOERR) handle_error(status);
```

4 Variables

4.1 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 0, 1, 2,..., 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 5 [Attributes], page 69) 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.

4.2 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 C interface:

Type	C #define	Bits
byte	NC_BYTE	8

char	NC_CHAR	8
short	NC_SHORT	16
int	NC_INT	32
float	NC_FLOAT	32
double	NC_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 C preprocessor macro for use in netCDF functions (the preprocessor macros are defined in the netCDF C header-file netcdf.h). 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.

4.3 Create a Variable: `nc_def_var`

The function nc_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.

Usage

```
int nc_def_var (int ncid, const char *name, nc_type xtype,
                int ndims, const int dimids[], int *varidp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`name` Variable name. Must be a legal netCDF identifier. A legal identifier is any sequence of one or more alphabetic characters, digits, and the following special characters: '_', '.', '-', '@', and '+'. The identifier must, however, start with an alphabetic character or underscore. Case is significant and names commencing with underscore are reserved for system use.

`xtype` One of the set of predefined netCDF external data types. The type of this parameter, nc_type, is defined in the netCDF header file. The valid netCDF external data types are NC_BYTE, NC_CHAR, NC_SHORT, NC_INT, NC_FLOAT, and NC_DOUBLE.

`ndims` Number of dimensions for the variable. For example, 2 specifies a matrix, 1 specifies a vector, and 0 means the variable is a scalar with no dimensions. Must not be negative or greater than the predefined constant NC_MAX_VAR_DIMS.

`dimids` Vector of ndims dimension IDs corresponding to the variable dimensions. If the ID of the unlimited dimension is included, it must be first. This argument is ignored if ndims is 0.

`varidp` Pointer to location for the returned variable ID.

Errors

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

- The netCDF dataset is not in define mode.
- The specified variable name is the name of another existing variable.
- The specified type is not a valid netCDF type.
- The specified number of dimensions is negative or more than the constant NC_MAX_VAR_DIMS, the maximum number of dimensions permitted for a netCDF variable.
- One or more of the dimension IDs in the list of dimensions is not a valid dimension ID for the netCDF dataset.
- The number of variables would exceed the constant NC_MAX_VARS, the maximum number of variables permitted in a netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using nc_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:

```
#include <netcdf.h>
   ...
int  status;                       /* error status */
int  ncid;                         /* netCDF ID */
int  lat_dim, lon_dim, time_dim;   /* dimension IDs */
int  rh_id;                        /* variable ID */
int  rh_dimids[3];                 /* variable shape */
   ...
status = nc_create("foo.nc", NC_NOCLOBBER, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
                                   /* define dimensions */
status = nc_def_dim(ncid, "lat", 5L, &lat_dim);
if (status != NC_NOERR) handle_error(status);
status = nc_def_dim(ncid, "lon", 10L, &lon_dim);
if (status != NC_NOERR) handle_error(status);
status = nc_def_dim(ncid, "time", NC_UNLIMITED, &time_dim);
if (status != NC_NOERR) handle_error(status);
   ...
                                   /* define variable */
rh_dimids[0] = time_dim;
rh_dimids[1] = lat_dim;
rh_dimids[2] = lon_dim;
status = nc_def_var (ncid, "rh", NC_DOUBLE, 3, rh_dimids, &rh_id);
if (status != NC_NOERR) handle_error(status);
```

4.4 Get a Variable ID from Its Name: nc_inq_varid

The function nc_inq_varid returns the ID of a netCDF variable, given its name.

Usage

```
int nc_inq_varid (int ncid, const char *name, int *varidp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`name` Variable name for which ID is desired.

`varidp` Pointer to location for returned variable ID.

Errors

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

- The specified variable name is not a valid name for a variable in the specified netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

Here is an example using nc_inq_varid to find out the ID of a variable named rh in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int  status, ncid, rh_id;
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
```

4.5 Get Information about a Variable from Its ID: nc_inq_var

family

A family of functions that 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.

The function nc_inq_var returns all the information about a netCDF variable, given its ID. The other functions each return just one item of information about a variable.

These other functions include nc_inq_varname, nc_inq_vartype, nc_inq_varndims, nc_inq_vardimid, and nc_inq_varnatts.

Usage

```
int nc_inq_var      (int ncid, int varid, char *name, nc_type *xtypep,
                     int *ndimsp, int dimids[], int *nattsp);
int nc_inq_varname  (int ncid, int varid, char *name);
int nc_inq_vartype  (int ncid, int varid, nc_type *xtypep);
int nc_inq_varndims (int ncid, int varid, int *ndimsp);
int nc_inq_vardimid (int ncid, int varid, int dimids[]);
int nc_inq_varnatts (int ncid, int varid, int *nattsp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_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 NC_MAX_NAME. (This doesn't include the null terminator, so declare your array to be size NC_MAX_NAME+1). The returned character array will be null-terminated.

`xtypep` Pointer to location for returned variable type, one of the set of predefined netCDF external data types. The type of this parameter, nc_type, is defined in the netCDF header file. The valid netCDF external data types are NC_BYTE, NC_CHAR, NC_SHORT, NC_INT, NC_FLOAT, and NC_DOUBLE.

`ndimsp` Pointer to location for 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 NC_MAX_VAR_DIMS.

`nattsp` Pointer to location for returned number of variable attributes assigned to this variable.

These functions return the value NC_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 nc_inq_var to find out about a variable named rh in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int  status                         /* error status */
int  ncid;                          /* netCDF ID */
int  rh_id;                         /* variable ID */
nc_type rh_type;                    /* variable type */
```

```
int rh_ndims;                      /* number of dims */
int rh_dimids[NC_MAX_VAR_DIMS];    /* dimension ids */
int rh_natts                       /* number of attributes */
   ...
status = nc_open ("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
/* we don't need name, since we already know it */
status = nc_inq_var (ncid, rh_id, 0, &rh_type, &rh_ndims, rh_dimids,
                    &rh_natts);
if (status != NC_NOERR) handle_error(status);
```

4.6 Write a Single Data Value: nc_put_var1_ *type*

The functions nc_put_var1_ *type* put a single data value of the specified type into a variable of an open netCDF dataset that is in data mode. Inputs are the netCDF ID, the variable ID, an index that specifies which value to add or alter, and the data value. The value is converted to the external data type of the variable, if necessary.

Usage

```
int nc_put_var1_text  (int ncid, int varid, const size_t index[],
                       const char *tp);
int nc_put_var1_uchar (int ncid, int varid, const size_t index[],
                       const unsigned char *up);
int nc_put_var1_schar (int ncid, int varid, const size_t index[],
                       const signed char *cp);
int nc_put_var1_short (int ncid, int varid, const size_t index[],
                       const short *sp);
int nc_put_var1_int   (int ncid, int varid, const size_t index[],
                       const int *ip);
int nc_put_var1_long  (int ncid, int varid, const size_t index[],
                       const long *lp);
int nc_put_var1_float (int ncid, int varid, const size_t index[],
                       const float *fp);
int nc_put_var1_double(int ncid, int varid, const size_t index[],
                       const double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`index[]` The index of the data value to be written. The indices are relative to 0, so for example, the first data value of a two-dimensional variable would have index (0,0). The elements of index must correspond to the variable's dimensions. Hence, if the variable uses the unlimited dimension, the first index would correspond to the unlimited dimension.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the data value to be written. If the type of data values differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_put_var1_ *type* returns the value NC_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 nc_put_var1_double to set the (1,2,3) 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 time, lat, and lon, so we want to set the value of rh that corresponds to the second time value, the third lat value, and the fourth lon value:

```
#include <netcdf.h>
   ...
int  status;                          /* error status */
int  ncid;                            /* netCDF ID */
int  rh_id;                           /* variable ID */
static size_t rh_index[] = {1, 2, 3}; /* where to put value */
static double rh_val = 0.5;           /* value to put */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_put_var1_double(ncid, rh_id, rh_index, &rh_val);
if (status != NC_NOERR) handle_error(status);
```

4.7 Write an Entire Variable: nc_put_var_ *type*

The nc_put_var_ *type* family of functions write all the values of a variable into a netCDF variable of an open netCDF dataset. This is the simplest interface to use for writing a value in a scalar variable or whenever all the values of a multidimensional variable can all be written at once. The values to be written are associated with the netCDF variable by assuming that the last dimension of the netCDF variable varies fastest in the C interface. The values are converted to the external data type of the variable, if necessary.

Take care when using the simplest forms of this interface with record variables 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 of a record variable but there are more records in the file than you assume, more data may be written to the file than you supply, which may result in a segmentation violation.

Usage

```
int nc_put_var_text  (int ncid, int varid, const char *tp);
int nc_put_var_uchar (int ncid, int varid, const unsigned char *up);
int nc_put_var_schar (int ncid, int varid, const signed char *cp);
int nc_put_var_short (int ncid, int varid, const short *sp);
int nc_put_var_int   (int ncid, int varid, const int *ip);
int nc_put_var_long  (int ncid, int varid, const long *lp);
int nc_put_var_float (int ncid, int varid, const float *fp);
int nc_put_var_double(int ncid, int varid, const double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to a block of data values to be written. The order in which the data will be written to the netCDF variable is with the last dimension of the specified variable varying fastest. If the type of data values differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

Members of the nc_put_var_ *type* family return the value NC_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.
- One or more of the specified values are out of the range of values representable by the external data type of the variable.

