

The NetCDF C Interface Guide

NetCDF Version 4.0.1
Last Updated 22 June 2009

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

Copyright © 2005-2006 University Corporation for Atmospheric Research

Permission is granted to make and distribute verbatim copies of this manual provided that the copyright notice and these paragraphs are preserved on all copies. The software and any accompanying written materials are provided “as is” without warranty of any kind. UCAR expressly disclaims all warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

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.

Mention of any commercial company or product in this document does not constitute an endorsement by the Unidata Program Center. Unidata does not authorize any use of information from this publication for advertising or publicity purposes.

Table of Contents

1	Use of the NetCDF Library	3
1.1	Creating a NetCDF Dataset	3
1.2	Reading a NetCDF Dataset with Known Names	4
1.3	Reading a netCDF Dataset with Unknown Names	5
1.4	Adding New Dimensions, Variables, Attributes	6
1.5	Error Handling	7
1.6	Compiling and Linking with the NetCDF Library	7
2	Datasets	9
2.1	NetCDF Library Interface Descriptions	9
2.2	Parallel Access for NetCDF Files	10
2.3	Get error message corresponding to error status: nc_strerror	12
2.4	Get netCDF library version: nc_inq_libvers	13
2.5	Create a NetCDF Dataset: nc_create	13
2.6	Create a NetCDF Dataset With Performance Options: nc__create	16
2.7	Create a NetCDF Dataset With Performance Options: nc_create_par	18
2.8	Open a NetCDF Dataset for Access: nc_open	19
2.9	Open a NetCDF Dataset for Access with Performance Tuning: nc__open	21
2.10	Open a NetCDF Dataset for Parallel Access	22
2.11	Put Open NetCDF Dataset into Define Mode: nc_redef	23
2.12	Leave Define Mode: nc_enddef	25
2.13	Leave Define Mode with Performance Tuning: nc__enddef	26
2.14	Close an Open NetCDF Dataset: nc_close	27
2.15	Inquire about an Open NetCDF Dataset: nc_inq Family	28
2.16	Synchronize an Open NetCDF Dataset to Disk: nc_sync	30
2.17	Back Out of Recent Definitions: nc_abort	31
2.18	Set Fill Mode for Writes: nc_set_fill	32
2.19	Set Default Creation Format: nc_set_default_format	34
3	Groups	37
3.1	Find a Group ID: nc_inq_ncid	37
3.2	Get a List of Groups in a Group: nc_inq_grps	38
3.3	Find all the Variables in a Group: nc_inq_varids	39
3.4	Find all Dimensions Visible in a Group: nc_inq_dimids	40
3.5	Find a Group's Name: nc_inq_grpname	41
3.6	Find a Group's Full Name: nc_inq_grpname_full	42
3.7	Find the Length of a Group's Full Name: nc_inq_grpname_len	43
3.8	Find a Group's Parent: nc_inq_grp_parent	44
3.9	Create a New Group: nc_def_grp	44

4	Dimensions	47
4.1	Dimensions Introduction	47
4.2	Create a Dimension: nc_def_dim	47
4.3	Get a Dimension ID from Its Name: nc_inq_dimid	48
4.4	Inquire about a Dimension: nc_inq_dim Family	49
4.5	Rename a Dimension: nc_rename_dim	50
4.6	Find All Unlimited Dimension IDs: nc_inq_unlimdims	51
5	User Defined Data Types	53
5.1	User Defined Types Introduction	53
5.2	Learn the IDs of All Types in Group: nc_inq_typeids	53
5.3	Learn About a User Defined Type: nc_inq_type	54
5.4	Learn About a User Defined Type: nc_inq_user_type	55
5.5	Compound Types Introduction	56
5.6	Creating a Compound Type: nc_def_compound	56
5.7	Inserting a Field into a Compound Type: nc_insert_compound	58
5.8	Inserting an Array Field into a Compound Type: nc_insert_array_compound	59
5.9	Learn About a Compound Type: nc_inq_compound	61
5.10	Learn the Name of a Compound Type: nc_inq_compound_name	63
5.11	Learn the Size of a Compound Type: nc_inq_compound_size	64
5.12	Learn the Number of Fields in a Compound Type: nc_inq_compound_nfields	64
5.13	Learn About a Field of a Compound Type: nc_inq_compound_field	65
5.14	Find the Name of a Field in a Compound Type: nc_inq_compound_fieldname	66
5.15	Get the FieldID of a Compound Type Field: nc_inq_compound_fieldindex	66
5.16	Get the Offset of a Field: nc_inq_compound_fieldoffset	67
5.17	Find the Type of a Field: nc_inq_compound_fieldtype	67
5.18	Find the Number of Dimensions in an Array Field: nc_inq_compound_fieldndims	68
5.19	Find the Sizes of Dimensions in an Array Field: nc_inq_compound_fielddim_sizes	69
5.20	Variable Length Array Introduction	70
5.21	Define a Variable Length Array (VLEN): nc_def_vlen	70
5.22	Learning about a Variable Length Array (VLEN) Type: nc_inq_vlen	71
5.23	Releasing Memory for a Variable Length Array (VLEN) Type: nc_free_vlen	72
5.24	Opaque Type Introduction	73
5.25	Creating Opaque Types: nc_def_opaque	73
5.26	Learn About an Opaque Type: nc_inq_opaque	74
5.27	Enum Type Introduction	74
5.28	Creating a Enum Type: nc_def_enum	75

5.29	Inserting a Field into a Enum Type: <code>nc_insert_enum</code>	77
5.30	Learn About a Enum Type: <code>nc_inq_enum</code>	78
5.31	Learn the Name of a Enum Type: <code>nc_inq_enum_member</code>	80
5.32	Learn the Name of a Enum Type: <code>nc_inq_enum_ident</code>	81
6	Variables	83
6.1	Introduction	83
6.2	Language Types Corresponding to netCDF external data types	83
6.3	NetCDF-3 Classic and 64-Bit Offset Data Types.....	84
6.4	NetCDF-4 Atomic Types.....	84
6.5	Create a Variable: <code>nc_def_var</code>	85
6.6	Define Chunking Parameters for a Variable: <code>nc_def_var_chunking</code>	87
6.7	Learn About Chunking Parameters for a Variable: <code>nc_inq_var_chunking</code>	89
6.8	Define Fill Parameters for a Variable: <code>nc_def_var_fill</code>	90
6.9	Learn About Fill Parameters for a Variable: <code>nc_inq_var_fill</code>	92
6.10	Define Compression Parameters for a Variable: <code>nc_def_var_deflate</code>	93
6.11	Learn About Deflate Parameters for a Variable: <code>nc_inq_var_deflate</code>	94
6.12	Define Checksum Parameters for a Variable: <code>nc_def_var_fletcher32</code>	94
6.13	Learn About Checksum Parameters for a Variable: <code>nc_inq_var_fletcher32</code>	95
6.14	Define Endianness of a Variable: <code>nc_def_var_endian</code>	96
6.15	Learn About Endian Parameters for a Variable: <code>nc_inq_var_endian</code>	97
6.16	Get a Variable ID from Its Name: <code>nc_inq_varid</code>	98
6.17	Get Information about a Variable from Its ID: <code>nc_inq_var</code>	98
6.18	Write a Single Data Value: <code>nc_put_var1_type</code>	100
6.19	Write an Entire Variable: <code>nc_put_var_type</code>	102
6.20	Write an Array of Values: <code>nc_put_vara_type</code>	104
6.21	Write a Subsampled Array of Values: <code>nc_put_vars_type</code>	107
6.22	Write a Mapped Array of Values: <code>nc_put_varm_type</code>	110
6.23	Read a Single Data Value: <code>nc_get_var1_type</code>	115
6.24	Read an Entire Variable <code>nc_get_var_type</code>	117
6.25	Read an Array of Values: <code>nc_get_vara_type</code>	119
6.26	Read a Subsampled Array of Values: <code>nc_get_vars_type</code>	122
6.27	Read a Mapped Array of Values: <code>nc_get_varm_type</code>	125
6.28	Reading and Writing Character String Values	129
6.28.1	Reading and Writing Character String Values in the Classic Model.....	130
6.28.2	Reading and Writing Arrays of Strings	131
6.29	Releasing Memory for a NC_STRING: <code>nc_free_string</code>	132
6.30	Fill Values.....	133

6.31	Rename a Variable: <code>nc_rename_var</code>	133
6.32	Copy a Variable from One File to Another: <code>nc_copy_var</code>	134
6.33	Change between Collective and Independent Parallel Access: <code>nc_var_par_access</code>	135
7	Attributes	137
7.1	Introduction	137
7.2	Create an Attribute: <code>nc_put_att_type</code>	137
7.3	Get Information about an Attribute: <code>nc_inq_att</code> Family	140
7.4	Get Attribute's Values: <code>nc_get_att_type</code>	142
7.5	Copy Attribute from One NetCDF to Another: <code>nc_copy_att</code> ..	144
7.6	Rename an Attribute: <code>nc_rename_att</code>	145
7.7	Delete an Attribute: <code>nc_del_att</code>	146
Appendix A	Summary of C Interface	149
Appendix B	NetCDF 3 to NetCDF 4 Transition Guide	155
B.1	Introduction	155
B.2	NetCDF-4 and HDF5	155
B.3	Backward Compatibility	155
B.4	The Classic and the Expanded NetCDF Data Models	156
B.5	Using NetCDF-4.0 with the Classic and 64-bit Offset Formats	157
B.6	Creating a NetCDF-4/HDF5 File	157
B.7	Using NetCDF-4.0 with the Classic Model	157
B.8	Use of the Expanded Model Impacts Fortran Portability	157
B.9	The C++ API Does Not Handle Expanded Model in this Release	158
Appendix C	NetCDF 2 to NetCDF 3 C Transition Guide	159
C.1	Overview of C interface changes	159
C.2	The New C Interface	159
C.3	Function Naming Conventions	160
C.4	Type Conversion	161
C.5	Error handling	162
C.6	NC_LONG and NC_INT	162
C.7	What's Missing?	162
C.8	Other Changes	163
Appendix D	NetCDF-3 Error Codes	167
Appendix E	NetCDF-4 Error Codes	169
8	Index	171

This document describes the C interface to the netCDF library; it applies to netCDF version 4.0.1 and was last updated on 22 June 2009.

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_att` family 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).

For netCDF-4/HDF5 format files, define mode is still important, but the user does not have to call `nc_enddef` - it is called automatically when needed. It may also be called by the user.

In netCDF-4/HDF5 files, there are some settings which can only be modified during the very first define mode of the file. For example the compression level of a variable may be

set only after the `nc_def_var` call and before the next `nc_enddef` call, whether it is called by the user explicitly, or when the user tries to read or write some data.

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 2.3 \[nc_strerror\]](#), page 12).

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, for a non-netcdf-4 build:

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

For a netCDF-4 enabled library, you must also link with two HDF5, at least one compression library, and (on some systems) the math library:

```
cc -o myprogram myprogram.o -L/usr/local/netcdf/lib -L/usr/local/hdf5/lib -lnetcdf -lhdf5 -lhdf5_lz4 -lm
```

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 2.3 \[nc_strerror\]](#), [page 12](#).

2.2 Parallel Access for NetCDF Files

For netCDF-4 files only, parallel read/write access is possible on systems which support it, and only if parallel HDF5 was installed on the system before netCDF, and only if the HDF5 parallel compiler was used during the netCDF configure. (Parallel HDF5 requires the MPI library).

To use parallel access, open or create the file with `nc_open_par` (see [Section 2.10 \[nc_open_par\]](#), page 22) or `nc_create_par` (see [Section 2.7 \[nc_create_par\]](#), page 18). Only netCDF-4 files can be opened or created for parallel access.

The following example shows the creation of a file using parallel access, and how a program might write data to such a file.

```
#include "netcdf.h"
#include <mpi.h>
#include <assert.h>
#include "hdf5.h"
#include <string.h>
#include <stdlib.h>

#define BAIL(e) do { \
printf("Bailing out in file %s, line %d, error:%s.\n", __FILE__, __LINE__, nc_strerror(e)); \
return e; \
} while (0)

#define FILE "test_par.nc"
#define NDIMS 2
#define DIMSIZE 24
#define QTR_DATA (DIMSIZE*DIMSIZE/4)
#define NUM_PROC 4

int
main(int argc, char **argv)
{
    /* MPI stuff. */
    int mpi_namelen;
    char mpi_name[MPI_MAX_PROCESSOR_NAME];
    int mpi_size, mpi_rank;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Info info = MPI_INFO_NULL;

    /* Netcdf-4 stuff. */
    int ncid, vlid, dimids[NDIMS];
    size_t start[NDIMS], count[NDIMS];

    int data[DIMSIZE*DIMSIZE], j, i, res;

    /* Initialize MPI. */
    MPI_Init(&argc,&argv);
```



```

        BAIL(res);

    /* Close the netcdf file. */
    if ((res = nc_close(ncid)))
        BAIL(res);

    /* Shut down MPI. */
    MPI_Finalize();

    return 0;
}

```

2.3 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 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);
    }
}

```

2.4 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());
```

2.5 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), the new 64-bit offset format (use `NC_64BIT_OFFSET`), or `NC_NETCDF4` for a netCDF-4/HDF5 file.

Usage

NOTE: When creating a netCDF-4 file HDF5 error reporting is turned off, if it is on. This doesn't stop the HDF5 error stack from recording the errors, it simply stops their display to the user through `stderr`.

```
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_NO_CLOBBER`, `NC_SHARE`, `NC_64BIT_OFFSET`, `NC_NETCDF4`, `NC_CLASSIC_MODEL`. Setting `NC_NO_CLOBBER` 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. This flag is ignored for netCDF-4 files. (See below.)

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*](#).

Setting NC_NETCDF4 causes netCDF to create a HDF5/NetCDF-4 file.

Setting NC_CLASSIC_MODEL causes netCDF to enforce the classic data model in this file. (This only has effect for netCDF-4/HDF5 files, as classic and 64-bit offset files always use the classic model.) When used with NC_NETCDF4, this flag ensures that the resulting netCDF-4/HDF5 file may never contain any new constructs from the enhanced data model. That is, it cannot contain groups, user defined types, multiple unlimited dimensions, or new atomic types. The advantage of this restriction is that such files are guaranteed to work with existing netCDF software.

`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.

Return Codes

NC_NOERR No error.

NC_ENOMEM System out of memory.

NC_EHDFERR HDF5 error (netCDF-4 files only).

NC_EFILEMETA Error writing netCDF-4 file-level metadata in HDF5 file. (netCDF-4 files only).

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_large.nc", NC_NOCLOBBER|NC_64BIT_OFFSET, &ncid);
if (status != NC_NOERR) handle_error(status);
```

In this example we create a netCDF dataset named `foo_HDF5.nc`. It will be in the HDF5 format.

```
#include <netcdf.h>
...
int status;
int ncid;
...
status = nc_create("foo_HDF5.nc", NC_NOCLOBBER|NC_NETCDF4, &ncid);
if (status != NC_NOERR) handle_error(status);
```

In this example we create a netCDF dataset named `foo_HDF5_classic.nc`. It will be in the HDF5 format, but will not allow the use of any netCDF-4 advanced features. That is, it will conform to the classic netCDF-3 data model.

```
#include <netcdf.h>
...
int status;
int ncid;
...
status = nc_create("foo_HDF5_classic.nc", NC_NOCLOBBER|NC_NETCDF4|NC_CLASSIC_MODEL, &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`. See [Section 2.6 \[nc__create\]](#), page 16.

2.6 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 *bufersizehintp, 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`, `NC_NETCDF4`, `NC_CLASSIC_MODEL`.

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. This flag is ignored for netCDF-4 files. (See below.)

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*.

Setting `NC_NETCDF4` causes netCDF to create a HDF5/NetCDF-4 file.

Setting `NC_CLASSIC_MODEL` causes netCDF to enforce the classic data model in this file. (This only has effect for netCDF-4/HDF5 files, as classic and 64-bit offset files always use the classic model.) When used with `NC_NETCDF4`, this flag ensures that the resulting netCDF-4/HDF5 file may never contain any new constructs from the enhanced data model. That is,

it cannot contain groups, user defined types, multiple unlimited dimensions, or new atomic types. The advantage of this restriction is that such files are guaranteed to work with existing netCDF software.

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.

bufsizehintp

The argument referenced by `bufsizehintp` 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 `bufsize` to 8192.

The `bufsize` 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.

Return Codes

NC_NOERR No error.

NC_ENOMEM
System out of memory.

NC_EHDFERR
HDF5 error (netCDF-4 files only).

NC_EFILEMETA
Error writing netCDF-4 file-level metadata in HDF5 file. (netCDF-4 files only).

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 `bufsize` and initial size for the file.

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

2.7 Create a NetCDF Dataset With Performance Options: `nc_create_par`

This function is a variant of `nc_create`, `nc_create_par` allows users to open a file on a MPI/IO or MPI/Posix parallel file system.

The parallel parameters are not written to the data file, they are only used for so long as the file remains open after an `nc_create_par`.

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.

This function is only available for netCDF-4 files. The creation mode flag must include `NC_NETCDF4`.

When a netCDF-4 file is created for parallel access, collective operations are the default. To use independent access on a variable, See [Section 6.33 \[nc_var_par_access\]](#), page 135.

Usage

```
int nc_create_par(const char *path, int cmode, MPI_Comm comm,
                  MPI_Info info, int ncidp);
```

path The file name of the new netCDF dataset.

cmode	Either the NC_MPIO or NC_MPIOPOSIX flags must be present. The NC_NETCDF4 flag is also required. 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 ignored.
comm	The MPI_Comm object returned by the MPI layer.
info	The MPI_Info object returned by the MPI layer, if MPI/IO is being used, or 0 if MPI/Posix is being used.
ncidp	Pointer to location where returned netCDF ID is to be stored.

Return Codes

NC_NOERR	No error.
NC_ENOTNC4	Parallel access with netcdf-4 files only.
NC_EPARINIT	Could not initialize parallel access at the HDF5 layer.
NC_EFILEMETA	HDF5 layer cannot handle create or root group open.
NC_EEXIST	Specifying a dataset name of a file that exists and also specifying NC_NOCLOBBER.
NC_EINVAL	Bad value for the creation mode.
NC_EHDFERR	Unexpected error from the HDF5 layer.

Examples

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

2.8 Open a NetCDF Dataset for Access: nc_open

The function `nc_open` opens an existing netCDF dataset for access. It determines the underlying file format automatically. Use the same call to open a netCDF classic, 64-bit offset, or netCDF-4 file.

Usage

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

path File name for netCDF dataset to be opened. When DAP support is enabled, then the path may be an OPeNDAP URL rather than a file path.

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 open 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 only used for netCDF classic and 64-bit offset files. It 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.

It is not necessary to pass any information about the format of the file being opened. The file type will be detected automatically by the netCDF library.

If a the path is a DAP URL, then the open mode is read-only. Setting NC_WRITE will be ignored.

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

Errors

When opening a netCDF-4 file HDF5 error reporting is turned off, if it is on. This doesn't stop the HDF5 error stack from recording the errors, it simply stops their display to the user through stderr.

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.

Return Codes

NC_NOERR No error.

NC_NOMEM Out of memory.

NC_EHDFERR HDF5 error. (NetCDF-4 files only.)

NC_EDIMMETA Error in netCDF-4 dimension metadata. (NetCDF-4 files only.)

NC_ENOCOMPOIND (NetCDF-4 files only.)

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);
```

2.9 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. When DAP support is enabled, it is possible to open a DAP data source through this interface, but it is deprecated because all of the performance tuning parameters are ignored. The standard `nc_open` interface should be used instead.

Usage

```
int nc__open(const char *path, int mode, size_t *bufsizehintp, 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 open 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.

bufsizehintp

The argument referenced by `bufsizehintp` 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 bufrsize to 8192.

The bufrsize 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;
size_t bufrsize;
...
*bufrsize = 1024;
status = nc_open("foo.nc", 0, &bufrsize, &ncid);
if (status != NC_NOERR) handle_error(status);
```

2.10 Open a NetCDF Dataset for Parallel Access

This function opens a netCDF-4 dataset for parallel access.

This opens the file using either MPI-IO or MPI-POSIX. The file must be a netCDF-4 file. (That is, it must have been created using `NC_NETCDF4` in the creation mode). DAP access is not allowed through this interface.

This function is only available if netCDF-4 was configured with the `-use-parallel` option before being built. Also HDF5 parallel must be installed (before netCDF-4 is installed.)

Before either HDF5 or netCDF-4 can be installed with support for parallel programming, and MPI layer must also be installed on the machine, and usually a parallel file system.

NetCDF-4 exposes the parallel access functionality of HDF5. For more information about what is required to install and use the parallel access functions, see the HDF5 web site.

When a netCDF-4 file is opened for parallel access, collective operations are the default. To use independent access on a variable, See [Section 6.33 \[nc_var_par_access\]](#), page 135.

Usage

```
int nc_open_par(const char *path, int mode, MPI_Comm comm,
               MPI_Info info, int *ncidp);
```

path	File name for netCDF dataset to be opened.
omode	<p>Either the NC_MPIO or NC_MPIPOSIX flags must be present.</p> <p>The flag NC_WRITE 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.)</p> <p>All other flags are ignored or not allowed. The NC_NETCDF4 flag is not required, as the file type is detected when the file is opened.</p>
comm	MPI_Comm object returned by the MPI layer.
info	MPI_Info object returned by the MPI layer, or NULL if MPI-POSIX access is desired.
ncidp	Pointer to location where returned netCDF ID is to be stored.

Return Codes

NC_NOERR	No error.
NC_ENOTNC4	<p>Not a netCDF-4 file.</p> <p>The specified netCDF dataset does not exist.</p> <p>A meaningless mode was specified.</p>

Example

Here is an example using `nc_open_par` to open an existing netCDF dataset named `foo.nc` for read-only, non-shared, MPI/IO access:

```
#include <netcdf.h>
...
int status;
int ncid;
int *bufsize;
...
```

2.11 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

For netCDF-4 files (i.e. files created with NC_NETCDF4 in the cmode, see [Section 2.5 \[nc_create\], page 13](#)), it is not necessary to call `nc_redef` unless the file was also created with NC_STRICT_NC3. For straight-up netCDF-4 files, `nc_redef` is called automatically, as needed.

For all netCDF-4 files, the root `ncid` must be used. This is the `ncid` returned by `nc_open` and `nc_create`, and points to the root of the hierarchy tree for netCDF-4 files.

```
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. This error code will only be returned for classic and 64-bit offset format files.
- The specified netCDF dataset was opened for read-only.
- The specified netCDF ID does not refer to an open netCDF dataset.

Errors

`NC_NOERR` No error.

`NC_EBADID` Bad `ncid`.

`NC_EBADGRPID` The `ncid` must refer to the root group of the file, that is, the group returned by `nc_open` or `nc_create`. (see [Section 2.8 \[nc_open\], page 19](#) see [Section 2.5 \[nc_create\], page 13](#)).

`NC_EINDEFINE` Already in define mode.

`NC_EPERM` File is read-only.

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);
```

2.12 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 2.18 \[nc_set_fill\], page 32](#). The netCDF dataset is then placed in data mode, so variable data can be read or written.

It's not necessary to call `nc_enddef` for netCDF-4 files. With netCDF-4 files, `nc_enddef` is called when needed by the `netcdf-4` library. User calls to `nc_enddef` for netCDF-4 files still flush the metadata to disk.

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*](#).

For netCDF-4/HDF5 format files there are some variable settings (the compression, endianness, fletcher32 error correction, and fill value) which must be set (if they are going to be set at all) between the `nc_def_var` and the next `nc_enddef`. Once the `nc_enddef` is called, these settings can no longer be changed for a variable.

Usage

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

ncid NetCDF ID, from a previous call to `nc_open` or `nc_create`. If you use a group id, the `enddef` will apply to the entire file. That all, the `enddef` will not just end define mode in one group, but in the entire file.

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_NOCLONBER, &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);

```

2.13 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 2.18 \[nc.set_fill\], page 32](#). 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 `bufsize` (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);
```

2.14 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

For netCDF-4 files, the `ncid` of the root group must be passed into `nc_close`.

```
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.

`NC_NOERR` No error.

`NC_EBADID` Invalid id passed.

`NC_EBADGRPID` `ncid` did not contain the root group id of this file. (NetCDF-4 only).

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);
```

2.15 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);

```

2.16 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);
```

2.17 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);
}
```

2.18 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 6.30 \[Fill Values\]](#), page 133. 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.

Return Codes

NC_NOERR No error.

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

NC_EPERM The specified netCDF ID refers to a dataset open for read-only access.

NC_EINVAL 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 */

```

2.19 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 directly from `nc_create`, using the proper `cmode` flag. (see [Section 2.5 \[nc_create\]](#), page 13).

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 Valid formats include `NC_FORMAT_CLASSIC` (the default), `NC_FORMAT_64BIT`, and, if `-enable-netcdf-4` was used during configure, `NC_FORMAT_NETCDF4` and `NC_FORMAT_NETCDF4_CLASSIC`

`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.

Return Codes

`NC_NOERR` No error.

`NC_EINVAL`

Invalid format. Valid formats include `NC_FORMAT_CLASSIC`, `NC_FORMAT_64BIT`, and, if `-enable-netcdf-4` was used during configure, `NC_FORMAT_NETCDF4` and `NC_FORMAT_NETCDF4_CLASSIC`. Trying to set the default format to something else will result in an invalid argument error.

Example

Here is an example using `nc_set_default_format` to create the same file in four formats with the same `nc_create` call (from `libsrc4/tst_utf8.c`):

```
#include <netcdf.h>
...
int ncid, varid, dimids[NDIMS];
int f;

for (f = NC_FORMAT_CLASSIC; f < NC_FORMAT_NETCDF4_CLASSIC; f++)
{
    if (nc_set_default_format(f, NULL)) ERR;
    if (nc_create(FILE_NAME, NC_CLOBBER, &ncid)) ERR;
...
}
```


3 Groups

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

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

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

Variables are only visible in the group in which they are defined. The same applies to attributes. “Global” attributes are associated with the group whose `ncid` is used.