- One or more of the specified values are out of the range of values representable by the external data type of the variable.
- The specified netCDF dataset 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 nc_put_var_double to add or change all the values 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 time, lat, and lon, and that there are three time values, five lat values, and ten lon values.

```
#include <netcdf.h>
   ...
#define TIMES 3
#define LATS  5
#define LONS  10
int  status;                       /* error status */
int  ncid;                         /* netCDF ID */
int  rh_id;                        /* variable ID */
double rh_vals[TIMES*LATS*LONS];   /* array to hold values */
int i;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
for (i = 0; i < TIMES*LATS*LONS; i++)
    rh_vals[i] = 0.5;
/* write values into netCDF variable */
status = nc_put_var_double(ncid, rh_id, rh_vals);
if (status != NC_NOERR) handle_error(status);
```

4.8 Write an Array of Values: nc_put_vara_ *type*

The function nc_put_vara_ *type* writes values into a netCDF variable of an open netCDF dataset. The part of the netCDF variable to write is specified by giving a corner and a vector of edge lengths that refer to an array section of the netCDF variable. The values to be written are associated with the netCDF variable by assuming that the last dimension of the netCDF variable varies fastest in the C interface. The netCDF dataset must be in data mode.

Usage

```
int nc_put_vara_ type   (int ncid, int varid, const size_t start[],
                       const size_t count[], const type *valuesp);
int nc_put_vara_text  (int ncid, int varid, const size_t start[],
```

```
                             const size_t count[], const char *tp);
     int nc_put_vara_uchar (int ncid, int varid, const size_t start[],
                             const size_t count[], const unsigned char *up);
     int nc_put_vara_schar (int ncid, int varid, const size_t start[],
                             const size_t count[], const signed char *cp);
     int nc_put_vara_short (int ncid, int varid, const size_t start[],
                             const size_t count[], const short *sp);
     int nc_put_vara_int   (int ncid, int varid, const size_t start[],
                             const size_t count[], const int *ip);
     int nc_put_vara_long  (int ncid, int varid, const size_t start[],
                             const size_t count[], const long *lp);
     int nc_put_vara_float (int ncid, int varid, const size_t start[],
                             const size_t count[], const float *fp);
     int nc_put_vara_double(int ncid, int varid, const size_t start[],
                             const size_t count[], const double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`start` A vector of size_t integers specifying the index in the variable where the first of the data values will be written. The indices are relative to 0, so for example, the first data value of a variable would have index (0, 0, ... , 0). The size of start must be the same as the number of dimensions of the specified variable. The elements of start must correspond to the variable's dimensions in order. Hence, if the variable is a record variable, the first index would correspond to the starting record number for writing the data values.

`count` A vector of size_t integers specifying the edge lengths along each dimension of the block of data values to be written. To write a single value, for example, specify count as (1, 1, ... , 1). The length of count is the number of dimensions of the specified variable. The elements of count correspond to the variable's dimensions. Hence, if the variable is a record variable, the first element of count corresponds to a count of the number of records to write.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to a block of data values to be written. The order in which the data will be written to the netCDF variable is with the last dimension of the specified variable varying fastest. If the type of data values differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_put_vara_ *type* returns the value NC_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 corner 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 edge lengths added to the specified corner would have referenced data out of range for the rank of the specified variable. For example, an edge length that is larger than the corresponding dimension length minus the corner index will cause an error.
- One or more of the specified values are out of the range of values representable by the external data type of the variable.
- The specified netCDF dataset 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 nc_put_vara_double to add or change all the values 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 time, lat, and lon, and that there are three time values, five lat values, and ten lon values.

```
#include <netcdf.h>
   ...
#define TIMES 3
#define LATS  5
#define LONS  10
int  status;                       /* error status */
int  ncid;                         /* netCDF ID */
int  rh_id;                        /* variable ID */
static size_t start[] = {0, 0, 0}; /* start at first value */
static size_t count[] = {TIMES, LATS, LONS};
double rh_vals[TIMES*LATS*LONS];   /* array to hold values */
int i;
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
for (i = 0; i < TIMES*LATS*LONS; i++)
    rh_vals[i] = 0.5;
/* write values into netCDF variable */
status = nc_put_vara_double(ncid, rh_id, start, count, rh_vals);
if (status != NC_NOERR) handle_error(status);
```

4.9 Write a Subsampled Array of Values: nc_put_vars_ *type*

Each member of the family of functions nc_put_vars_ *type* writes a subsampled (strided) array section of values into a netCDF variable of an open netCDF dataset. The subsampled array section is specified by giving a corner, a vector of counts, and a stride vector. The netCDF dataset must be in data mode.

Usage

```
int nc_put_vars_text  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const char *tp);
int nc_put_vars_uchar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const unsigned char *up);
int nc_put_vars_schar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const signed char *cp);
int nc_put_vars_short (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const short *sp);
int nc_put_vars_int   (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const int *ip);
int nc_put_vars_long  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const long *lp);
int nc_put_vars_float (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const float *fp);
int nc_put_vars_double(int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`start` A vector of size_t integers specifying the index in the variable where the first of the data values will be written. The indices are relative to 0, so for example, the first data value of a variable would have index (0, 0, ... , 0). The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the first index corresponds to the starting record number for writing the data values.

`count` A vector of size_t 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 first element of count corresponds to a count of the number of records to write.

`stride` A vector of ptrdiff_t 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[0] gives the sampling interval along the most slowly 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.). A NULL stride argument is treated as (1, 1, ... , 1).

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to a block of data values to be written. The order in which the data will be written to the netCDF variable is with the last dimension of the specified variable varying fastest. If the type of data values differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_put_vars_ *type* returns the value NC_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 start, count and stride generate an index which is out of range.
- One or more of the specified values are 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 of using nc_put_vars_float to write – from an internal array – every other point of a netCDF variable named rh which is described by the C declaration float rh[4][6] (note the size of the dimensions):

```
#include <netcdf.h>
   ...
#define NDIM 2                /* rank of netCDF variable */
int ncid;                     /* netCDF ID */
int status;                   /* error status */
int rhid;                     /* variable ID */
static size_t start[NDIM]     /* netCDF variable start point: */
                 = {0, 0};    /* first element */
static size_t count[NDIM]     /* size of internal array: entire */
```

```
                  = {2, 3};  /* (subsampled) netCDF variable */
static ptrdiff_t stride[NDIM] /* variable subsampling intervals: */
                  = {2, 2};    /* access every other netCDF element */
float rh[2][3];               /* note subsampled sizes for */
                              /* netCDF variable dimensions */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid(ncid, "rh", &rhid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_put_vars_float(ncid, rhid, start, count, stride, rh);
if (status != NC_NOERR) handle_error(status);
```

4.10 Write a Mapped Array of Values: nc_put_varm_ *type*

The nc_put_varm_ *type* family of functions writes a mapped array section of values into a netCDF variable of an open netCDF dataset. The mapped array section is specified by giving a corner, a vector of counts, a stride vector, and an index mapping vector. The index mapping vector is 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. No assumptions are made about the ordering or length of the dimensions of the data array. The netCDF dataset must be in data mode.

Usage

```
int nc_put_varm_text  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const char *tp);
int nc_put_varm_uchar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const unsigned char *up);
int nc_put_varm_schar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const signed char *cp);
int nc_put_varm_short (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const short *sp);
int nc_put_varm_int   (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const int *ip);
int nc_put_varm_long  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const long *lp);
int nc_put_varm_float (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const float *fp);
```

```
int nc_put_varm_double(int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], const double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`start` A vector of size_t integers specifying the index in the variable where the first of the data values will be written. The indices are relative to 0, so for example, the first data value of a variable would have index (0, 0, ... , 0). The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the first index corresponds to the starting record number for writing the data values.

`count` A vector of size_t 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 first element of count corresponds to a count of the number of records to write.

`stride` A vector of ptrdiff_t 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[0] gives the sampling interval along the most slowly 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.). A NULL stride argument is treated as (1, 1, ... , 1).

`imap` A vector of ptrdiff_t 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 (imap[0] gives the distance between elements of the internal array corresponding to the most slowly varying dimension of the netCDF variable). Distances between elements are specified in type-independent units of elements (the distance between internal elements that occupy adjacent memory locations is 1 and not the element's byte-length as in netCDF 2). A NULL argument means the memory-resident values have the same structure as the associated netCDF variable.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the location used for computing where the data values will be found; the data should be of the type appropriate for the function called. If the type of

data values differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_put_varm_ *type* returns the value NC_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 start, count, and stride generate an index which is out of range. Note that no error checking is possible on the imap vector.
- One or more of the specified values are 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

The following imap vector maps in the trivial way a 4x3x2 netCDF variable and an internal array of the same shape:

```
float a[4][3][2];       /* same shape as netCDF variable */
int   imap[3] = {6, 2, 1};
                        /* netCDF dimension       inter-element distance */
                        /* ----------------       ---------------------- */
                        /* most rapidly varying       1                  */
                        /* intermediate               2 (=imap[2]*2)     */
                        /* most slowly varying        6 (=imap[1]*3)     */
```

Using the imap vector above with nc_put_varm_float obtains the same result as simply using nc_put_var_float.

Here is an example of using nc_put_varm_float to write – from a transposed, internal array – a netCDF variable named rh which is described by the C declaration float rh[6][4] (note the size and order of the dimensions):

```
#include <netcdf.h>
   ...
#define NDIM 2                /* rank of netCDF variable */
int ncid;                     /* netCDF ID */
int status;                   /* error status */
int rhid;                     /* variable ID */
static size_t start[NDIM]     /* netCDF variable start point: */
                 = {0, 0};    /* first element */
static size_t count[NDIM]     /* size of internal array: entire netCDF */
                 = {6, 4};    /* variable; order corresponds to netCDF */
                              /* variable -- not internal array */
static ptrdiff_t stride[NDIM]/* variable subsampling intervals: */
                 = {1, 1};    /* sample every netCDF element */
static ptrdiff_t imap[NDIM]   /* internal array inter-element distances; */
                 = {1, 6};    /* would be {4, 1} if not transposing */
```