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

Group operations are only permitted on netCDF-4 files - that is, files created with the HDF5 flag in `nc_create`. (see [Section 2.5 \[nc_create\]](#), [page 13](#)). Groups are not compatible with the netCDF classic data model, so files created with the `NC_CLASSIC_MODEL` file cannot contain groups (except the root group).

3.1 Find a Group ID: `nc_inq_ncid`

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

Usage

```
int nc_inq_ncid(int ncid, char *name, int *grp_ncid);
```

ncid The group id for this operation.

name A char array that holds the name of the desired group.

grp_ncid An int pointer that will receive the group id, if the group is found.

Errors

`NC_NOERR` No error.

`NC_EBADID` Bad group id.

`NC_ENOTNC4` Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), [page 19](#)).

`NC_ESTRICTNC3` This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), [page 19](#)).

NC_EHDFERR

An error was reported by the HDF5 layer.

Example

```
int root_ncid, child_ncid;
char file[] = "nc4_test.nc";

/* Open the file. */
if ((res = nc_open(file, NC_NOWRITE, &root_ncid)))
    return res;

/* Get the ncid of an existing group. */
if ((res = nc_inq_ncid(root_ncid, "group1", &child_ncid)))
    return res;
```

3.2 Get a List of Groups in a Group: nc_inq_grps

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

Usage

```
int nc_inq_grps(int ncid, int *numgrps, int *ncids);
```

ncid The group id for this operation.

numgrps Pointer to an int which will get number of groups in this group. If NULL, it's ignored.

ncids Pointer to a already allocated array of ints which will receive the ids of all the groups in this group. If NULL, it's ignored. Call this function with NULL for ncids parameter to find out how many groups there are.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), [page 19](#)).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), [page 19](#)).

NC_EHDFERR An error was reported by the HDF5 layer.

Example

```
int root_ncid, numgrps;
int *ncids;
char file[] = "nc4_test.nc";

/* Open the file. */
if ((res = nc_open(file, NC_NOWRITE, &root_ncid)))
    return res;

/* Get a list of ncids for the root group. (That is, find out of
there are any groups already defined. */
if ((res = nc_inq_grps(root_ncid, &numgrps, NULL)))
    return res;
ncids = malloc(sizeof(int) * numgrps);
if ((res = nc_inq_grps(root_ncid, NULL, ncids)))
    return res;
```

3.3 Find all the Variables in a Group: nc_inq_varids

Find all varids for a location.

Usage

```
nc_inq_varids(int ncid, int *nvars, int *varids);
```

ncid The group id for this operation.

nvars The integer pointed to by this parameter will get the number of variable IDs found.

varids An already allocated array to store the list of varids. Ignored if NULL.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_EHDFERR An error was reported by the HDF5 layer.

Example

This example is from libsrc4/tst_vars.c.

```
int nvars_in, varids_in[2];

...

/* Open the file and make sure nc_inq_varids yeilds correct
 * result. */
if (nc_open(FILE_NAME, NC_NOWRITE, &ncid)) ERR;
if (nc_inq_varids(ncid, &nvars_in, varids_in)) ERR;
```

3.4 Find all Dimensions Visible in a Group: nc_inq_dimids

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

Usage

```
int nc_inq_dimids(int ncid, int *ndims, int *dimids, int include_parents);
```

ncid The group id for this operation.

dimids An already allocated array of ints when the dimids of the visible dimensions will be stashed. Use nc_inq_ndims to find out how many dims are visible from this group. (see [Section 2.15 \[nc_inq Family\]](#), page 28).

include_parents

If non-zero then all the dimensions in all parent groups will also be retrieved.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_EHDFERR An error was reported by the HDF5 layer.

Example

This example is from libsrc4/tst_dims.c.

```

int ncid, dimid;
int ndims_in, dimids_in[MAX_DIMS];

...

/* Open the file and make sure nc_inq_dimids yeilds correct
 * result. */
if (nc_open(FILE_NAME, NC_NOWRITE, &ncid)) ERR;
if (nc_inq_dimids(ncid, &ndims_in, dimids_in, 0)) ERR;
if (ndims_in != 1 || dimids_in[0] != 0) ERR;

```

3.5 Find a Group's Name: `nc_inq_grpname`

Given a group ID find its name. (Root group is named "/"). See [Section 3.6 \[nc_inq_grpname_full\]](#), page 42.

Usage

```
int nc_inq_grpname(int ncid, char *name);
```

ncid The group id for this operation.

name Pointer to allocated space of correct length. The name of the group will be copied there. The name will be less than NC_MAX_NAME, not including a terminating NULL byte.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_EHDFERR An error was reported by the HDF5 layer.

Example

This example is from libsrc4/tst_grps.c.

```

int grpid_in[MAX_SIBLING_GROUPS];
char name_in[NC_MAX_NAME + 1];

```

```
...
```

```
if (nc_inq_grpname(grpid_in[0], name_in)) ERR;
```

3.6 Find a Group's Full Name: `nc_inq_grpname_full`

Given `ncid`, find complete name of group. (Root group is named `"/"`, a full "path" for each group is provided in the name, with groups separated with a forward slash `/` as in Unix directory names. For example `"/group1/subgrp1/subsubgrp1"`)

Usage

```
int nc_inq_grpname_full(int ncid, size_t *lenp, char *full_name);
```

`ncid` The group id for this operation.

`lenp` Pointer to a `size_t` which will get the length. (Note that this does not include the NULL terminator byte.)

`full_name` Pointer to allocated space of correct length. The name of the group will be copied there. To find the required length, call `nc_inq_grpname_len` call this function with NULL for the `full_name` parameter, or call (see [Section 3.7 \[nc_inq_grpname_len\]](#), page 43).

Errors

`NC_NOERR` No error.

`NC_EBADID` Bad group id.

`NC_ENOTNC4` Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), page 19).

`NC_ESTRICNC3` This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

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

Example

This example is from the test program `libsrc4/tst_grps.c`.

```
int grpid_in[MAX_SIBLING_GROUPS];
char full_name_in[NC_MAX_NAME * 10];
size_t len;
```



```

...

if (nc_inq_grpname_full(grpid_in[0], &len, full_name_in)) ERR;

```

3.7 Find the Length of a Group's Full Name: `nc_inq_grpname_len`

Given `ncid`, find `len` of the full name, as returned by `nc_inq_grpname_full`. See [Section 3.6 \[nc_inq_grpname_full\]](#), page 42. (Root group is named "/", with length 1.)

Usage

```
int nc_inq_grpname_len(int ncid, size_t *lenp);
```

`ncid` The group id for this operation.

`lenp` Pointer to an int where the length will be placed. Length does not include NULL terminator.

Errors

`NC_NOERR` No error.

`NC_EBADID` Bad group id.

`NC_ENOTNC4` Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc_open\]](#), page 19).

`NC_ESTRICNC3` This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

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

Example

This example is from `libsrc4/tst_grps.c`.

```

int ncid;
size_t len;

...

if (nc_inq_grpname_len(ncid, &len)) ERR;

```

3.8 Find a Group's Parent: `nc_inq_grp_parent`

Given `ncid`, find the `ncid` of the parent group.

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

Usage

```
int nc_inq_grp_parent(int ncid, int *parent_ncid);
```

`ncid` The group id.

`parent_ncid`
Pointer to an int. The `ncid` of the group will be copied there.

Errors

`NC_NOERR` No error.

`NC_EBADID`
Bad group id.

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

`NC_ENOTNC4`
Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag `HDF5`. (see [Section 2.8 \[nc-open\]](#), page 19).

`NC_ESTRCTNC3`
This file was created with the strict `netcdf-3` flag, therefore `netcdf-4` operations are not allowed. (see [Section 2.8 \[nc-open\]](#), page 19).

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

Example

```
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_grp(ncid, HENRY_VII, &henry_vii_id)) ERR;

if (nc_inq_grp_parent(henry_vii_id, &parent_ncid)) ERR;
if (parent_ncid != ncid) ERR;
if (nc_close(ncid)) ERR;
```

3.9 Create a New Group: `nc_def_grp`

Create a group. Its location id is returned in the `new_ncid` pointer.

Usage

```
int nc_def_grp(int parent_ncid, char *name, int *new_ncid);
```

parent_ncid The group id of the parent group.

name The name of the new group.

new_ncid A pointer to an int. The ncid of the new group will be placed there.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENAMEINUSE That name is in use. Group names must be unique within a group, and must not be the same as any variable or type in the group.

NC_EMAXNAME Name exceed max length NC_MAX_NAME.

NC_EBADNAME Name contains illegal characters.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag HDF5. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_EHDFERR An error was reported by the HDF5 layer.

NC_EPERM Attempt to write to a read-only file.

NC_ENOTINDEFINE Not in define mode.

Example

```
int root_ncid, a1_ncid;
char grpname[] = "assimilation1";

/* Get the ncid of the root group. */
if ((res = nc_inq_ncid(root_ncid, NULL, &root_ncid)))
    return res;

/* Create a group. */
```

```
if ((res = nc_def_grp(root_ncid, grpname, &a1_ncid)))  
    return res;
```

4 Dimensions

4.1 Dimensions Introduction

Dimensions for a netCDF dataset are defined when it is created, while the netCDF dataset is in define mode. Additional dimensions may be added later by reentering define mode. A netCDF dimension has a name and a length. In a netCDF classic or 64-bit offset file, at most one dimension can have the unlimited length, which means variables using this dimension can grow along this dimension. In a netCDF-4 file multiple unlimited dimensions are supported.

There is a suggested limit (100) 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.

4.2 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 classic or 64-bit offset netCDF dataset. NetCDF-4 datasets may have multiple unlimited dimensions.

Usage

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

ncid NetCDF group ID, from a previous call to `nc_open`, `nc_create`, `nc_def_grp`, etc.

name Dimension name.

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

4.3 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

When searching for a dimension, the specified group is searched, and then its parent group, and then its grandparent group, etc., up to the root group.

```
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.

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);
```

4.4 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 <code>NC_MAX_NAME</code> . (This doesn't include the null terminator, so declare your array to be size <code>NC_MAX_NAME+1</code>). 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);
```

4.5 Rename a Dimension: `nc_rename_dim`

The function `nc_rename_dim` renames an existing dimension in a netCDF dataset open for writing. You cannot rename a dimension to have the same name as another dimension.

For netCDF classic and 64-bit offset files, if the new name is longer than the old name, the netCDF dataset must be in define mode.

For netCDF-4 files the dataset is switched to define mode for the rename, regardless of the name length.

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);
```

4.6 Find All Unlimited Dimension IDs: `nc_inq_unlimdims`

In netCDF-4 files, it's possible to have multiple unlimited dimensions. This function returns a list of the unlimited dimension ids visible in a group.

Dimensions are visible in a group if they have been defined in that group, or any ancestor group.

Usage

```
int nc_inq_unlimdims(int ncid, int *nunlimdimsp, int *unlimdimidsp);
```

ncid NetCDF group ID, from a previous call to `nc_open`, `nc_create`, `nc_def_grp`, etc.

nunlimdimsp
 A pointer to an int which will get the number of visible unlimited dimensions.
 Ignored if NULL.

unlimdimidsp
 A pointer to an already allocated array of int which will get the ids of all
 visible unlimited dimensions. Ignored if NULL. To allocate the correct length
 for this array, call `nc_inq_unlimdims` with a NULL for this parameter and use
 the `nunlimdimsp` parameter to get the number of visible unlimited dimensions.

Errors

NC_NOERR No error.

NC_EBADID
 Bad group id.

NC_ENOTNC4
 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations
 can only be performed on files defined with a create mode which includes flag
 HDF5. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_ESTRICNC3
 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations
 are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_EHDFERR
 An error was reported by the HDF5 layer.

Example

```
int root_ncid, num_unlimdims, unlimdims[NC_MAX_DIMS];
char file[] = "nc4_test.nc";
int res;

/* Open the file. */
if ((res = nc_open(file, NC_NOWRITE, &root_ncid)))
    return res;

/* Find out if there are any unlimited dimensions in the root
   group. */
if ((res = nc_inq_unlimdims(root_ncid, &num_unlimdims, unlimdims)))
    return res;

printf("nc_inq_unlimdims reports %d unlimited dimensions\n", num_unlimdims);
```

5 User Defined Data Types

5.1 User Defined Types Introduction

NetCDF-4 has added support for four different user defined data types. User defined type may only be used in files created with the NC_NETCDF4 and without NC_CLASSIC_MODEL.

compound type

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

variable length array type

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

opaque type

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

enum type

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

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

Once types are constructed, define variables of the new type with `nc_def_var` (see [Section 6.5 \[nc_def_var\]](#), [page 85](#)). Write to them with `nc_put_var1`, `nc_put_var`, `nc_put_vara`, or `nc_put_vars` (see [Chapter 6 \[Variables\]](#), [page 83](#)). Read data of user-defined type with `nc_get_var1`, `nc_get_var`, `nc_get_vara`, or `nc_get_vars` (see [Chapter 6 \[Variables\]](#), [page 83](#)).