```
float rh[4][6];              /* note transposition of netCDF variable */
                             /* dimensions */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid(ncid, "rh", &rhid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_put_varm_float(ncid, rhid, start, count, stride, imap, rh);
if (status != NC_NOERR) handle_error(status);
```

Here is another example of using nc_put_varm_float to write – from a transposed, internal array – a subsample of the same netCDF variable, by writing every other point of the netCDF variable:

```
#include <netcdf.h>
   ...
#define NDIM 2                  /* rank of netCDF variable */
int ncid;                       /* netCDF ID */
int status;                     /* error status */
int rhid;                       /* variable ID */
static size_t start[NDIM]       /* netCDF variable start point: */
                 = {0, 0};      /* first element */
static size_t count[NDIM]       /* size of internal array: entire */
                   = {3, 2};    /* (subsampled) netCDF variable; order of */
                                /* dimensions corresponds to netCDF */
                                /* variable -- not internal array */
static ptrdiff_t stride[NDIM]   /* variable subsampling intervals: */
                 = {2, 2};      /* sample every other netCDF element */
static ptrdiff_t imap[NDIM]     /* internal array inter-element distances; */
                 = {1, 3};      /* would be {2, 1} if not transposing */
float rh[2][3];                 /* note transposition of (subsampled) */
                                /* netCDF variable dimensions */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid(ncid, "rh", &rhid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_put_varm_float(ncid, rhid, start, count, stride, imap, rh);
if (status != NC_NOERR) handle_error(status);
```

4.11 Read a Single Data Value: nc_get_var1_ *type*

The functions nc_get_var1_ *type* get a single data value from a variable of an open netCDF dataset that is in data mode. Inputs are the netCDF ID, the variable ID, a multidimensional index that specifies which value to get, and the address of a location into which the data value will be read. The value is converted from the external data type of the variable, if necessary.

Usage

```
int nc_get_var1_text  (int ncid, int varid, const size_t index[],
                        char *tp);
int nc_get_var1_uchar (int ncid, int varid, const size_t index[],
                        unsigned char *up);
int nc_get_var1_schar (int ncid, int varid, const size_t index[],
                        signed char *cp);
int nc_get_var1_short (int ncid, int varid, const size_t index[],
                        short *sp);
int nc_get_var1_int   (int ncid, int varid, const size_t index[],
                        int *ip);
int nc_get_var1_long  (int ncid, int varid, const size_t index[],
                        long *lp);
int nc_get_var1_float (int ncid, int varid, const size_t index[],
                        float *fp);
int nc_get_var1_double(int ncid, int varid, const size_t index[],
                        double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`index[]` The index of the data value to be read. The indices are relative to 0, so for example, the first data value of a two-dimensional variable would have index (0,0). The elements of index must correspond to the variable's dimensions. Hence, if the variable is a record variable, the first index is the record number.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the location into which the data value is read. If the type of data value differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_get_var1_ *type* returns the value NC_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 value is out of the range of values representable by the desired data type.
- 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 nc_get_var1_double to get the (1,2,3) element of the variable named rh in an existing netCDF dataset named foo.nc. For simplicity in this example, we assume that we know that rh is dimensioned with time, lat, and lon, so we want to get the value of rh that corresponds to the second time value, the third lat value, and the fourth lon value:

```
#include <netcdf.h>
   ...
int  status;                           /* error status */
int ncid;                              /* netCDF ID */
int rh_id;                             /* variable ID */
static size_t rh_index[] = {1, 2, 3};  /* where to get value from */
double rh_val;                         /* where to put it */
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_get_var1_double(ncid, rh_id, rh_index, &rh_val);
if (status != NC_NOERR) handle_error(status);
```

4.12 Read an Entire Variable nc_get_var_ *type*

The members of the nc_get_var_ *type* family of functions read all the values from a netCDF variable of an open netCDF dataset. This is the simplest interface to use for reading the value of a scalar variable or when all the values of a multidimensional variable can be read at once. The values are read into consecutive locations with the last dimension varying fastest. The netCDF dataset must be in data mode.

Take care when using the simplest forms of this interface with record variables 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.

Usage

```
int nc_get_var_text  (int ncid, int varid, char *tp);
int nc_get_var_uchar (int ncid, int varid, unsigned char *up);
int nc_get_var_schar (int ncid, int varid, signed char *cp);
int nc_get_var_short (int ncid, int varid, short *sp);
int nc_get_var_int   (int ncid, int varid, int *ip);
```

```
int nc_get_var_long  (int ncid, int varid, long *lp);
int nc_get_var_float (int ncid, int varid, float *fp);
int nc_get_var_double(int ncid, int varid, double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the location into which the data value is read. If the type of data value differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_get_var_ *type* returns the value NC_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.
- One or more of the 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.

Example

Here is an example using nc_get_var_double to read all the values 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 time, lat, and lon, and that there are three time values, five lat values, and ten lon values.

```
#include <netcdf.h>
   ...
#define TIMES 3
#define LATS 5
#define LONS 10
int  status;                      /* error status */
int ncid;                         /* netCDF ID */
int rh_id;                        /* variable ID */
double rh_vals[TIMES*LATS*LONS];  /* array to hold values */
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
```

```
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
/* read values from netCDF variable */
status = nc_get_var_double(ncid, rh_id, rh_vals);
if (status != NC_NOERR) handle_error(status);
```

4.13 Read an Array of Values: nc_get_vara_ *type*

The members of the nc_get_vara_ *type* family of functions read an array of values from a netCDF variable of an open netCDF dataset. The array is specified by giving a corner and a vector of edge lengths. The values are read into consecutive locations with the last dimension varying fastest. The netCDF dataset must be in data mode.

Usage

```
int nc_get_vara_text  (int ncid, int varid, const size_t start[],
                       const size_t count[] char *tp);
int nc_get_vara_uchar (int ncid, int varid, const size_t start[],
                       const size_t count[] unsigned char *up);
int nc_get_vara_schar (int ncid, int varid, const size_t start[],
                       const size_t count[] signed char *cp);
int nc_get_vara_short (int ncid, int varid, const size_t start[],
                       const size_t count[] short *sp);
int nc_get_vara_int   (int ncid, int varid, const size_t start[],
                       const size_t count[] int *ip);
int nc_get_vara_long  (int ncid, int varid, const size_t start[],
                       const size_t count[] long *lp);
int nc_get_vara_float (int ncid, int varid, const size_t start[],
                       const size_t count[] float *fp);
int nc_get_vara_double(int ncid, int varid, const size_t start[],
                       const size_t count[] double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`start` A vector of size_t integers specifying the index in the variable where the first of the data values will be read. The indices are relative to 0, so for example, the first data value of a variable would have index (0, 0, ... , 0). The length of start must be the same as the number of dimensions of the specified variable. The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the first index would correspond to the starting record number for reading the data values.

`count` A vector of size_t integers specifying the edge lengths along each dimension of the block of data values to be read. To read a single value, for example, specify count as (1, 1, ... , 1). The length of count is the number of dimensions of the specified variable. The elements of count correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the first element of count corresponds to a count of the number of records to read.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the location into which the data value is read. If the type of data value differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_get_vara_ *type* returns the value NC_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 corner 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 edge lengths added to the specified corner would have referenced data out of range for the rank of the specified variable. For example, an edge length that is larger than the corresponding dimension length minus the corner index will cause an error.
- One or more of the 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.

Example

Here is an example using nc_get_vara_double to read all the values 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 time, lat, and lon, and that there are three time values, five lat values, and ten lon values.

```
#include <netcdf.h>
   ...
#define TIMES 3
 #define LATS 5
#define LONS 10
int  status;                       /* error status */
int ncid;                          /* netCDF ID */
int rh_id;                         /* variable ID */
static size_t start[] = {0, 0, 0}; /* start at first value */
static size_t count[] = {TIMES, LATS, LONS};
```

```
double rh_vals[TIMES*LATS*LONS];   /* array to hold values */
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
/* read values from netCDF variable */
status = nc_get_vara_double(ncid, rh_id, start, count, rh_vals);
if (status != NC_NOERR) handle_error(status);
```

4.14 Read a Subsampled Array of Values: nc_get_vars_ *type*

The nc_get_vars_ *type* family of functions read a subsampled (strided) array section of values from a netCDF variable of an open netCDF dataset. The subsampled array section is specified by giving a corner, a vector of edge lengths, and a stride vector. The values are read with the last dimension of the netCDF variable varying fastest. The netCDF dataset must be in data mode.

Usage

```
int nc_get_vars_text  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       char *tp);
int nc_get_vars_uchar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       unsigned char *up);
int nc_get_vars_schar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       signed char *cp);
int nc_get_vars_short (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       short *sp);
int nc_get_vars_int   (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       int *ip);
int nc_get_vars_long  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       long *lp);
int nc_get_vars_float (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       float *fp);
int nc_get_vars_double(int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       double *dp)
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`start` A vector of size_t integers specifying the index in the variable where the first of the data values will be read. The indices are relative to 0, so for example, the first data value of a variable would have index (0, 0, ... , 0). The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the first index corresponds to the starting record number for reading the data values.

`count` A vector of size_t 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 first element of count corresponds to a count of the number of records to read.

`stride` A vector of ptrdiff_t integers specifying, for each dimension, the interval between selected indices. The elements of the stride vector correspond, in order, to the variable's dimensions. A value of 1 accesses adjacent values of the netCDF variable in the corresponding dimension; a value of 2 accesses every other value of the netCDF variable in the corresponding dimension; and so on. A NULL stride argument is treated as (1, 1, ... , 1).

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the location into which the data value is read. If the type of data value differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_get_vars_ *type* returns the value NC_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 start, count and stride generate an index which is out of range.
- One or more of the 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.

Example

Here is an example that uses nc_get_vars_double to read every other value in each dimension 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 time, lat, and lon, and that there are three time values, five lat values, and ten lon values.