Create attributes of the new type with `nc_put_att` (see [Section 7.2 \[nc_put_att_ type\]](#), [page 137](#)). Read attributes of the new type with `nc_get_att` (see [Section 7.4 \[nc_get_att_ type\]](#), [page 142](#)).

5.2 Learn the IDs of All Types in Group: `nc_inq_typeids`

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

Usage

```
int nc_inq_typeids(int ncid, int *ntypes, int *typeids);
```

`ncid` The group id.

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

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

Errors

`NC_NOERR` No error.

`NC_BADID` Bad `ncid`.

Example

The following example is from the test program `libsrc4/tst_enums.c`.

```
if (nc_open(FILE_NAME, NC_NOWRITE, &ncid)) ERR;

/* Get type info. */
if (nc_inq_typeids(ncid, &ntypes, typeids)) ERR;
if (ntypes != 1 || !typeids[0]) ERR;
```

5.3 Learn About a User Defined Type: `nc_inq_type`

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

For even more information about a user defined type [Section 5.4 \[nc_inq_user_type\]](#), [page 55](#).

Usage

```
nc_inq_type(int ncid, nc_type xtype, char *name, size_t *sizep);
```

<code>ncid</code>	The <code>ncid</code> for the group containing the type (ignored for atomic types).
<code>xtype</code>	The <code>typeid</code> for this type, as returned by <code>nc_def_compound</code> , <code>nc_def_opaque</code> , <code>nc_def_enum</code> , <code>nc_def_vlen</code> , or <code>nc_inq_var</code> , or as found in <code>netcdf.h</code> in the list of atomic types (<code>NC_CHAR</code> , <code>NC_INT</code> , etc.).
<code>name</code>	If non-NULL, the name of the user defined type will be copied here. It will be <code>NC_MAX_NAME</code> bytes or less. For atomic types, the type name from CDL will be given.
<code>sizep</code>	If non-NULL, the size of the type (in bytes) will be copied here. <code>VLEN</code> type size is the size of one element of the <code>VLEN</code> . String size is returned as zero, since it varies from string to string.

Return Codes

`NC_NOERR` No error.

`NC_EBADTYPEID`
Bad `typeid`.

`NC_ENOTNC4`
Seeking a user-defined type in a netCDF-3 file.

`NC_ESTRINCTNC3`
Seeking a user-defined type in a netCDF-4 file for which classic model has been turned on.

`NC_EBADGRPID`
Bad group ID in `ncid`.

`NC_EBADID`
Type ID not found.

NC_EHDFERR

An error was reported by the HDF5 layer.

Example

This example is from the test program `tst_enums.c`, and it uses all the possible inquiry functions on an enum type.

```
/* Check it out. */
if (nc_inq_user_type(ncid, typeids[0], name_in, &base_size_in, &base_nc_type_in,
                    &nfields_in, &class_in)) ERR;
if (strcmp(name_in, TYPE_NAME) || base_size_in != sizeof(int) ||
    base_nc_type_in != NC_INT || nfields_in != NUM_MEMBERS || class_in != NC_ENUM)
    ERR;
if (nc_inq_type(ncid, typeids[0], name_in, &base_size_in)) ERR;
if (strcmp(name_in, TYPE_NAME) || base_size_in != sizeof(int)) ERR;
if (nc_inq_enum(ncid, typeids[0], name_in, &base_nc_type, &base_size_in, &num_members)) ERR;
if (strcmp(name_in, TYPE_NAME) || base_nc_type != NC_INT || num_members != NUM_MEMBERS)
    ERR;
for (i = 0; i < NUM_MEMBERS; i++)
{
    if (nc_inq_enum_member(ncid, typeid, i, name_in, &value_in)) ERR;
    if (strcmp(name_in, member_name[i]) || value_in != member_value[i]) ERR;
    if (nc_inq_enum_ident(ncid, typeid, member_value[i], name_in)) ERR;
    if (strcmp(name_in, member_name[i])) ERR;
}

if (nc_close(ncid)) ERR;
```

5.4 Learn About a User Defined Type: `nc_inq_user_type`

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

Usage

```
nc_inq_user_type(int ncid, nc_type xtype, char *name, size_t *sizep,
                nc_type *base_nc_typep, size_t *nfieldsp, int *classp);
```

- | | |
|----------------------|--|
| ncid | The <code>ncid</code> for the group containing the user defined type. |
| xtype | The <code>typeid</code> for this type, as returned by <code>nc_def_compound</code> , <code>nc_def_opaque</code> , <code>nc_def_enum</code> , <code>nc_def_vlen</code> , or <code>nc_inq_var</code> . |
| name | If non-NULL, the name of the user defined type will be copied here. It will be NC_MAX_NAME bytes or less. |
| sizep | If non-NULL, the size of the user defined type will be copied here. |
| base_nc_typep | If non-NULL, the base <code>typeid</code> will be copied here for <code>vlen</code> and <code>enum</code> types. |
| nfieldsp | If non-NULL, the number of fields will be copied here for <code>enum</code> and <code>compound</code> types. |

classp Return the class of the user defined type, NC_VLEN, NC_OPAQUE, NC_ENUM, or NC_COMPOUND.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad typeid.

NC_EBADFIELDID
Bad fieldid.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

```
/* Create a file. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;

/* Create an enum type. */
if (nc_def_enum(ncid, NC_INT, TYPE_NAME, &typeid)) ERR;
for (i = 0; i < NUM_MEMBERS; i++)
    if (nc_insert_enum(ncid, typeid, member_name[i],
                      &member_value[i])) ERR;

/* Check it out. */
if (nc_inq_user_type(ncid, typeid, name_in, &base_size_in, &base_nc_type_in,
                    &nfields_in, &class_in)) ERR;
if (strcmp(name_in, TYPE_NAME) || base_size_in != sizeof(int) ||
    base_nc_type_in != NC_INT || nfields_in != NUM_MEMBERS || class_in != NC_ENUM)
```

5.5 Compound Types Introduction

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

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

To write data in a compound type, first use `nc_def_compound` to create the type, multiple calls to `nc_insert_compound` to add to the compound type, and then write data with the appropriate `nc_put_var1`, `nc_put_vara`, `nc_put_vars`, or `nc_put_varm` call.

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

5.6 Creating a Compound Type: `nc_def_compound`

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

After calling this function, fill out the type with repeated calls to `nc_insert_compound` (see [Section 5.7 \[nc_insert_compound\]](#), page 58). Call `nc_insert_compound` once for each field you wish to insert into the compound type.

Note that there does not seem to be a way to read such types into structures in Fortran 90 (and there are no structures in Fortran 77).

Usage

```
int nc_def_compound(int ncid, size_t size, char *name, nc_type *typeidp);
```

ncid The groupid where this compound type will be created.

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

name The name of the new compound type.

typeidp A pointer to an `nc_type`. The typeid of the new type will be placed there.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENAMEINUSE That name is in use.

NC_EMAXNAME Name exceeds max length `NC_MAX_NAME`.

NC_EBADNAME Name contains illegal characters.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag `NC_NETCDF4`. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\]](#), page 19).

NC_EHDFERR An error was reported by the HDF5 layer.

NC_EPERM Attempt to write to a read-only file.

NC_ENOTINDEFINE Not in define mode.

Example

```

struct s1
{
    int i1;
    int i2;
};
struct s1 data[DIM_LEN], data_in[DIM_LEN];

/* Create a file with a compound type. Write a little data. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_compound(ncid, sizeof(struct s1), SVC_REC, &typeid)) ERR;
if (nc_insert_compound(ncid, typeid, BATTLES_WITH_KLINGONS,
                      HOFFSET(struct s1, i1), NC_INT)) ERR;
if (nc_insert_compound(ncid, typeid, DATES_WITH_ALIENS,
                      HOFFSET(struct s1, i2), NC_INT)) ERR;
if (nc_def_dim(ncid, STARDATE, DIM_LEN, &dimid)) ERR;
if (nc_def_var(ncid, SERVICE_RECORD, typeid, 1, dimids, &varid)) ERR;
if (nc_put_var(ncid, varid, data)) ERR;
if (nc_close(ncid)) ERR;

```

5.7 Inserting a Field into a Compound Type: nc_insert_compound

Insert a named field into a compound type.

Usage

```

int nc_insert_compound(nc_type typeid, char *name, size_t offset,
                      nc_type field_typeid);

```

typeid The typeid for this compound type, as returned by nc_def_compound, or nc_inq_var.

name The name of the new field.

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

field_typeid
 The type of the field to be inserted.

Errors

NC_NOERR No error.

NC_EBADID
 Bad group id.

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

NC_EMAXNAME
 Name exceed max length NC_MAX_NAME.

NC_EBADNAME

Name contains illegal characters.

NC_ENOTNC4

Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NC_NETCDF4. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_ESTRICNC3

This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_EHDFERR

An error was reported by the HDF5 layer.

NC_ENOTINDEFINE

Not in define mode.

Example

5.8 Inserting an Array Field into a Compound Type: **nc_insert_array_compound**

Insert a named field into a compound type.

Usage

```
int nc_insert_array_compound(int ncid, nc_type xtype, char *name,
                             size_t offset, nc_type field_typeid,
                             int ndims, int *dim_sizes);
```

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

xtype The typeid for this compound type, as returned by `nc_def_compound`, or `nc_inq_var`.

name The name of the new field.

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

field_typeid
 The base type of the array to be inserted.

Errors

NC_NOERR No error.

NC_EBADID

Bad group id.

NC_ENAMEINUSE

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

NC_EMAXNAME

Name exceed max length NC_MAX_NAME.

NC_EBADNAME

Name contains illegal characters.

NC_ENOTNC4

Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NC_NETCDF4. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_ESTRICNC3

This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_EHDFERR

An error was reported by the HDF5 layer.

NC_ENOTINDEFINE

Not in define mode.

NC_ETYPEDEFINED

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

Example

This example comes from the test file libsrc4/tst_compounds.c, which writes data about some Star Fleet officers who are known to use netCDF data.

```
/* Since some aliens exists in different, or more than one,
 * dimensions, StarFleet keeps track of the dimensional abilities
 * of everyone on 7 dimensions. */
#define NUM_DIMENSIONS 7
struct dim_rec
{
    int starfleet_id;
    int abilities[NUM_DIMENSIONS];
};
struct dim_rec dim_data_out[DIM_LEN], dim_data_in[DIM_LEN];

/* Create some phoney data. */
for (i=0; i<DIM_LEN; i++)
{
    /* snip */
    /* Dimensional data. */
    dim_data_out[i].starfleet_id = i;
    for (j = 0; j < NUM_DIMENSIONS; j++)
        dim_data_out[i].abilities[j] = j;
    /* snip */
}
```

```

printf("*** testing compound variable containing an array of ints...");
{
    nc_type field_typeid;
    int dim_sizes[] = {NUM_DIMENSIONS};

    /* Create a file with a compound type which contains an array of
     * int. Write a little data. */
    if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
    if (nc_def_compound(ncid, sizeof(struct dim_rec), "SFDimRec", &typeid)) ERR;
    if (nc_insert_compound(ncid, typeid, "starfleet_id",
                          HOFFSET(struct dim_rec, starfleet_id), NC_INT)) ERR;
    if (nc_insert_array_compound(ncid, typeid, "abilities",
                                HOFFSET(struct dim_rec, abilities), NC_INT, 1, dim_sizes))
    if (nc_inq_compound_field(ncid, xtype, 1, name, &offset, &field_typeid,
                              &field_ndims, field_sizes)) ERR;
    if (strcmp(name, "abilities") || offset != 4 || field_typeid != NC_INT ||
        field_ndims != 1 || field_sizes[0] != dim_sizes[0]) ERR;
    if (nc_def_dim(ncid, STARDATE, DIM_LEN, &dimid)) ERR;
    if (nc_def_var(ncid, "dimension_data", typeid, 1, dimids, &varid)) ERR;
    if (nc_put_var(ncid, varid, dim_data_out)) ERR;
    if (nc_close(ncid)) ERR;

    /* Open the file and take a look. */
    if (nc_open(FILE_NAME, NC_WRITE, &ncid)) ERR;
    if (nc_inq_var(ncid, 0, name, &xtype, &ndims, dimids, &natts)) ERR;
    if (strcmp(name, "dimension_data") || ndims != 1 || natts != 0 || dimids[0] != 0)
    if (nc_inq_compound(ncid, xtype, name, &size, &nfields)) ERR;
    if (nfields != 2 || size != sizeof(struct dim_rec) || strcmp(name, "SFDimRec"))
    if (nc_inq_compound_field(ncid, xtype, 1, name, &offset, &field_typeid,
                              &field_ndims, field_sizes)) ERR;
    if (strcmp(name, "abilities") || offset != 4 || field_typeid != NC_INT ||
        field_ndims != 1 || field_sizes[0] != NUM_DIMENSIONS) ERR;
    if (nc_get_var(ncid, varid, dim_data_in)) ERR;
    for (i=0; i<DIM_LEN; i++)
    {
        if (dim_data_in[i].starfleet_id != dim_data_out[i].starfleet_id) ERR;
        for (j = 0; j < NUM_DIMENSIONS; j++)
            if (dim_data_in[i].abilities[j] != dim_data_out[i].abilities[j]) ERR;
    }
    if (nc_close(ncid)) ERR;
}

```

5.9 Learn About a Compound Type: nc_inq_compound

Get the number of fields, len, and name of a compound type.

Usage

```
int nc_inq_compound(int ncid, nc_type xtype, char *name, size_t *sizep,
                   size_t *nfieldsp);
```

ncid	The ID of any group in the file that contains the compound type.
xtype	The typeid for this compound type, as returned by <code>nc_def_compound</code> , or <code>nc_inq_var</code> .
name	Pointer to an already allocated char array which will get the name, as a null terminated string. It will have a maximum length of <code>NC_MAX_NAME + 1</code> . Ignored if NULL.
sizep	A pointer to a <code>size_t</code> . The size of the compound type will be put here. Ignored if NULL.
nfieldsp	A pointer to a <code>size_t</code> . The number of fields in the compound type will be placed here. Ignored if NULL.

Return Codes

NC_NOERR	No error.
NC_EBADID	Couldn't find this ncid.
NC_ENOTNC4	Not a netCDF-4/HDF5 file.
NC_ESTRICNC3	A netCDF-4/HDF5 file, but with <code>CLASSIC_MODEL</code> . No user defined types are allowed in the classic model.
NC_EBADTYPE	This type not a compound type.
NC_EBADTYPEID	Bad type id.
NC_EHDFERR	An error was reported by the HDF5 layer.

Example

The following example is from the test program `libsrc4/tst_compounds.c`. See also the example for See [Section 5.8 \[nc_insert_array_compound\]](#), page 59.

```
#define BATTLES_WITH_KLINGONS "Number_of_Battles_with_Klingons"
#define DATES_WITH_ALIENS "Dates_with_Alien_Hotties"

struct s1
{
    int i1;
    int i2;
```

```

};

/* Create a file with a compound type. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_compound(ncid, sizeof(struct s1), SVC_REC, &typeid)) ERR;
if (nc_inq_compound(ncid, typeid, name, &size, &nfields)) ERR;
if (nfields) ERR;
if (nc_insert_compound(ncid, typeid, BATTLES_WITH_KLINGONS,
                      HOFFSET(struct s1, i1), NC_INT)) ERR;
if (nc_insert_compound(ncid, typeid, DATES_WITH_ALIENS,
                      HOFFSET(struct s1, i2), NC_INT)) ERR;

/* Check the compound type. */
if (nc_inq_compound(ncid, xtype, name, &size, &nfields)) ERR;
if (nfields != 2 || size != 8 || strcmp(name, SVC_REC)) ERR;
if (nc_inq_compound_name(ncid, xtype, name)) ERR;
if (strcmp(name, SVC_REC)) ERR;
if (nc_inq_compound_size(ncid, xtype, &size)) ERR;
if (size != 8) ERR;
if (nc_inq_compound_nfields(ncid, xtype, &nfields)) ERR;
if (nfields != 2) ERR;
if (nc_inq_compound_field(ncid, xtype, 0, name, &offset, &field_xtype, &field_ndims)
if (strcmp(name, BATTLES_WITH_KLINGONS) || offset != 0 || (field_xtype != NC_INT ||
if (nc_inq_compound_field(ncid, xtype, 1, name, &offset, &field_xtype, &field_ndims)
if (strcmp(name, DATES_WITH_ALIENS) || offset != 4 || field_xtype != NC_INT) ERR;
if (nc_inq_compound_fieldname(ncid, xtype, 1, name)) ERR;
if (strcmp(name, DATES_WITH_ALIENS)) ERR;
if (nc_inq_compound_fieldindex(ncid, xtype, BATTLES_WITH_KLINGONS, &fieldid)) ERR;
if (fieldid != 0) ERR;
if (nc_inq_compound_fieldoffset(ncid, xtype, 1, &offset)) ERR;
if (offset != 4) ERR;
if (nc_inq_compound_fieldtype(ncid, xtype, 1, &field_xtype)) ERR;
if (field_xtype != NC_INT) ERR;

```

5.10 Learn the Name of a Compound Type:

`nc_inq_compound_name`

Get the name of a compound type.

Usage

```
int nc_inq_compound_name(int ncid, nc_type xtype, char *name);
```

<code>ncid</code>	The groupid where this compound type exists.
<code>xtype</code>	The typeid for this compound type.
<code>name</code>	Pointer to an already allocated char array which will get the name, as a null terminated string. It will have a maximum length of NC_MAX_NAME+1.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad type id.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.11 Learn the Size of a Compound Type: `nc_inq_compound_size`

Get the len of a compound type.

Usage

```
int nc_inq_compound_size(int ncid, nc_type xtype, size_t *sizep);
```

ncid The groupid where this compound type exists.

xtype The typeid for this compound type.

size A pointer, which, if not NULL, get the size of the compound type.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad type id.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.12 Learn the Number of Fields in a Compound Type: `nc_inq_compound_nfields`

Get the number of fields of a compound type.

Usage

```
nc_inq_compound_nfields(int ncid, nc_type xtype, size_t *nfieldsp);
```

ncid The groupid where this compound type exists.

xtype The typeid for this compound type.

nfieldsp A pointer, which, if not NULL, get the number of fields of the compound type.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad type id.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.13 Learn About a Field of a Compound Type: nc_inq_compound_field

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

Usage

```
int nc_inq_compound_field(int ncid, nc_type xtype, int fieldid, char *name,
                          size_t *offsetp, nc_type *field_typeidp, int *ndimsp,
                          int *dim_sizesp);
```

ncid The groupid where this compound type exists.

xtype The typeid for this compound type, as returned by nc_def_compound, or nc_inq_var.

fieldid A zero-based index number specifying a field in the compound type.

name A pointer which, if non-NULL, will get the name of the field.

offsetp A pointer which, if non-NULL, will get the offset of the field.

field_typeid
A pointer which, if non-NULL, will get the typeid of the field.

ndimsp A pointer which, if non-NULL, will get the number of dimensions of the field.

dim_sizesp
A pointer which, if non-NULL, will get the dimension sizes of the field.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad type id.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.14 Find the Name of a Field in a Compound Type: `nc_inq_compound_fieldname`

Given the typeid and the fieldid, get the name.

Usage

```
int nc_inq_compound_fieldname(nc_type typeid, int fieldid, char *name);
```

typeid The typeid for this compound type, as returned by `nc_def_compound`, or `nc_inq_var`.

fieldid The id of the field in the compound type. Fields are numbered starting with 0 for the first inserted field.

name Pointer to an already allocated char array which will get the name, as a null terminated string. It will have a maximum length of `NC_MAX_NAME+1`.

Errors

`NC_NOERR` No error.

`NC_EBADTYPEID`
Bad type id.

`NC_EBADFIELDID`
Bad field id.

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

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.15 Get the FieldID of a Compound Type Field: `nc_inq_compound_fieldindex`

Given the typeid and the name, get the fieldid.

Usage

```
int nc_inq_compound_fieldindex(nc_type typeid, char *name, int *fieldidp);
```

typeid The typeid for this compound type, as returned by `nc_def_compound`, or `nc_inq_var`.

name The name of the field.

fieldidp A pointer to an int which will get the field id of the named field.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad type id.

NC_EUNKNAME
Can't find field of this name.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.16 Get the Offset of a Field: `nc_inq_compound_fieldoffset`

Given the typeid and fieldid, get the offset.

Usage

```
int nc_inq_compound_fieldoffset(nc_type typeid, int fieldid,
                               size_t *offsetp);
```

typeid The typeid for this compound type, as returned by `nc_def_compound`, or `nc_inq_var`.

fieldid The id of the field in the compound type. Fields are numbered starting with 0 for the first inserted field.

offsetp A pointer to a `size_t`. The offset of the field will be placed there.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad typeid.

NC_EBADFIELDID
Bad fieldid.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.17 Find the Type of a Field: `nc_inq_compound_fieldtype`

Given the typeid and the fieldid, get the type of that field.

Usage

```
nc_inq_compound_fieldtype(nc_type typeid, int fieldid, nc_type *field_typeidp);
```

typeid The typeid for this compound type, as returned by `nc_def_compound`, or `nc_inq_var`.

fieldid The id of the field in the compound type. Fields are numbered starting with 0 for the first inserted field.

field_typeidp
Pointer to a `nc_type` which will get the typeid of the field.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad typeid.

NC_EBADFIELDID
Bad fieldid.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.18 Find the Number of Dimensions in an Array Field: `nc_inq_compound_fieldndims`

Given the typeid and the fieldid, get the number of dimensions of that field.

Usage

```
int nc_inq_compound_fieldndims(int ncid, nc_type xtype, int fieldid,
                               int *ndimsp);
```

ncid The file or group ID.

xtype The typeid for this compound type, as returned by `nc_def_compound`, or `nc_inq_var`.

fieldid The id of the field in the compound type. Fields are numbered starting with 0 for the first inserted field.

ndimsp Pointer to an int which will get the number of dimensions of the field. Non-array fields have 0 dimensions.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad typeid.

NC_EBADFIELDID
Bad fieldid.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.19 Find the Sizes of Dimensions in an Array Field: nc_inq_compound_fielddim_sizes

Given the xtype and the fieldid, get the sizes of dimensions for that field. User must have allocated storage for the dim_sizes.

Usage

```
int nc_inq_compound_fielddim_sizes(int ncid, nc_type xtype, int fieldid,
                                   int *dim_sizes);
```

ncid The file or group ID.

xtype The typeid for this compound type, as returned by nc_def_compound, or nc_inq_var.

fieldid The id of the field in the compound type. Fields are numbered starting with 0 for the first inserted field.

dim_sizesp
Pointer to an array of int which will get the sizes of the dimensions of the field. Non-array fields have 0 dimensions.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad typeid.

NC_EBADFIELDID
Bad fieldid.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

See the example section for [Section 5.9 \[nc_inq_compound\]](#), page 61.

5.20 Variable Length Array Introduction

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

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

So an array of VLEN in C is an array of `nc_vlen_t` structures.

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

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

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

When dynamically allocating space to hold an array of VLEN, allocate storage for an array of `nc_vlen_t`.

5.21 Define a Variable Length Array (VLEN): `nc_def_vlen`

Use this function to define a variable length array type.

Usage

```
nc_def_vlen(int ncid, char *name, nc_type base_typeid, nc_type *xtypep);
```

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

name A name for the VLEN type.

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

xtypep A pointer to an `nc_type` variable. The typeid of the new VLEN type will be set here.

Errors

NC_NOERR No error.

NC_EMAXNAME `NC_MAX_NAME` exceeded.

NC_ENAMEINUSE Name is already in use.

NC_EBADNAME Attribute or variable name contains illegal characters.

NC_EBADID
ncid invalid.

NC_EBADGRPID
Group ID part of ncid was invalid.

NC_EINVAL
Size is invalid.

NC_ENOMEM
Out of memory.

Example

```
#define DIM_LEN 3
#define ATT_NAME "att_name"

nc_vlen_t data[DIM_LEN];
int *phony;

/* Create phony data. */
for (i=0; i<DIM_LEN; i++)
{
    if (!(phony = malloc(sizeof(int) * i+1)))
        return NC_ENOMEM;
    for (j=0; j<i+1; j++)
        phony[j] = -99;
    data[i].p = phony;
    data[i].len = i+1;
}

/* Define a VLEN of NC_INT, and write an attribute of that
type. */
if (nc_def_vlen(ncid, "name1", NC_INT, &typeid)) ERR;
if (nc_put_att(ncid, NC_GLOBAL, ATT_NAME, typeid, DIM_LEN, data)) ERR;
```

5.22 Learning about a Variable Length Array (VLEN)

Type: nc_inq_vlen

Use this type to learn about a vlen.

Usage

```
nc_inq_vlen(int ncid, nc_type xtype, char *name, size_t *datum_sizep,
            nc_type *base_nc_typep);
```

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

xtype The type of the VLEN to inquire about.

name A pointer for storage for the types name. The name will be NC_MAX_NAME characters or less.

`datum_sizep`

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

`base_nc_typep`

A pointer to an `nc_type`, this will get the type of the VLEN base type. (In other words, what type is this a VLEN of?)

Errors

`NC_NOERR` No error.

`NC_EBADTYPE`

Can't find the typeid.

`NC_EBADID`

ncid invalid.

`NC_EBADGRPID`

Group ID part of ncid was invalid.