```
#include <netcdf.h>
   ...
#define TIMES 3
#define LATS  5
#define LONS 10
int  status;                        /* error status */
int ncid;                           /* netCDF ID */
int rh_id;                          /* variable ID */
static size_t start[] = {0, 0, 0};    /* start at first value */
static size_t count[] = {TIMES, LATS, LONS};
static ptrdiff_t stride[] = {2, 2, 2};/* every other value */
double data[TIMES][LATS][LONS];      /* array to hold values */
 ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
 ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
 ...
/* read subsampled values from netCDF variable into array */
status = nc_get_vars_double(ncid, rh_id, start, count, stride,
                            &data[0][0][0]);
if (status != NC_NOERR) handle_error(status);
 ...
```

4.15 Read a Mapped Array of Values: nc_get_varm_ *type*

The nc_get_varm_ *type* family of functions reads a mapped array section of values from a netCDF variable of an open netCDF dataset. The mapped array section is specified by giving a corner, a vector of edge lengths, a stride vector, and an index mapping vector. The index mapping vector is 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. No assumptions are made about the ordering or length of the dimensions of the data array. The netCDF dataset must be in data mode.

Usage

```
int nc_get_varm_text  (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], char *tp);
int nc_get_varm_uchar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], unsigned char *up);
int nc_get_varm_schar (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
                       const ptrdiff_t imap[], signed char *cp);
int nc_get_varm_short (int ncid, int varid, const size_t start[],
                       const size_t count[], const ptrdiff_t stride[],
```

```
                         const ptrdiff_t imap[], short *sp);
int nc_get_varm_int    (int ncid, int varid, const size_t start[],
                         const size_t count[], const ptrdiff_t stride[],
                         const ptrdiff_t imap[], int *ip);
int nc_get_varm_long   (int ncid, int varid, const size_t start[],
                         const size_t count[], const ptrdiff_t stride[],
                         const ptrdiff_t imap[], long *lp);
int nc_get_varm_float  (int ncid, int varid, const size_t start[],
                         const size_t count[], const ptrdiff_t stride[],
                         const ptrdiff_t imap[], float *fp);
int nc_get_varm_double(int ncid, int varid, const size_t start[],
                         const size_t count[], const ptrdiff_t stride[],
                         const ptrdiff_t imap[], double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

`varid` Variable ID.

`start` A vector of size_t integers specifying the index in the variable where the first of the data values will be read. The indices are relative to 0, so for example, the first data value of a variable would have index (0, 0, ... , 0). The elements of start correspond, in order, to the variable's dimensions. Hence, if the variable is a record variable, the first index corresponds to the starting record number for reading the data values.

`count` A vector of size_t 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 first element of count corresponds to a count of the number of records to read.

`stride` A vector of ptrdiff_t integers specifying, for each dimension, the interval between selected indices. The elements of the stride vector correspond, in order, to the variable's dimensions. A value of 1 accesses adjacent values of the netCDF variable in the corresponding dimension; a value of 2 accesses every other value of the netCDF variable in the corresponding dimension; and so on. A NULL stride argument is treated as (1, 1, ... , 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. imap[0] gives the distance between elements of the internal array corresponding to the most slowly varying dimension of the netCDF variable. imap[n-1] (where n is the rank of the netCDF variable) gives the distance between elements of the internal array corresponding to the most rapidly varying dimension of the netCDF variable. Intervening imap elements correspond to other dimensions of the netCDF variable in the obvious way. Distances between elements are specified in type-independent units of elements (the distance between internal elements that occupy adjacent memory locations is 1 and not the element's byte-length as in netCDF 2).

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to the location used for computing where the data values are read; the data should be of the type appropriate for the function called. If the type of data value differs from the netCDF variable type, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_get_varm_ *type* returns the value NC_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 start, count, and stride generate an index which is out of range. Note that no error checking is possible on the imap vector.
- One or more of the 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.

Example

The following imap vector maps in the trivial way a 4x3x2 netCDF variable and an internal array of the same shape:

```
float a[4][3][2];       /* same shape as netCDF variable */
size_t imap[3] = {6, 2, 1};
                        /* netCDF dimension       inter-element distance */
                        /* ----------------       ---------------------- */
                        /* most rapidly varying       1                  */
                        /* intermediate               2 (=imap[2]*2)     */
                        /* most slowly varying        6 (=imap[1]*3)     */
```

Using the imap vector above with nc_get_varm_float obtains the same result as simply using nc_get_var_float.

Here is an example of using nc_get_varm_float to transpose a netCDF variable named rh which is described by the C declaration float rh[6][4] (note the size and order of the dimensions):

```
#include <netcdf.h>
   ...
#define NDIM 2                /* rank of netCDF variable */
int ncid;                     /* netCDF ID */
int status;                   /* error status */
int rhid;                     /* variable ID */
```

```
static size_t start[NDIM]     /* netCDF variable start point: */
                 = {0, 0};    /* first element */
static size_t count[NDIM]     /* size of internal array: entire netCDF */
                 = {6, 4};    /* variable; order corresponds to netCDF */
                              /* variable -- not internal array */
static ptrdiff_t stride[NDIM] /* variable subsampling intervals: */
                 = {1, 1};    /* sample every netCDF element */
static ptrdiff_t imap[NDIM]   /* internal array inter-element distances; */
                 = {1, 6};    /* would be {4, 1} if not transposing */
float rh[4][6];               /* note transposition of netCDF variable */
                              /* dimensions */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid(ncid, "rh", &rhid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_get_varm_float(ncid, rhid, start, count, stride, imap, rh);
if (status != NC_NOERR) handle_error(status);
```

Here is another example of using nc_get_varm_float to simultaneously transpose and subsample the same netCDF variable, by accessing every other point of the netCDF variable:

```
#include <netcdf.h>
   ...
#define NDIM 2               /* rank of netCDF variable */
int ncid;                    /* netCDF ID */
int status;                  /* error status */
int rhid;                    /* variable ID */
static size_t start[NDIM]    /* netCDF variable start point: */
                 = {0, 0};   /* first element */
static size_t count[NDIM]    /* size of internal array: entire */
                   = {3, 2}; /* (subsampled) netCDF variable; order of */
                             /* dimensions corresponds to netCDF */
                             /* variable -- not internal array */
static ptrdiff_t stride[NDIM]/* variable subsampling intervals: */
                 = {2, 2};   /* sample every other netCDF element */
static ptrdiff_t imap[NDIM]  /* internal array inter-element distances; */
                 = {1, 3};   /* would be {2, 1} if not transposing */
float rh[2][3];              /* note transposition of (subsampled) */
                             /* netCDF variable dimensions */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid(ncid, "rh", &rhid);
if (status != NC_NOERR) handle_error(status);
```

```
    ...
status = nc_get_varm_float(ncid, rhid, start, count, stride, imap, rh);
if (status != NC_NOERR) handle_error(status);
```

4.16 Reading and Writing Character String Values

Character strings are not a primitive netCDF external data type, 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 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 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 last dimension for the variable in C). 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 zero for C. 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:(1, 1, ... , 1, n).

In C, fixed-length strings may be written to a netCDF dataset without the terminating zero byte, to save space. Variable-length strings should be written with a terminating zero byte so that the intended length of the string can be determined when it is later read.

Here is an example that defines a record variable, tx, for character strings and stores a character-string value into the third record using nc_put_vara_text. In this example, we assume the string variable and data are to be added to an existing netCDF dataset named foo.nc that already has an unlimited record dimension time.

```
#include <netcdf.h>
    ...
int  ncid;           /* netCDF ID */
int  chid;           /* dimension ID for char positions */
int  timeid;         /* dimension ID for record dimension */
int  tx_id;          /* variable ID */
#define TDIMS 2      /* rank of tx variable */
int tx_dims[TDIMS];  /* variable shape */
```

```
size_t tx_start[TDIMS];
size_t tx_count[TDIMS];
static char tx_val[] =
        "example string"; /* string to be put */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
status = nc_redef(ncid);       /* enter define mode */
if (status != NC_NOERR) handle_error(status);
   ...
/* define character-position dimension for strings of max length 40 */
status = nc_def_dim(ncid, "chid", 40L, &chid);
if (status != NC_NOERR) handle_error(status);
   ...
/* define a character-string variable */
tx_dims[0] = timeid;
tx_dims[1] = chid;    /* character-position dimension last */
status = nc_def_var (ncid, "tx", NC_CHAR, TDIMS, tx_dims, &tx_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_enddef(ncid);       /* leave define mode */
if (status != NC_NOERR) handle_error(status);
   ...
/* write tx_val into tx netCDF variable in record 3 */
tx_start[0] = 3;      /* record number to write */
tx_start[1] = 0;      /* start at beginning of variable */
tx_count[0] = 1;      /* only write one record */
tx_count[1] = strlen(tx_val) + 1;  /* number of chars to write */
status = nc_put_vara_text(ncid, tx_id, tx_start, tx_count, tx_val);
if (status != NC_NOERR) handle_error(status);
```

4.17 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 nc_set_fill before 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. Their are default fill values for each type, defined in the include file netcdf.h: NC_FILL_CHAR, NC_FILL_BYTE, NC_FILL_SHORT, NC_FILL_INT, NC_FILL_FLOAT, and NC_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.