Example

```
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_vlen(ncid, "name1", NC_INT, &typeid)) ERR;
if (nc_inq_vlen(ncid, typeid, name_in, &size_in, &base_nc_type_in)) ERR;
if (base_nc_type_in != NC_INT || (size_in != sizeof(int) || strcmp(name_in, VLEN
if (nc_inq_user_type(ncid, typeid, name_in, &size_in, &base_nc_type_in, NULL, &c
if (base_nc_type_in != NC_INT || (size_in != sizeof(int) || strcmp(name_in, VLEN
if (nc_inq_compound(ncid, typeid, name_in, &size_in, NULL) != NC_EBADTYPE) ERR;
if (nc_put_att(ncid, NC_GLOBAL, ATT_NAME, typeid, DIM_LEN, data)) ERR;
if (nc_close(ncid)) ERR;
```

5.23 Releasing Memory for a Variable Length Array (VLEN) Type: `nc_free_vlen`

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

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

To save the user the trouble calling `free()` on each element of the VLEN array (i.e. the array of arrays), the `nc_free_vlen` function is provided.

Usage

```
int nc_free_vlen(nc_vlen_t *vl);
```

`vl` A pointer to the variable length array structure which is to be freed.

Errors

NC_NOERR No error.

NC_EBADTYPE
Can't find the typeid.

Example

This example is from test program libsrc4/tst_vl.c.

```
/* Free the memory used in our phony data. */
for (i=0; i<DIM_LEN; i++)
    if (nc_free_vlen(&data[i])) ERR;
```

5.24 Opaque Type Introduction

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

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

To use an opaque type, first define it with [Section 5.25 \[nc_def_opaque\]](#), page 73. If encountering an enum type in a new data file, use [Section 5.26 \[nc_inq_opaque\]](#), page 74 to learn its name and size.

5.25 Creating Opaque Types: nc_def_opaque

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

Usage

```
nc_def_opaque(int ncid, char *name, size_t size, nc_type *typeidp);
```

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

name The name for this type. Must be shorter than NC_MAX_NAME.

size The size of each opaque object.

typeidp Pointer where the new typeid for this type is returned. Use this typeid when defining variables of this type with [Section 6.5 \[nc_def_var\]](#), page 85.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad typeid.

NC_EBADFIELDID
Bad fieldid.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

This example is from the test program `libsrc4/tst_opaques.c`.

```
/* Create a file that has an opaque attribute. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_opaque(ncid, BASE_SIZE, TYPE_NAME, &xtype)) ERR;
```

5.26 Learn About an Opaque Type: `nc_inq_opaque`

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

Usage

```
int nc_inq_opaque(int ncid, nc_type xtype, char *name, size_t *sizep);
```

<code>ncid</code>	The <code>ncid</code> for the group containing the opaque type.
<code>xtype</code>	The <code>typeid</code> for this opaque type, as returned by <code>nc_def_compound</code> , or <code>nc_inq_var</code> .
<code>name</code>	If non-NULL, the name of the opaque type will be copied here. It will be NC_MAX_NAME bytes or less.
<code>sizep</code>	If non-NULL, the size of the opaque type will be copied here.

Errors

NC_NOERR	No error.
NC_EBADTYPEID	Bad <code>typeid</code> .
NC_EBADFIELDID	Bad <code>fieldid</code> .
NC_EHDFERR	An error was reported by the HDF5 layer.

Example

This example is from test program `libsrc4/tst_opaques.c`:

```
if (nc_def_opaque(ncid, BASE_SIZE, TYPE_NAME, &xtype)) ERR;

if (nc_inq_opaque(ncid, xtype, name_in, &base_size_in)) ERR;
if (strcmp(name_in, TYPE_NAME) || base_size_in != BASE_SIZE) ERR;
```

5.27 Enum Type Introduction

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

5.28 Creating a Enum Type: `nc_def_enum`

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

After calling this function, fill out the type with repeated calls to `nc_insert_enum` (see [Section 5.29 \[nc_insert_enum\], page 77](#)). Call `nc_insert_enum` once for each value you wish to make part of the enumeration.

Usage

```
int nc_def_enum(int ncid, nc_type base_typeid, const char *name,
               nc_type *typeidp);
```

ncid The groupid where this compound type will be created.

base_typeid The base integer type for this enum. Must be one of: `NC_BYTE`, `NC_UBYTE`, `NC_SHORT`, `NC_USHORT`, `NC_INT`, `NC_UINT`, `NC_INT64`, `NC_UINT64`.

name The name of the new enum type.

typeidp A pointer to an `nc_type`. The typeid of the new type will be placed there.

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENAMEINUSE That name is in use.

NC_EMAXNAME Name exceeds max length `NC_MAX_NAME`.

NC_EBADNAME Name contains illegal characters.

NC_ENOTNC4 Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag `NC_NETCDF4`. (see [Section 2.8 \[nc_open\], page 19](#)).

NC_ESTRICNC3 This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc_open\], page 19](#)).

NC_EHDFERR An error was reported by the HDF5 layer.

NC_EPERM Attempt to write to a read-only file.

NC_ENOTINDEFINE Not in define mode.

The following example, from `libsrc4/tst_enums.c`, shows the creation and use of an enum type, including the use of a fill value.

```

int dimid, varid;
size_t num_members_in;
int class_in;
unsigned char value_in;

enum clouds {          /* a C enumeration */
    CLEAR=0,
    CUMULONIMBUS=1,
    STRATUS=2,
    STRATOCUMULUS=3,
    CUMULUS=4,
    ALTOSTRATUS=5,
    NIMBOSTRATUS=6,
    ALTOCUMULUS=7,
    CIRROSTRATUS=8,
    CIRROCUMULUS=9,
    CIRRUS=10,
    MISSING=255};

struct {
    char *name;
    unsigned char value;
} cloud_types[] = {
    {"Clear", CLEAR},
    {"Cumulonimbus", CUMULONIMBUS},
    {"Stratus", STRATUS},
    {"Stratocumulus", STRATOCUMULUS},
    {"Cumulus", CUMULUS},
    {"Altostratus", ALTOSTRATUS},
    {"Nimbostratus", NIMBOSTRATUS},
    {"Alto cumulus", ALTOCUMULUS},
    {"Cirrostratus", CIRROSTRATUS},
    {"Cirro cumulus", CIRROCUMULUS},
    {"Cirrus", CIRRUS},
    {"Missing", MISSING}
};

int var_dims[VAR2_RANK];
unsigned char att_val;
unsigned char cloud_data[DIM2_LEN] = {
    CLEAR, STRATUS, CLEAR, CUMULONIMBUS, MISSING};
unsigned char cloud_data_in[DIM2_LEN];

if (nc_create(FILE_NAME, NC_CLOBBER | NC_NETCDF4, &ncid)) ERR;

```

```

/* Create an enum type. */
if (nc_def_enum(ncid, NC_UBYTE, TYPE2_NAME, &typeid)) ERR;
num_members = (sizeof cloud_types) / (sizeof cloud_types[0]);
for (i = 0; i < num_members; i++)
    if (nc_insert_enum(ncid, typeid, cloud_types[i].name,
                      &cloud_types[i].value)) ERR;

/* Declare a station dimension */
if (nc_def_dim(ncid, DIM2_NAME, DIM2_LEN, &dimid)) ERR;
/* Declare a variable of the enum type */
var_dims[0] = dimid;
if (nc_def_var(ncid, VAR2_NAME, typeid, VAR2_RANK, var_dims, &varid)) ERR;
/* Create and write a variable attribute of the enum type */
att_val = MISSING;
if (nc_put_att(ncid, varid, ATT2_NAME, typeid, ATT2_LEN, &att_val)) ERR;
if (nc_enddef(ncid)) ERR;
/* Store some data of the enum type */
if(nc_put_var(ncid, varid, cloud_data)) ERR;
/* Write the file. */
if (nc_close(ncid)) ERR;

```

5.29 Inserting a Field into a Enum Type: nc_insert_enum

Insert a named member into a enum type.

Usage

```
int nc_insert_enum(int ncid, nc_type xtype, const char *identifier,
                  const void *value);
```

ncid The ncid of the group which contains the type.

typeid The typeid for this enum type, as returned by nc_def_enum, or nc_inq_var.

identifier The identifier of the new member.

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

Errors

NC_NOERR No error.

NC_EBADID Bad group id.

NC_ENAMEINUSE That name is in use. Field names must be unique within a enum type.

NC_EMAXNAME Name exceed max length NC_MAX_NAME.

NC_EBADNAME

Name contains illegal characters.

NC_ENOTNC4

Attempting a netCDF-4 operation on a netCDF-3 file. NetCDF-4 operations can only be performed on files defined with a create mode which includes flag NC_NETCDF4. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_ESTRICNC3

This file was created with the strict netcdf-3 flag, therefore netcdf-4 operations are not allowed. (see [Section 2.8 \[nc-open\]](#), page 19).

NC_EHDFERR

An error was reported by the HDF5 layer.

NC_ENOTINDEFINE

Not in define mode.

Example

This example is from libsrc4/tst_enums.c; also see the example in [See Section 5.28 \[nc-def_enum\]](#), page 75.

```
char brady_name[NUM_BRADYS][NC_MAX_NAME + 1] = {"Mike", "Carol", "Greg", "Marsha",
                                                "Peter", "Jan", "Bobby", "Whats",
                                                "Alice"};

unsigned char brady_value[NUM_BRADYS] = {0, 1,2,3,4,5,6,7,8};
unsigned char data[BRADY_DIM_LEN] = {0, 4, 8};
unsigned char value_in;

/* Create a file. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;

/* Create an enum type based on unsigned bytes. */
if (nc_def_enum(ncid, NC_UBYTE, BRADYS, &typeid)) ERR;
for (i = 0; i < NUM_BRADYS; i++)
    if (nc_insert_enum(ncid, typeid, brady_name[i],
                      &brady_value[i])) ERR;
```

5.30 Learn About a Enum Type: nc_inq_enum

Get information about a user-define enumeration type.

Usage

```
nc_inq_enum(int ncid, nc_type xtype, char *name, nc_type *base_nc_typep,
            size_t *base_sizep, size_t *num_membersp);
```

ncid The group ID of the group which holds the enum type.

xtype The typeid for this enum type, as returned by nc_def_enum, or nc_inq_var.

name Pointer to an already allocated char array which will get the name, as a null terminated string. It will have a maximum length of NC_MAX_NAME+1.

base_nc_type If non-NULL, a pointer to a nc_type, which will get the base integer type of this enum.

base_size If non-NULL, a size_t pointer, which will get the size (in bytes) of the base integer type of this enum.

num_members If non-NULL, a size_t pointer which will get the number of members defined for this enumeration type.

Errors

NC_NOERR No error.

NC_EBADTYPEID Bad type id.

NC_EHDFERR An error was reported by the HDF5 layer.

Example

This example is from libsrc4/tst_enums.c, and is a continuation of the example above for nc_insert_enum. First an enum type is created, with one element for each of the nine members of the Brady family on a popular American television show which occupies far too much memory space in my brain!

In the example, the enum type is created, then checked using the nc_inq_enum and nc_inq_enum_member functions. See [Section 5.31 \[nc_inq_enum_member\]](#), page 80.

```
char brady_name[NUM_BRADYS][NC_MAX_NAME + 1] = {"Mike", "Carol", "Greg", "Marsha",
                                                "Peter", "Jan", "Bobby", "Whats",
                                                "Alice"};

unsigned char brady_value[NUM_BRADYS] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
unsigned char data[BRADY_DIM_LEN] = {0, 4, 8};
unsigned char value_in;

/* Create a file. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;

/* Create an enum type based on unsigned bytes. */
if (nc_def_enum(ncid, NC_UBYTE, BRADYS, &typeid)) ERR;
for (i = 0; i < NUM_BRADYS; i++)
    if (nc_insert_enum(ncid, typeid, brady_name[i],
                      &brady_value[i])) ERR;

/* Check it out. */
```

```

if (nc_inq_enum(ncid, typeid, name_in, &base_nc_type, &base_size_in, &num_member
if (strcmp(name_in, BRADYS) || base_nc_type != NC_UBYTE || base_size_in != 1 ||
    num_members != NUM_BRADYS) ERR;
for (i = 0; i < NUM_BRADYS; i++)
{
    if (nc_inq_enum_member(ncid, typeid, i, name_in, &value_in)) ERR;
    if (strcmp(name_in, brady_name[i]) || value_in != brady_value[i]) ERR;
    if (nc_inq_enum_ident(ncid, typeid, brady_value[i], name_in)) ERR;
    if (strcmp(name_in, brady_name[i])) ERR;
}

```

5.31 Learn the Name of a Enum Type: `nc_inq_enum_member`

Get information about a member of an enum type.

Usage

```

int nc_inq_enum_member(int ncid, nc_type xtype, int idx, char *name,
                      void *value);

```

<code>ncid</code>	The groupid where this enum type exists.
<code>xtype</code>	The typeid for this enum type.
<code>idx</code>	The zero-based index number for the member of interest.
<code>name</code>	If non-NULL, a pointer to an already allocated char array which will get the name, as a null terminated string. It will have a maximum length of NC_MAX_NAME+1.
<code>value</code>	If non-NULL, a pointer to storage of the correct integer type (i.e. the base type of this enum type). It will get the value associated with this member.

Errors

NC_NOERR No error.

NC_EBADTYPEID
Bad type id.

NC_EHDFERR
An error was reported by the HDF5 layer.