4.18 Rename a Variable: nc_rename_var

The function nc_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

int nc_rename_var(int ncid, int varid, const char* name);

ncid NetCDF ID, from a previous call to nc_open or nc_create.

varid Variable ID.

name New name for the specified variable.

Errors

nc_rename_var returns the value NC_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 nc_rename_var to rename the variable rh to rel_hum in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int  status;             /* error status */
int  ncid;               /* netCDF ID */
int  rh_id;              /* variable ID */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
```

```
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_redef(ncid);  /* put in define mode to rename variable */
if (status != NC_NOERR) handle_error(status);
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
status = nc_rename_var (ncid, rh_id, "rel_hum");
if (status != NC_NOERR) handle_error(status);
status = nc_enddef(ncid); /* leave define mode */
if (status != NC_NOERR) handle_error(status);
```

5 Attributes

5.1 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 nc_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 NC_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.

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.

5.2 Create an Attribute: nc_put_att_ *type*

The function nc_put_att_ *type* adds 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.

```
int nc_put_att_text   (int ncid, int varid, const char *name,
                                    size_t len, const char *tp);
int nc_put_att_uchar  (int ncid, int varid, const char *name,
                          nc_type xtype, size_t len, const unsigned char *up);
int nc_put_att_schar  (int ncid, int varid, const char *name,
                          nc_type xtype, size_t len, const signed char *cp);
```

```
int nc_put_att_short  (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const short *sp);
int nc_put_att_int    (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const int *ip);
int nc_put_att_long   (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const long *lp);
int nc_put_att_float  (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const float *fp);
int nc_put_att_double (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

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

`name` Attribute name. Must be a legal netCDF identifier. A legal identifier is any sequence of one or more alphabetic characters, digits, and the following special characters: '_', '.', '-', '@', and '+'. The identifier must, however, start with an alphabetic character or underscore. Case is significant and names commencing with underscore are reserved for system use.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. For examples of attribute conventions see section "Attribute Conventions" in *The NetCDF Users Guide*.

`xtype` One of the set of predefined netCDF external data types. The type of this parameter, nc_type, is defined in the netCDF header file. The valid netCDF external data types are NC_BYTE, NC_CHAR, NC_SHORT, NC_INT, NC_FLOAT, and NC_DOUBLE. Although it's possible to create attributes of all types, NC_CHAR and NC_DOUBLE attributes are adequate for most purposes.

`len` Number of values provided for the attribute.

`tp, up, cp, sp, ip, lp, fp, or dp`
Pointer to one or more values. If the type of values differs from the netCDF attribute type specified as xtype, type conversion will occur. See section "Type Conversion" in *The NetCDF Users Guide*.

Errors

nc_put_att_ *type* returns the value NC_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 NC_MAX_ATTRS.

Example

Here is an example using nc_put_att_double 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:

```
#include <netcdf.h>
   ...
int  status;                          /* error status */
int  ncid;                            /* netCDF ID */
int  rh_id;                           /* variable ID */
static double rh_range[] = {0.0, 100.0};/* attribute vals */
static char title[] = "example netCDF dataset";
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_redef(ncid);              /* enter define mode */
if (status != NC_NOERR) handle_error(status);
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_put_att_double (ncid, rh_id, "valid_range",
                            NC_DOUBLE, 2, rh_range);
if (status != NC_NOERR) handle_error(status);
status = nc_put_att_text (ncid, NC_GLOBAL, "title",
                          strlen(title), title)
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_enddef(ncid);             /* leave define mode */
if (status != NC_NOERR) handle_error(status);
```

5.3 Get Information about an Attribute: nc_inq_att Family

This family of functions returns information about a netCDF attribute. All but one of these functions require the variable ID and attribute name; the exception is nc_inq_attname. Information about an attribute includes its type, length, name, and number. See the nc_get_att family for getting attribute values.

The function nc_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.

The function nc_inq_att returns the attribute's type and length. The other functions each return just one item of information about an attribute.

Usage

```
int nc_inq_att    (int ncid, int varid, const char *name,
                   nc_type *xtypep, size_t *lenp);
int nc_inq_atttype(int ncid, int varid, const char *name,
                   nc_type *xtypep);
int nc_inq_attlen  (int ncid, int varid, const char *name, size_t *lenp);
int nc_inq_attname(int ncid, int varid, int attnum, char *name);
int nc_inq_attid   (int ncid, int varid, const char *name, int *attnump);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

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

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

`xtypep` Pointer to location for returned attribute type, one of the set of predefined netCDF external data types. The type of this parameter, nc_type, is defined in the netCDF header file. The valid netCDF external data types are NC_BYTE, NC_CHAR, NC_SHORT, NC_INT, NC_FLOAT, and NC_DOUBLE. If this parameter is given as '0' (a null pointer), no type will be returned so no variable to hold the type needs to be declared.

`lenp` Pointer to location for returned number of values currently stored in the attribute. For attributes of type NC_CHAR, you should not assume that this includes a trailing zero byte; it doesn't if the attribute was stored without a trailing zero byte, for example from a FORTRAN program. Before using the value as a C string, make sure it is null-terminated. If this parameter is given as '0' (a null pointer), no length will be returned so no variable to hold this information needs to be declared.

`attnum` For nc_inq_attname, attribute number. The attributes for each variable are numbered from 0 (the first attribute) to natts-1, where natts is the number of attributes for the variable, as returned from a call to nc_inq_varnatts.

`attnump` For nc_inq_attid, pointer to location for returned attribute number that specifies which attribute this is for this variable (or which global attribute). If you already know the attribute name, knowing its number is not very useful, because accessing information about an attribute requires its name.

Errors

Each function returns the value NC_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 nc_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 nc_inq_att to find out the type and length of a variable attribute named valid_range for a netCDF variable named rh and a global attribute named title in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int  status;              /* error status */
int  ncid;                /* netCDF ID */
int  rh_id;               /* variable ID */
nc_type vr_type, t_type;  /* attribute types */
int  vr_len, t_len;       /* attribute lengths */

   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_att (ncid, rh_id, "valid_range", &vr_type, &vr_len);
if (status != NC_NOERR) handle_error(status);
status = nc_inq_att (ncid, NC_GLOBAL, "title", &t_type, &t_len);
if (status != NC_NOERR) handle_error(status);
```

5.4 Get Attribute's Values:nc_get_att_ *type*

Members of the nc_get_att_ *type* family of functions get the value(s) of a netCDF attribute, given its variable ID and name.

Usage

```
int nc_get_att_text   (int ncid, int varid, const char *name,
                       char *tp);
int nc_get_att_uchar  (int ncid, int varid, const char *name,
                       unsigned char *up);
int nc_get_att_schar  (int ncid, int varid, const char *name,
                       signed char *cp);
int nc_get_att_short  (int ncid, int varid, const char *name,
                       short *sp);
int nc_get_att_int    (int ncid, int varid, const char *name,
                       int *ip);
int nc_get_att_long   (int ncid, int varid, const char *name,
                       long *lp);
int nc_get_att_float  (int ncid, int varid, const char *name,
```

```
                              float *fp);
int nc_get_att_double (int ncid, int varid, const char *name,
                              double *dp);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

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

`name` Attribute name.

`tp`
`up`
`cp`
`sp`
`ip`
`lp`
`fp`
`dp` Pointer to location for returned attribute value(s). All elements of the vector of attribute values are returned, so you must allocate enough space to hold them. For attributes of type NC_CHAR, you should not assume that the returned values include a trailing zero byte; they won't if the attribute was stored without a trailing zero byte, for example from a FORTRAN program. Before using the value as a C string, make sure it is null-terminated. If you don't know how much space to reserve, call nc_inq_attlen first to find out the length of the attribute.

Errors

nc_get_att_ *type* returns the value NC_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 nc_get_att_double to determine the values of a variable 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, but that we do know the types of the attributes. Hence, to allocate enough space to store them, we must first inquire about the length of the attributes.

```
#include <netcdf.h>
   ...
int  status;              /* error status */
int  ncid;                /* netCDF ID */
int  rh_id;               /* variable ID */
int  vr_len, t_len;       /* attribute lengths */
```

```
double *vr_val;           /* ptr to attribute values */
char *title;              /* ptr to attribute values */
extern char *malloc();    /* memory allocator */

   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
/* find out how much space is needed for attribute values */
status = nc_inq_attlen (ncid, rh_id, "valid_range", &vr_len);
if (status != NC_NOERR) handle_error(status);
status = nc_inq_attlen (ncid, NC_GLOBAL, "title", &t_len);
if (status != NC_NOERR) handle_error(status);

/* allocate required space before retrieving values */
vr_val = (double *) malloc(vr_len * sizeof(double));
title = (char *) malloc(t_len + 1);  /* + 1 for trailing null */

/* get attribute values */
status = nc_get_att_double(ncid, rh_id, "valid_range", vr_val);
if (status != NC_NOERR) handle_error(status);
status = nc_get_att_text(ncid, NC_GLOBAL, "title", title);
if (status != NC_NOERR) handle_error(status);
title[t_len] = '\0';       /* null terminate */
   ...
```

5.5 Copy Attribute from One NetCDF to Another: nc_copy_att

The function nc_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.

Usage

```
int nc_copy_att (int ncid_in, int varid_in, const char *name,
                 int ncid_out, int varid_out);
```

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

`varid_in` ID of the variable in the input netCDF dataset from which the attribute will be copied, or NC_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 nc_open or nc_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 NC_GLOBAL to copy to a global attribute.

Errors

nc_copy_att returns the value NC_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 nc_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:

```
#include <netcdf.h>
   ...
int  status;               /* error status */
int  ncid1, ncid2;         /* netCDF IDs */
int  rh_id, avgrh_id;      /* variable IDs */
   ...
status = nc_open("foo.nc", NC_NOWRITE, ncid1);
if (status != NC_NOERR) handle_error(status);
status = nc_open("bar.nc", NC_WRITE, ncid2);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid1, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
status = nc_inq_varid (ncid2, "avgrh", &avgrh_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_redef(ncid2);  /* enter define mode */
if (status != NC_NOERR) handle_error(status);
/* copy variable attribute from "rh" to "avgrh" */
status = nc_copy_att(ncid1, rh_id, "units", ncid2, avgrh_id);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_enddef(ncid2); /* leave define mode */
if (status != NC_NOERR) handle_error(status);
```

5.6 Rename an Attribute: nc_rename_att

The function nc_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

```
int nc_rename_att (int ncid, int varid, const char* name,
                   const char* newname);
```

`ncid` NetCDF ID, from a previous call to nc_open or nc_create

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

`name` 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

nc_rename_att returns the value NC_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 nc_rename_att to rename the variable attribute units to Units for a variable rh in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int  status;      /* error status */
int  ncid;        /* netCDF ID */
int  rh_id;       /* variable id */
   ...
status = nc_open("foo.nc", NC_NOWRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
/* rename attribute */
status = nc_rename_att(ncid, rh_id, "units", "Units");
if (status != NC_NOERR) handle_error(status);
```

5.7 Delete an Attribute: nc_del_att

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

Usage

int nc_del_att (int ncid, int varid, const char* name);

`ncid` NetCDF ID, from a previous call to nc_open or nc_create.

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

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

Errors

nc_del_att returns the value NC_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 nc_del_att to delete the variable attribute Units for a variable rh in an existing netCDF dataset named foo.nc:

```
#include <netcdf.h>
   ...
int  status;      /* error status */
int  ncid;        /* netCDF ID */
int  rh_id;       /* variable ID */
   ...
status = nc_open("foo.nc", NC_WRITE, &ncid);
if (status != NC_NOERR) handle_error(status);
   ...
status = nc_inq_varid (ncid, "rh", &rh_id);
if (status != NC_NOERR) handle_error(status);
   ...
/* delete attribute */
status = nc_redef(ncid);        /* enter define mode */
if (status != NC_NOERR) handle_error(status);
status = nc_del_att(ncid, rh_id, "Units");
if (status != NC_NOERR) handle_error(status);
status = nc_enddef(ncid);       /* leave define mode */
if (status != NC_NOERR) handle_error(status);
```

Appendix A Summary of C Interface

```
const char* nc_inq_libvers (void);
const char* nc_strerror    (int ncerr);

int nc_create         (const char *path, int cmode, int *ncidp);
int nc_open           (const char *path, int mode, int *ncidp);
int nc_set_fill       (int ncid, int fillmode, int *old_modep);
int nc_redef          (int ncid);
int nc_enddef         (int ncid);
int nc_sync           (int ncid);
int nc_abort          (int ncid);
int nc_close          (int ncid);
int nc_inq            (int ncid, int *ndimsp, int *nvarsp,
                       int *ngattsp, int *unlimdimidp);
int nc_inq_ndims      (int ncid, int *ndimsp);
int nc_inq_nvars      (int ncid, int *nvarsp);
int nc_inq_natts      (int ncid, int *ngattsp);
int nc_inq_unlimdim   (int ncid, int *unlimdimidp);

int nc_def_dim        (int ncid, const char *name, size_t len,
                       int *idp);
int nc_inq_dimid      (int ncid, const char *name, int *idp);
int nc_inq_dim        (int ncid, int dimid, char *name, size_t *lenp);
int nc_inq_dimname    (int ncid, int dimid, char *name);
int nc_inq_dimlen     (int ncid, int dimid, size_t *lenp);
int nc_rename_dim     (int ncid, int dimid, const char *name);

int nc_def_var        (int ncid, const char *name, nc_type xtype,
                       int ndims, const int *dimidsp, int *varidp);
int nc_inq_var        (int ncid, int varid, char *name,
                       nc_type *xtypep, int *ndimsp, int *dimidsp,
                       int *nattsp);
int nc_inq_varid      (int ncid, const char *name, int *varidp);
int nc_inq_varname    (int ncid, int varid, char *name);
int nc_inq_vartype    (int ncid, int varid, nc_type *xtypep);
int nc_inq_varndims   (int ncid, int varid, int *ndimsp);
int nc_inq_vardimid   (int ncid, int varid, int *dimidsp);
int nc_inq_varnatts   (int ncid, int varid, int *nattsp);
int nc_rename_var     (int ncid, int varid, const char *name);
int nc_put_var_text   (int ncid, int varid, const char *op);
int nc_get_var_text   (int ncid, int varid,       char *ip);
int nc_put_var_uchar  (int ncid, int varid, const unsigned char *op);
int nc_get_var_uchar  (int ncid, int varid,       unsigned char *ip);
int nc_put_var_schar  (int ncid, int varid, const signed char *op);
int nc_get_var_schar  (int ncid, int varid,       signed char *ip);
int nc_put_var_short  (int ncid, int varid, const short *op);
```

```
int nc_get_var_short  (int ncid, int varid,       short *ip);
int nc_put_var_int    (int ncid, int varid, const int *op);
int nc_get_var_int    (int ncid, int varid,       int *ip);
int nc_put_var_long   (int ncid, int varid, const long *op);
int nc_get_var_long   (int ncid, int varid,       long *ip);
int nc_put_var_float  (int ncid, int varid, const float *op);
int nc_get_var_float  (int ncid, int varid,       float *ip);
int nc_put_var_double (int ncid, int varid, const double *op);
int nc_get_var_double (int ncid, int varid,       double *ip);
int nc_put_var1_text  (int ncid, int varid, const size_t *indexp,
                       const char *op);
int nc_get_var1_text  (int ncid, int varid, const size_t *indexp,
                       char *ip);
int nc_put_var1_uchar (int ncid, int varid, const size_t *indexp,
                       const unsigned char *op);
int nc_get_var1_uchar (int ncid, int varid, const size_t *indexp,
                       unsigned char *ip);
int nc_put_var1_schar (int ncid, int varid, const size_t *indexp,
                       const signed char *op);
int nc_get_var1_schar (int ncid, int varid, const size_t *indexp,
                       signed char *ip);
int nc_put_var1_short (int ncid, int varid, const size_t *indexp,
                       const short *op);
int nc_get_var1_short (int ncid, int varid, const size_t *indexp,
                       short *ip);
int nc_put_var1_int   (int ncid, int varid, const size_t *indexp,
                       const int *op);
int nc_get_var1_int   (int ncid, int varid, const size_t *indexp,
                       int *ip);
int nc_put_var1_long  (int ncid, int varid, const size_t *indexp,
                       const long *op);
int nc_get_var1_long  (int ncid, int varid, const size_t *indexp,
                       long *ip);
int nc_put_var1_float (int ncid, int varid, const size_t *indexp,
                       const float *op);
int nc_get_var1_float (int ncid, int varid, const size_t *indexp,
                       float *ip);
int nc_put_var1_double(int ncid, int varid, const size_t *indexp,
                       const double *op);
int nc_get_var1_double(int ncid, int varid, const size_t *indexp,
                       double *ip);
int nc_put_vara_text  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const char *op);
int nc_get_vara_text  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, char *ip);
int nc_put_vara_uchar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const unsigned char *op);
```

```
int nc_get_vara_uchar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, unsigned char *ip);
int nc_put_vara_schar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const signed char *op);
int nc_get_vara_schar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, signed char *ip);
int nc_put_vara_short (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const short *op);
int nc_get_vara_short (int ncid, int varid, const size_t *startp,
                       const size_t *countp, short *ip);
int nc_put_vara_int   (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const int *op);
int nc_get_vara_int   (int ncid, int varid, const size_t *startp,
                       const size_t *countp, int *ip);
int nc_put_vara_long  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const long *op);
int nc_get_vara_long  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, long *ip);
int nc_put_vara_float (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const float *op);
int nc_get_vara_float (int ncid, int varid, const size_t *startp,
                       const size_t *countp, float *ip);
int nc_put_vara_double(int ncid, int varid, const size_t *startp,
                       const size_t *countp, const double *op);
int nc_get_vara_double(int ncid, int varid, const size_t *startp,
                       const size_t *countp, double *ip);
int nc_put_vars_text  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const char *op);
int nc_get_vars_text  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       char *ip);
int nc_put_vars_uchar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const unsigned char *op);
int nc_get_vars_uchar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       unsigned char *ip);
int nc_put_vars_schar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const signed char *op);
int nc_get_vars_schar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       signed char *ip);
int nc_put_vars_short (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const short *op);
```

```
int nc_get_vars_short (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       short *ip);
int nc_put_vars_int   (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const int *op);
int nc_get_vars_int   (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       int *ip);
int nc_put_vars_long  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const long *op);
int nc_get_vars_long  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       long *ip);
int nc_put_vars_float (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const float *op);
int nc_get_vars_float (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       float *ip);
int nc_put_vars_double(int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const double *op);
int nc_get_vars_double(int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       double *ip);
int nc_put_varm_text  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imapp, const char *op);
int nc_get_varm_text  (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imapp, char *ip);
int nc_put_varm_uchar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imapp, const unsigned char *op);
int nc_get_varm_uchar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imapp, unsigned char *ip);
int nc_put_varm_schar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imapp, const signed char *op);
int nc_get_varm_schar (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imapp, signed char *ip);
int nc_put_varm_short (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
```

```
                           const ptrdiff_t *imapp, const short *op);
int nc_get_varm_short (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, short *ip);
int nc_put_varm_int   (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, const int *op);
int nc_get_varm_int   (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, int *ip);
int nc_put_varm_long  (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, const long *op);
int nc_get_varm_long  (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, long *ip);
int nc_put_varm_float (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, const float *op);
int nc_get_varm_float (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, float *ip);
int nc_put_varm_double(int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imapp, const double *op);
int nc_get_varm_double(int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t * imap, double *ip);

int nc_inq_att        (int ncid, int varid, const char *name,
                       nc_type *xtypep, size_t *lenp);
int nc_inq_attid      (int ncid, int varid, const char *name, int *idp);
int nc_inq_atttype    (int ncid, int varid, const char *name,
                       nc_type *xtypep);
int nc_inq_attlen     (int ncid, int varid, const char *name,
                       size_t *lenp);
int nc_inq_attname    (int ncid, int varid, int attnum, char *name);
int nc_copy_att       (int ncid_in, int varid_in, const char *name,
                       int ncid_out, int varid_out);
int nc_rename_att     (int ncid, int varid, const char *name,
                       const char *newname);
int nc_del_att        (int ncid, int varid, const char *name);
int nc_put_att_text   (int ncid, int varid, const char *name, size_t len,
                       const char *op);
int nc_get_att_text   (int ncid, int varid, const char *name, char *ip);
int nc_put_att_uchar  (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const unsigned char *op);
```

```
int nc_get_att_uchar  (int ncid, int varid, const char *name,
                       unsigned char *ip);
int nc_put_att_schar  (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const signed char *op);
int nc_get_att_schar  (int ncid, int varid, const char *name,
                       signed char *ip);
int nc_put_att_short  (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const short *op);
int nc_get_att_short  (int ncid, int varid, const char *name, short *ip);
int nc_put_att_int    (int ncid, int varid, const char *name,
                       nc_type xtype,size_t len, const int *op);
int nc_get_att_int    (int ncid, int varid, const char *name, int *ip);
int nc_put_att_long   (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const long *op);
int nc_get_att_long   (int ncid, int varid, const char *name, long *ip);
int nc_put_att_float  (int ncid, int varid, const char *name,
                        nc_type xtype, size_t len, const float *op);
int nc_get_att_float  (int ncid, int varid, const char *name, float *ip);
int nc_put_att_double (int ncid, int varid, const char *name,
                       nc_type xtype, size_t len, const double *op);
int nc_get_att_double (int ncid, int varid, const char *name,
                       double *ip);
```