Example

This is the continuation of the example in [Section 5.28 \[nc_def_enum\]](#), [page 75](#). The file is reopened and the cloud enum type is examined.

```

/* Reopen the file. */
if (nc_open(FILE_NAME, NC_NOWRITE, &ncid)) ERR;

```

```

    if (nc_inq_user_type(ncid, typeid, name_in, &base_size_in, &base_nc_type_in,
                        &nfields_in, &class_in)) ERR;
    if (strcmp(name_in, TYPE2_NAME) ||
        base_size_in != sizeof(unsigned char) ||
        base_nc_type_in != NC_UBYTE ||
        nfields_in != num_members ||
        class_in != NC_ENUM) ERR;
    if (nc_inq_enum(ncid, typeid, name_in,
                   &base_nc_type_in, &base_size_in, &num_members_in)) ERR;
    if (strcmp(name_in, TYPE2_NAME) ||
        base_nc_type_in != NC_UBYTE ||
        num_members_in != num_members) ERR;
    for (i = 0; i < num_members; i++)
    {
        if (nc_inq_enum_member(ncid, typeid, i, name_in, &value_in)) ERR;
        if (strcmp(name_in, cloud_types[i].name) ||
            value_in != cloud_types[i].value) ERR;
        if (nc_inq_enum_ident(ncid, typeid, cloud_types[i].value,
                             name_in)) ERR;
        if (strcmp(name_in, cloud_types[i].name)) ERR;
    }
    if (nc_inq_varid(ncid, VAR2_NAME, &varid)) ERR;

    if (nc_get_att(ncid, varid, ATT2_NAME, &value_in)) ERR;
    if (value_in != MISSING) ERR;

    if(nc_get_var(ncid, varid, cloud_data_in)) ERR;
    for (i = 0; i < DIM2_LEN; i++) {
        if (cloud_data_in[i] != cloud_data[i]) ERR;
    }

    if (nc_close(ncid)) ERR;

```

5.32 Learn the Name of a Enum Type: nc_inq_enum_ident

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

Usage

```
int nc_inq_enum_ident(int ncid, nc_type xtype, long long value, char *identifier);
```

<code>ncid</code>	The groupid where this enum type exists.
<code>xtype</code>	The typeid for this enum type.
<code>value</code>	The value for which an identifier is sought.

`identifier`

If non-NULL, a pointer to an already allocated char array which will get the identifier, as a null terminated string. It will have a maximum length of `NC_MAX_NAME+1`.

Return Code

`NC_NOERR` No error.

`NC_EBADTYPEID`

Bad type id, or not an enum type.

`NC_EHDFERR`

An error was reported by the HDF5 layer.

`NC_EINVAL`

The value was not found in the enum.

Example

See the example section for [Section 5.30 \[nc_inq_enum\]](#), page 78 for a full example.

6 Variables

6.1 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 7 \[Attributes\]](#), [page 137](#)) may be associated with a variable to specify such properties as units.

Operations supported on variables are:

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

6.2 Language Types Corresponding to netCDF external data types

NetCDF supported six atomic data types through version 3.6.0 (char, byte, short, int, float, and double). Starting with version 4.0, many new atomic and user defined data types are supported (unsigned int types, strings, compound types, variable length arrays, enums, opaque).

The additional data types are only supported in netCDF-4/HDF5 files. To create netCDF-4/HDF5 files, use the HDF5 flag in `nc_create`. (see [Section 2.5 \[nc_create\]](#), [page 13](#)).

6.3 NetCDF-3 Classic and 64-Bit Offset Data Types

NetCDF-3 classic and 64-bit offset files support 6 atomic data types, and none of the user defined datatype introduced in NetCDF-4.

The following table gives the netCDF-3 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 pre-processor macro for use in netCDF functions (the pre-processor 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.

6.4 NetCDF-4 Atomic Types

NetCDF-4 files support all of the atomic data types from netCDF-3, plus additional unsigned integer types, 64-bit integer types, and a string type.

Type	C #define	Bits
byte	NC_BYTE	8
unsigned byte	NC_UBYTE^	8
char	NC_CHAR	8
short	NC_SHORT	16
unsigned short	NC_USHORT^	16
int	NC_INT	32
unsigned int	NC_UINT^	32
unsigned long long	NC_UINT64^	64

long long	NC_INT64 [^]	64
float	NC_FLOAT	32
double	NC_DOUBLE	64
char **	NC_STRING [^]	string length + 1

[^]This type was introduced in netCDF-4, and is not supported in netCDF classic or 64-bit offset format files, or in netCDF-4 files if they are created with the NC_CLASSIC_MODEL flags.

6.5 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.

The total size the chunk must be less than 4 GiB. That is, the product of all chunk sizes and the size of the data (or the size of `nc_vlen_t` for VLEN types) must be less than 4 GiB.

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 <code>nc_open</code> or <code>nc_create</code> .
name	Variable name.
xtype	One of the set of predefined netCDF external data types. The type of this parameter, <code>nc_type</code> , is defined in the netCDF header file. The valid netCDF external data types are <code>NC_BYTE</code> , <code>NC_CHAR</code> , <code>NC_SHORT</code> , <code>NC_INT</code> , <code>NC_FLOAT</code> , and <code>NC_DOUBLE</code> . If the file is a NetCDF-4/HDF5 file, the additional types <code>NC_UBYTE</code> , <code>NC_USHORT</code> , <code>NC_UINT</code> , <code>NC_INT64</code> , <code>NC_UINT64</code> , and <code>NC_STRING</code> may be used, as well as a user defined type ID.
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 <code>NC_MAX_VAR_DIMS</code> .
dimids	Vector of <code>ndims</code> dimension IDs corresponding to the variable dimensions. For classic model netCDF files, if the ID of the unlimited dimension is included, it must be first. This argument is ignored if <code>ndims</code> is 0. For expanded model netCDF4/HDF5 files, there may be any number of unlimited dimensions, and they may be used in any element of the <code>dimids</code> array.
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:

`NC_NOERR` No error.

`NC_BADID` Bad `ncid`.

`NC_ENOTINDEFINE`

Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with `NC_STRICT_NC3` flag. (see [Section 2.5 \[nc_create\]](#), page 13).

`NC_ESTRICNC3`

Trying to create a var some place other than the root group in a netCDF file with `NC_STRICT_NC3` turned on.

`NC_MAX_VARS`

Max number of variables exceeded in a classic or 64-bit offset file, or an netCDF-4 file with `NC_STRICT_NC3` on.

`NC_EBADTYPE`

Bad type.

`NC_EINVAL`

Number of dimensions too large.

`NC_ENAMEINUSE`

Name already in use.

`NC_EPERM` Attempt to create object in read-only file.

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_NOCLIBBER, &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);

```

6.6 Define Chunking Parameters for a Variable: `nc_def_var_chunking`

The function `nc_def_var_chunking` sets the chunking parameters for a variable in a netCDF-4 file. It can set the chunk sizes to get chunked storage, or it can set the contiguous flag to get contiguous storage.

The total size of a chunk must be less than 4 GiB. That is, the product of all chunk sizes and the size of the data (or the size of `nc_vlen_t` for VLEN types) must be less than 4 GiB.

This function may only be called after the variable is defined, but before `nc_enddef` is called. Once the chunking parameters are set for a variable, they cannot be changed.

Note that this does not work for scalar variables. Only non-scalar variables can have chunking.

Usage

```
int nc_def_var_chunking(int ncid, int varid, int storage, size_t *chunksizep);
```

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

varid Variable ID.

storage If `NC_CONTIGUOUS`, then contiguous storage is used for this variable. Variables with one or more unlimited dimensions cannot use contiguous storage. If contiguous storage is turned on, the `chunksizep` parameter is ignored.

If `NC_CHUNKED`, then chunked storage is used for this variable. Chunk sizes may be specified with the `chunksizep` parameter or default sizes will be used if that parameter is `NULL`.

***chunksizep**

A pointer to an array list of chunk sizes. The array must have one chunk size for each dimension of the variable. If `NC_CONTIGUOUS` storage is set, then the `chunksizep` parameter is ignored.

Errors

`nc_def_var_chunking` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

`NC_NOERR` No error.

NC_BADID Bad ncid.

NC_ENOTNC4
Not a netCDF-4 file.

NC_ENOTVAR
Can't find this variable.

NC_ELATEDEF
This variable has already been the subject of a `nc_enddef` call. In netCDF-4 files `nc_enddef` will be called automatically for any data read or write. Once `nc_enddef` has been called after the `nc_def_var` call for a variable, it is impossible to set the chunking for that variable.

NC_ENOTINDEFINE
Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with NC_STRICT_NC3 flag. (see [Section 2.5 \[nc_create\]](#), page 13).

NC_ESTRICNC3
Trying to create a var some place other than the root group in a netCDF file with NC_STRICT_NC3 turned on.

NC_EPERM Attempt to create object in read-only file.

Example

In this example from `libsrc4/tst_vars2.c`, chunksizes are set with `nc_var_def_chunking`, and checked with `nc_var_inq_chunking`.

```
printf("**** testing chunking...");
{
#define NDIMS5 1
#define DIM5_NAME "D5"
#define VAR_NAME5 "V5"
#define DIM5_LEN 1000

    int dimids[NDIMS5], dimids_in[NDIMS5];
    int varid;
    int ndims, nvars, natts, unlimdimid;
    nc_type xtype_in;
    char name_in[NC_MAX_NAME + 1];
    int data[DIM5_LEN], data_in[DIM5_LEN];
    size_t chunksize[NDIMS5] = {5};
    size_t chunksize_in[NDIMS5];
    int storage_in;
    int i, d;

    for (i = 0; i < DIM5_LEN; i++)
        data[i] = i;
```

```

/* Create a netcdf-4 file with one dim and one var. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_dim(ncid, DIM5_NAME, DIM5_LEN, &dimids[0])) ERR;
if (nc_def_var(ncid, VAR_NAME5, NC_INT, NDIMS5, dimids, &varid)) ERR;
if (nc_def_var_chunking(ncid, varid, NC_CHUNKED, chunksize)) ERR;
if (nc_put_var_int(ncid, varid, data)) ERR;

/* Check stuff. */
if (nc_inq_var_chunking(ncid, varid, &storage_in, chunksize_in)) ERR;
for (d = 0; d < NDIMS5; d++)
    if (chunksize[d] != chunksize_in[d]) ERR;
if (storage_in != NC_CHUNKED) ERR;

```

6.7 Learn About Chunking Parameters for a Variable: `nc_inq_var_chunking`

The function `nc_inq_var_chunking` returns the chunking settings for a variable in a netCDF-4 file.

Usage

```
int nc_inq_var_chunking(int ncid, int varid, int *storagep, int *chunksizesp);
```

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

varid Variable ID.

storagep Address of returned storage property, returned as `NC_CONTIGUOUS` if this variable uses contiguous storage, or `NC_CHUNKED` if it uses chunked storage.

***chunksizesp**
A pointer to an array list of chunk sizes. The array must have one chunksize for each dimension in the variable.

Errors

`nc_inq_var_chunking` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

NC_NOERR No error.

NC_BADID Bad ncid.

NC_ENOTNC4
Not a netCDF-4 file.

NC_ENOTVAR
Can't find this variable.

Example

This example is from `libsrc4/tst_vars2.c` in which a variable with contiguous storage is created, and then checked with `nc_inq_var_chunking`:

```

    printf("**** testing contiguous storage...");
    {
#define NDIMS6 1
#define DIM6_NAME "D5"
#define VAR_NAME6 "V5"
#define DIM6_LEN 100

        int dimids[NDIMS6], dimids_in[NDIMS6];
        int varid;
        int ndims, nvars, natts, unlimdimid;
        nc_type xtype_in;
        char name_in[NC_MAX_NAME + 1];
        int data[DIM6_LEN], data_in[DIM6_LEN];
        size_t chunksize_in[NDIMS6];
        int storage_in;
        int i, d;

        for (i = 0; i < DIM6_LEN; i++)
            data[i] = i;

        /* Create a netcdf-4 file with one dim and one var. */
        if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
        if (nc_def_dim(ncid, DIM6_NAME, DIM6_LEN, &dimids[0])) ERR;
        if (dimids[0] != 0) ERR;
        if (nc_def_var(ncid, VAR_NAME6, NC_INT, NDIMS6, dimids, &varid)) ERR;
        if (nc_def_var_chunking(ncid, varid, NC_CONTIGUOUS, NULL)) ERR;
        if (nc_put_var_int(ncid, varid, data)) ERR;

        /* Check stuff. */
        if (nc_inq_var_chunking(ncid, 0, &storage_in, chunksize_in)) ERR;
        if (storage_in != NC_CONTIGUOUS) ERR;

```

6.8 Define Fill Parameters for a Variable: `nc_def_var_fill`

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

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

Usage