Appendix B NetCDF 2 C Transition Guide

B.1 Overview of C interface changes

NetCDF version 3 includes a complete rewrite of the netCDF library. It is about twice as fast as the previous version. The netCDF file format is unchanged, so files written with version 3 can be read with version 2 code and vice versa.

The core library is now written in ANSI C. For example, prototypes are used throughout as well as const qualifiers where appropriate. You must have an ANSI C compiler to compile this version.

Rewriting the library offered an opportunity to implement improved C and FORTRAN interfaces that provide some significant benefits:

type safety, by eliminating the need to use generic void* pointers;

automatic type conversions, by eliminating the undesirable coupling between the language-independent external netCDF types (NC_BYTE, ..., NC_DOUBLE) and language-dependent internal data types (char, ..., double);

support for future enhancements, by eliminating obstacles to the clean addition of support for packed data and multithreading;

more standard error behavior, by uniformly communicating an error status back to the calling program in the return value of each function.

It is not necessary to rewrite programs that use the version 2 C interface, because the netCDF-3 library includes a backward compatibility interface that supports all the old functions, globals, and behavior. We are hoping that the benefits of the new interface will be an incentive to use it in new netCDF applications. It is possible to convert old applications to the new interface incrementally, replacing netCDF-2 calls with the corresponding netCDF-3 calls one at a time. If you want to check that only netCDF-3 calls are used in an application, a preprocessor macro (NO_NETCDF_2) is available for that purpose.

Other changes in the implementation of netCDF result in improved portability, maintainability, and performance on most platforms. A clean separation between I/O and type layers facilitates platform-specific optimizations. The new library no longer uses a vendor-provided XDR library, which simplifies linking programs that use netCDF and speeds up data access significantly in most cases.

B.2 The New C Interface

First, here's an example of C code that uses the netCDF-2 interface:

```
void *bufferp;
nc_type xtype;
ncvarinq(ncid, varid, ..., &xtype, ...
...
/* allocate bufferp based on dimensions and type */
...
if (ncvarget(ncid, varid, start, count, bufferp) == -1) {
    fprintf(stderr, "Can't get data, error code = %d\n",ncerr);
    /* deal with it */
```

```
        ...
    }
    switch(xtype) {
        /* deal with the data, according to type */
    ...
    case  NC_FLOAT:
        fanalyze((float *)bufferp);
        break;
    case NC_DOUBLE:
        danalyze((double *)bufferp);
        break;
    }
```

Here's how you might handle this with the new netCDF-3 C interface:

```
    /*
     * I want to use doubles for my analysis.
     */
    double dbuf[NDOUBLES];
    int status;

    /* So, I use the function that gets the data as doubles. */
    status = nc_get_vara_double(ncid, varid, start, count, dbuf)
    if (status != NC_NOERR) {
       fprintf(stderr, "Can't get data: %s\n", nc_strerror(status));
        /* deal with it */
        ...
    }
    danalyze(dbuf);
```

The example above illustrates changes in function names, data type conversion, and error handling, discussed in detail in the sections below.

B.3 Function Naming Conventions

The netCDF-3 C library employs a new naming convention, intended to make netCDF programs more readable. For example, the name of the function to rename a variable is now nc_rename_var instead of the previous ncvarrename.

All netCDF-3 C function names begin with the nc_ prefix. The second part of the name is a verb, like get, put, inq (for inquire), or open. The third part of the name is typically the object of the verb: for example dim, var, or att for functions dealing with dimensions, variables, or attributes. To distinguish the various I/O operations for variables, a single character modifier is appended to var:

var entire variable access var1 single value access vara array or array section access vars strided access to a subsample of values varm mapped access to values not contiguous in memory

At the end of the name for variable and attribute functions, there is a component indicating the type of the final argument: text, uchar, schar, short, int, long, float, or double.

This part of the function name indicates the type of the data container you are using in your program: character string, unsigned char, signed char, and so on.

Also, all macro names in the public C interface begin with the prefix NC_. For example, the macro which was formerly MAX_NC_NAME is now NC_MAX_NAME, and the former FILL_FLOAT is now NC_FILL_FLOAT.

As previously mentioned, all the old names are still supported for backward compatibility.

B.4 Type Conversion

With the new interface, users need not be aware of the external type of numeric variables, since automatic conversion to or from any desired numeric type is now available. You can use this feature to simplify code, by making it independent of external types. The elimination of void* pointers provides detection of type errors at compile time that could not be detected with the previous interface. Programs may be made more robust with the new interface, because they need not be changed to accommodate a change to the external type of a variable.

If conversion to or from an external numeric type is necessary, it is handled by the library. This automatic conversion and separation of external data representation from internal data types will become even more important in netCDF version 4, when new external types will be added for packed data for which there is no natural corresponding internal type, for example, arrays of 11-bit values.

Converting from one numeric type to another may result in an error if the target type is not capable of representing the converted value. (In netCDF-2, such overflows can only happen in the XDR layer.) For example, a float may not be able to hold data stored externally as an NC_DOUBLE (an IEEE floating-point number). When accessing an array of values, an NC_ERANGE error is returned if one or more values are out of the range of representable values, but other values are converted properly.

Note that mere loss of precision in type conversion does not return an error. Thus, if you read double precision values into an int, for example, no error results unless the magnitude of the double precision value exceeds the representable range of ints on your platform. Similarly, if you read a large integer into a float incapable of representing all the bits of the integer in its mantissa, this loss of precision will not result in an error. If you want to avoid such precision loss, check the external types of the variables you access to make sure you use an internal type that has a compatible precision.

The new interface distinguishes arrays of characters intended to represent text strings from arrays of 8-bit bytes intended to represent small integers. The interface supports the internal types text, uchar, and schar, intended for text strings, unsigned byte values, and signed byte values.

The _uchar and _schar functions were introduced in netCDF-3 to eliminate an ambiguity, and support both signed and unsigned byte data. In netCDF-2, whether the external NC_BYTE type represented signed or unsigned values was left up to the user. In netcdf-3, we treat NC_BYTE as signed for the purposes of conversion to short, int, long, float, or double. (Of course, no conversion takes place when the internal type is signed char.) In the _uchar functions, we treat NC_BYTE as if it were unsigned. Thus, no NC_ERANGE error can occur converting between NC_BYTE and unsigned char.

B.5 Error handling

The new interface handles errors differently than netCDF-2. In the old interface, the default behavior when an error was detected was to print an error message and exit. To get control of error handling, you had to set flag bits in a global variable, ncopts, and to determine the cause of an error, you had to test the value of another global variable ncerr.

In the new interface, functions return an integer status that indicates not only success or failure, but also the cause of the error. The global variables ncerr and ncopt have been eliminated. The library will never try to print anything, nor will it call exit (unless you are using the netCDF version 2 compatibility functions). You will have to check the function return status and do this yourself. We eliminated these globals in the interest of supporting parallel (multiprocessor) execution cleanly, as well as reducing the number of assumptions about the environment where netCDF is used. The new behavior should provide better support for using netCDF as a hidden layer in applications that have their own GUI interface.

B.6 NC_LONG and NC_INT

Where the netCDF-2 interface used NC_LONG to identify an external data type corresponding to 32-bit integers, the new interface uses NC_INT instead. NC_LONG is defined to have the same value as NC_INT for backward compatibility, but it should not be used in new code. With new 64-bit platforms using long for 64-bit integers, we would like to reduce the confusion caused by this name clash. Note that there is still no netCDF external data type corresponding to 64-bit integers.

B.7 What's Missing?

The new C interface omits three "record I/O" functions, ncrecput, ncrecget, and ncrecinq, from the netCDF-2 interface, although these functions are still supported via the netCDF-2 compatibility interface.

This means you may have to replace one record-oriented call with multiple type-specific calls, one for each record variable. For example, a single call to ncrecput can always be replaced by multiple calls to the appropriate nc_put_var functions, one call for each variable accessed. The record-oriented functions were omitted, because there is no simple way to provide type-safety and automatic type conversion for such an interface.

There is no function corresponding to the nctypelen function from the version 2 interface. The separation of internal and external types and the new type-conversion interfaces make nctypelen unnecessary. Since users read into and write out of native types, the sizeof operator is perfectly adequate to determine how much space to allocate for a value.

In the previous library, there was no checking that the characters used in the name of a netCDF object were compatible with CDL restrictions. Now this restriction is also enforced by the library for creation of new dimensions, variables, and attributes. Previously existing components with less restrictive names will still work OK.

B.8 Other Changes

There are two new functions in netCDF-3 that don't correspond to any netCDF-2 functions: nc_inq_libvers and nc_strerror. The version of the netCDF library in use is returned as a string by nc_inq_libvers. An error message corresponding to the status returned by a netCDF function call is returned as a string by the nc_strerror function.

A new NC_SHARE flag is available for use in an nc_open or nc_create call, to suppress the default buffering of accesses. The use of NC_SHARE for concurrent access to a netCDF dataset means you don't have to call nc_sync after every access to make sure that disk updates are synchronous. It is important to note that changes to ancillary data, such as attribute values, are not propagated automatically by use of the NC_SHARE flag. Use of the nc_sync function is still required for this purpose.

The version 2 interface had a single inquiry function, ncvarinq for getting the name, type, and shape of a variable. Similarly, only a single inquiry function was available for getting information about a dimension, an attribute, or a netCDF dataset. When you only wanted a subset of this information, you had to provide NULL arguments as placeholders for the unneeded information. The new interface includes additional inquire functions that return each item separately, so errors are less likely from miscounting arguments.

The previous implementation returned an error when 0-valued count components were specified in ncvarput and ncvarget calls. This restriction has been removed, so that now functions in the nc_put_var and nc_get_var families may be called with 0-valued count components, resulting in no data being accessed. Although this may seem useless, it simplifies some programs to not treat 0-valued counts as a special case.

The previous implementation returned an error when the same dimension was used more than once in specifying the shape of a variable in ncvardef. This restriction is relaxed in the netCDF-3 implementation, because an autocorrelation matrix is a good example where using the same dimension twice makes sense.

In the new interface, units for the imap argument to the nc_put_varm and nc_get_varm families of functions are now in terms of the number of data elements of the desired internal type, not in terms of bytes as in the netCDF version-2 mapped access interfaces.

Following is a table of netCDF-2 function names and names of the corresponding netCDF-3 functions. For parameter lists of netCDF-2 functions, see the netCDF-2 User's Guide.

`ncabort` nc_abort

`ncattcopy`
nc_copy_att

`ncattdel` nc_del_att

`ncattget` nc_get_att_double, nc_get_att_float, nc_get_att_int, nc_get_att_long, nc_get_att_schar, nc_get_att_short, nc_get_att_text, nc_get_att_uchar

`ncattinq` nc_inq_att, nc_inq_attid, nc_inq_attlen, nc_inq_atttype

`ncattname`
nc_inq_attname

`ncattput` nc_put_att_double, nc_put_att_float, nc_put_att_int, nc_put_att_long, nc_put_att_schar, nc_put_att_short, nc_put_att_text, nc_put_att_uchar

`ncattrename`
nc_rename_att

ncclose nc_close

nccreate nc_create

ncdimdef nc_def_dim

ncdimid nc_inq_dimid

ncdiminq nc_inq_dim, nc_inq_dimlen, nc_inq_dimname

ncdimrename
nc_rename_dim

ncendef nc_enddef

ncinquire
nc_inq, nc_inq_natts, nc_inq_ndims, nc_inq_nvars, nc_inq_unlimdim

ncopen nc_open

ncrecget (none)

ncrecinq (none)

ncrecput (none)

ncredef nc_redef

ncsetfill
nc_set_fill

ncsync nc_sync

nctypelen
(none)

ncvardef nc_def_var

ncvarget nc_get_vara_double, nc_get_vara_float, nc_get_vara_int, nc_get_vara_long, nc_get_vara_schar, nc_get_vara_short, nc_get_vara_text, nc_get_vara_uchar

ncvarget1
nc_get_var1_double, nc_get_var1_float, nc_get_var1_int, nc_get_var1_long, nc_get_var1_schar, nc_get_var1_short, nc_get_var1_text, nc_get_var1_uchar

ncvargetg
nc_get_varm_double, nc_get_varm_float, nc_get_varm_int, nc_get_varm_long, nc_get_varm_schar, nc_get_varm_short, nc_get_varm_text, nc_get_varm_uchar, nc_get_vars_double, nc_get_vars_float, nc_get_vars_int, nc_get_vars_long, nc_get_vars_schar, nc_get_vars_short, nc_get_vars_text, nc_get_vars_uchar

ncvarid nc_inq_varid

ncvarinq nc_inq_var, nc_inq_vardimid, nc_inq_varname, nc_inq_varnatts, nc_inq_varndims, nc_inq_vartype

ncvarput nc_put_vara_double, nc_put_vara_float, nc_put_vara_int, nc_put_vara_long, nc_put_vara_schar, nc_put_vara_short, nc_put_vara_text, nc_put_vara_uchar

`ncvarput1`
: nc_put_var1_double, nc_put_var1_float, nc_put_var1_int, nc_put_var1_long, nc_put_var1_schar, nc_put_var1_short, nc_put_var1_text, nc_put_var1_uchar

`ncvarputg`
: nc_put_varm_double, nc_put_varm_float, nc_put_varm_int, nc_put_varm_long, nc_put_varm_schar, nc_put_varm_short, nc_put_varm_text, nc_put_varm_uchar, nc_put_vars_double, nc_put_vars_float, nc_put_vars_int, nc_put_vars_long, nc_put_vars_schar, nc_put_vars_short, nc_put_vars_text, nc_put_vars_uchar

`ncvarrename`
: nc_rename_var

`(none)` nc_inq_libvers

`(none)` nc_strerror

Appendix C Error Codes

```
#define NC_NOERR        0       /* No Error */

#define NC_EBADID       (-33)   /* Not a netcdf id */
#define NC_ENFILE       (-34)   /* Too many netcdfs open */
#define NC_EEXIST       (-35)   /* netcdf file exists && NC_NOCLOBBER */
#define NC_EINVAL       (-36)   /* Invalid Argument */
#define NC_EPERM        (-37)   /* Write to read only */
#define NC_ENOTINDEFINE (-38)   /* Operation not allowed in data mode */
#define NC_EINDEFINE    (-39)   /* Operation not allowed in define mode */
#define NC_EINVALCOORDS (-40)   /* Index exceeds dimension bound */
#define NC_EMAXDIMS     (-41)   /* NC_MAX_DIMS exceeded */
#define NC_ENAMEINUSE   (-42)   /* String match to name in use */
#define NC_ENOTATT      (-43)   /* Attribute not found */
#define NC_EMAXATTS     (-44)   /* NC_MAX_ATTRS exceeded */
#define NC_EBADTYPE     (-45)   /* Not a netcdf data type */
#define NC_EBADDIM      (-46)   /* Invalid dimension id or name */
#define NC_EUNLIMPOS    (-47)   /* NC_UNLIMITED in the wrong index */
#define NC_EMAXVARS     (-48)   /* NC_MAX_VARS exceeded */
#define NC_ENOTVAR      (-49)   /* Variable not found */
#define NC_EGLOBAL      (-50)   /* Action prohibited on NC_GLOBAL varid */
#define NC_ENOTNC       (-51)   /* Not a netcdf file */
#define NC_ESTS         (-52)   /* In Fortran, string too short */
#define NC_EMAXNAME     (-53)   /* NC_MAX_NAME exceeded */
#define NC_EUNLIMIT     (-54)   /* NC_UNLIMITED size already in use */
#define NC_ENORECVARS   (-55)   /* nc_rec op when there are no record vars */
#define NC_ECHAR        (-56)   /* Attempt to convert between text & numbers */
#define NC_EEDGE        (-57)   /* Edge+start exceeds dimension bound */
#define NC_ESTRIDE      (-58)   /* Illegal stride */
#define NC_EBADNAME     (-59)   /* Attribute or variable name
                                         contains illegal characters */
/* N.B. following must match value in ncx.h */
#define NC_ERANGE       (-60)   /* Math result not representable */
#define NC_ENOMEM       (-61)   /* Memory allocation (malloc) failure */

#define NC_EVARSIZE     (-62)   /* One or more variable sizes violate
                                   format constraints */
#define NC_EDIMSIZE     (-63)   /* Invalid dimension size */
#define NC_ETRUNC       (-64)   /* File likely truncated or possibly corrupted */
```

Index

R