```
int nc_def_var_fill(int ncid, int varid, int no_fill, void *fill_value);
```

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

varid Variable ID.

- no_fill** Set no_fill mode on a variable. When this mode is on, fill values will not be written for the variable. This is helpful in high performance applications. For netCDF-4/HDF5 files (whether classic model or not), this may only be changed after the variable is defined, but before it is committed to disk (i.e. before the first nc_enddef after the nc_def_var.) For classic and 64-bit offset file, the no_fill mode may be turned on and off at any time.
- *fill_value** A pointer to a value which will be used as the fill value for the variable. Must be the same type as the variable. This will be written to a _FillValue attribute, created for this purpose. If NULL, this argument will be ignored.

Return Codes

- NC_NOERR** No error.
- NC_BADID** Bad ncid.
- NC_ENOTNC4**
Not a netCDF-4 file.
- NC_ENOTVAR**
Can't find this variable.
- NC_ELATEDEF**
This variable has already been the subject of a nc_enddef call. In netCDF-4 files nc_enddef will be called automatically for any data read or write. Once enddef has been called, it is impossible to set the fill for a variable.
- NC_ENOTINDEFINE**
Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with NC_STRICT_NC3 flag. (see [Section 2.5 \[nc_create\]](#), page 13).
- NC_EPERM** Attempt to create object in read-only file.

Example

This example is from libsrc4/tst_vars.c

```
int dimids[NDIMS];
size_t index[NDIMS];
int varid;
int no_fill;
unsigned short ushort_data = 42, ushort_data_in, fill_value_in;

/* Create a netcdf-4 file with one dim and 1 NC_USHORT var. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_dim(ncid, DIM7_NAME, DIM7_LEN, &dimids[0])) ERR;
if (nc_def_var(ncid, VAR7_NAME, NC_USHORT, NDIMS, dimids,
&varid)) ERR;
if (nc_def_var_fill(ncid, varid, 1, NULL)) ERR;
```

6.9 Learn About Fill Parameters for a Variable: `nc_inq_var_fill`

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

Usage

```
int nc_inq_var_fill(int ncid, int varid, int *no_fill, void *fill_value);
```

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

varid Variable ID.

***no_fill** Pointer to an integer which will get a 1 if no_fill mode is set for this variable. See [Section 6.8 \[nc_def_var_fill\]](#), page 90. This parameter will be ignored if it is NULL.

***fill_value** A pointer which will get the fill value for this variable. This parameter will be ignored if it is NULL.

Return Codes

NC_NOERR No error.

NC_BADID Bad ncid.

NC_ENOTNC4 Not a netCDF-4 file.

NC_ENOTVAR Can't find this variable.

Example

This example is from `libsrc4/tst_vars.c`

```
int dimids[NDIMS];
size_t index[NDIMS];
int varid;
int no_fill;
unsigned short ushort_data = 42, ushort_data_in, fill_value_in;

/* Create a netcdf-4 file with one dim and 1 NC_USHORT var. */
if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
if (nc_def_dim(ncid, DIM7_NAME, DIM7_LEN, &dimids[0])) ERR;
if (nc_def_var(ncid, VAR7_NAME, NC_USHORT, NDIMS, dimids,
&varid)) ERR;
if (nc_def_var_fill(ncid, varid, 1, NULL)) ERR;

/* Check stuff. */
if (nc_inq_var_fill(ncid, varid, &no_fill, &fill_value_in)) ERR;
if (!no_fill) ERR;
```

6.10 Define Compression Parameters for a Variable: `nc_def_var_deflate`

The function `nc_def_var_deflate` sets the deflate parameters for a variable in a netCDF-4 file.

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

This does not work with scalar variables.

Usage

```
nc_def_var_deflate(int ncid, int varid, int shuffle, int deflate,
                  int deflate_level);
```

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

varid Variable ID.

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

deflate If non-zero, turn on the deflate filter at the level specified by the `deflate_level` parameter.

deflate_level

If the deflate parameter is non-zero, set the deflate level to this value. Must be between 0 and 9.

Errors

`nc_def_var_deflate` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

NC_NOERR No error.

NC_BADID Bad `ncid`.

NC_ENOTNC4

Not a netCDF-4 file.

NC_ENOTVAR

Can't find this variable.

NC_ELATEDEF

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

NC_ENOTINDEFINE

Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with `NC_STRICT_NC3` flag. (see [Section 2.5 \[nc_create\]](#), page 13).

NC_EPERM Attempt to create object in read-only file.

NC_EINVAL

Invalid `deflate_level`. The deflate level must be between 0 and 9, inclusive.

Example

6.11 Learn About Deflate Parameters for a Variable: `nc_inq_var_deflate`

The function `nc_inq_var_deflate` returns the deflate settings for a variable in a netCDF-4 file.

Usage

```
nc_inq_var_deflate(int ncid, int varid, int *shufflep,
                  int *deflatep, int *deflate_levelp);
```

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

varid Variable ID.

***shufflep**

If this pointer is non-NULL, the `nc_inq_var_deflate` function will write a 1 if the shuffle filter is turned on for this variable, and a 0 otherwise.

***deflatep**

If this pointer is non-NULL, the `nc_inq_var_deflate` function will write a 1 if the deflate filter is turned on for this variable, and a 0 otherwise.

***deflate_levelp**

If this pointer is non-NULL, and the deflate filter is in use for this variable, the `nc_inq_var_deflate` function will write the `deflate_level` here.

Errors

`nc_inq_var_deflate` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

`NC_NOERR` No error.

`NC_BADID` Bad `ncid`.

`NC_ENOTNC4`

Not a netCDF-4 file.

`NC_ENOTVAR`

Can't find this variable.

Example

6.12 Define Checksum Parameters for a Variable: `nc_def_var_fletcher32`

The function `nc_def_var_fletcher32` sets the checksum parameters for a variable in a netCDF-4 file.

This function may only be called after the variable is defined, but before `nc_enddef` is called.

Usage

```
nc_def_var_fletcher32(int ncid, int varid, int checksum);
```

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

varid Variable ID.

checksum If this is `NC_FLETCHER32`, fletcher32 checksums will be turned on for this variable.

Errors

`nc_def_var_fletcher32` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

`NC_NOERR` No error.

`NC_BADID` Bad ncid.

`NC_ENOTNC4`
Not a netCDF-4 file.

`NC_ENOTVAR`
Can't find this variable.

`NC_ELATEDEF`
This variable has already been the subject of a `nc_enddef` call. In netCDF-4 files `nc_enddef` will be called automatically for any data read or write. Once `enddef` has been called, it is impossible to set the checksum property for a variable.

`NC_ENOTINDEFINE`
Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with `NC_STRICT_NC3` flag. (see [Section 2.5 \[nc_create\]](#), page 13).

`NC_EPERM` Attempt to create object in read-only file.

Example

6.13 Learn About Checksum Parameters for a Variable: `nc_inq_var_fletcher32`

The function `nc_inq_var_fletcher32` returns the checksum settings for a variable in a netCDF-4 file.

Usage

```
nc_inq_var_fletcher32(int ncid, int varid, int *checksump);
```

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

varid Variable ID.

***checksump**

If not-NULL, the `nc_inq_var_fletcher32` function will set the int pointed at to `NC_FLETCHER32` if the fletcher32 checksum filter is turned on for this variable, and `NC_NOCHECKSUM` if it is not.

Errors

`nc_inq_var_fletcher32` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

`NC_NOERR` No error.

`NC_BADID` Bad ncid.

`NC_ENOTNC4`
Not a netCDF-4 file.

`NC_ENOTVAR`
Can't find this variable.

Example**6.14 Define Endianness of a Variable: `nc_def_var_endian`**

The function `nc_def_var_endian` sets the endianness for a variable in a netCDF-4 file.

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

By default, netCDF-4 variables are in native endianness. That is, they are big-endian on a big-endian machine, and little-endian on a little endian machine.

In some cases a user might wish to change from native endianness to either big or little-endianness. This function allows them to do that.

Usage

```
nc_def_var_endian(int ncid, int varid, int endian);
```

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

varid Variable ID.

endian Set to `NC_ENDIAN_NATIVE` for native endianness. (This is the default). Set to `NC_ENDIAN_LITTLE` for little endian, or `NC_ENDIAN_BIG` for big endian.

Errors

`nc_def_var_endian` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

`NC_NOERR` No error.

NC_BADID Bad ncid.

NC_ENOTNC4
Not a netCDF-4 file.

NC_ENOTVAR
Can't find this variable.

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

NC_ENOTINDEFINE
Not in define mode. This is returned for netCDF classic or 64-bit offset files, or for netCDF-4 files, when they were been created with `NC_STRICT_NC3` flag, and the file is not in define mode. (see [Section 2.5 \[nc-create\]](#), page 13).

NC_EPERM Attempt to create object in read-only file.

Example

6.15 Learn About Endian Parameters for a Variable: `nc_inq_var_endian`

The function `nc_inq_var_endian` returns the endianness settings for a variable in a netCDF-4 file.

Usage

```
nc_inq_var_endian(int ncid, int varid, int *endianp);
```

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

varid Variable ID.

***endianp** If not-NULL, the `nc_inq_var_endian` function will set the int pointed to this to `NC_ENDIAN_LITTLE` if this variable is stored in little-endian format, `NC_ENDIAN_BIG` if it is stored in big-endian format, and `NC_ENDIAN_NATIVE` if the endianness is not set, and the variable is not created yet.

Errors

`nc_inq_var_endian` returns the value `NC_NOERR` if no errors occurred. Otherwise, the returned status indicates an error.

Possible return codes include:

NC_NOERR No error.

NC_BADID Bad ncid.

NC_ENOTNC4
Not a netCDF-4 file.

NC_ENOTVAR

Can't find this variable.

Example

6.16 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);
```

6.17 Get Information about a Variable from Its ID: `nc_inq_var`

family &index `nc_inq_varid`

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);

```

6.18 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_put_var1()` function will write a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_put_var1_ubyte (int ncid, int varid, const size_t index[],
        const unsigned char *up);
int nc_put_var1_ushort(int ncid, int varid, const size_t index[],
        const unsigned short *sp);
int nc_put_var1_uint  (int ncid, int varid, const size_t index[],
        const unsigned int *ip);
int nc_put_var1_longlong(int ncid, int varid, const size_t index[],
        const long long *lp);
int nc_put_var1_ulonglong(int ncid, int varid, const size_t index[],
        const unsigned long long *up);
int nc_put_var1_string(int ncid, int varid, const size_t index[],
        const char **ip);
int nc_put_var1(int ncid, int varid, const size_t *indexp,
        const void *op);

```

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*.

Return Codes

- **NC_NOERR** No error.
- **NC_EHDFERR** Error reported by HDF5 layer.
- **NC_ENOTVAR** The variable ID is invalid for the specified netCDF dataset.
- **NC_EINVALCOORDS** 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.
- **NC_ERANGE** The specified value is out of the range of values representable by the external data type of the variable. (Does not apply to `nc_put_var1()` function).

- NC_EINDEFINE The specified netCDF is in define mode rather than data mode.
- NC_EBADID 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);
```

6.19 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_put_var()` function will write a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_put_var_ubyte (int ncid, int varid, const unsigned char *op);
int nc_put_var_ushort(int ncid, int varid, const unsigned short *op);
int nc_put_var_uint  (int ncid, int varid, const unsigned int *op);
int nc_put_var_longlong (int ncid, int varid, const long long *op);
int nc_put_var_ulonglong(int ncid, int varid, const unsigned long long *op);
int nc_put_var_string(int ncid, int varid, const char **op);
int nc_put_var        (int ncid, int varid, const void *op);

```

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*](#).

Return Codes

- **NC_NOERR** The variable ID is invalid for the specified netCDF dataset.
- **NC_EHDFERR** Error reported by HDF5 layer.
- **NC_ERANGE** One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to `nc_put_var()` function).
- **NC_EINDEFINE** The specified netCDF dataset is in define mode rather than data mode.
- **NC_BADID** The specified netCDF ID does not refer to an open netCDF dataset.
- **NC_ENOTVAR** Bad variable ID.

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);
```

6.20 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_put_var()` function will write a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_put_vara_ubyte (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const unsigned char *op);
int nc_put_vara_ushort(int ncid, int varid, const size_t *startp,
                                const size_t *countp, const unsigned short *op);
int nc_put_vara_uint (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const unsigned int *op);
int nc_put_vara_longlong (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const long long *op);
int nc_put_vara_ulonglong(int ncid, int varid, const size_t *startp,
                                const size_t *countp, const unsigned long long *op);
int nc_put_vara_string(int ncid, int varid, const size_t *startp,
                                const size_t *countp, const char **op);
int nc_put_vara (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const void *op);

```

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.

Note: setting any element of the count array to zero causes the function to exit without error, and without doing anything.

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*](#).

Return Codes

- NC_NOERR No error.
- NC_EHDFERR Error reported by HDF5 layer.
- NC_ENOTVAR The variable ID is invalid for the specified netCDF dataset.
- NC_EINVALCOORDS 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.
- NC_EEDGE 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.
- NC_ERANGE One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_put_vara()` function).
- NC_EINDEFINE The specified netCDF dataset is in define mode rather than data mode.
- NC_EBADID The specified netCDF ID does not refer to an open netCDF dataset.
- NC_ECHAR Attempt to convert to or from char.
- NC_ENOMEM Out of memory.
- NC_EBADTYPE Bad type.

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);

```

6.21 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_put_vars()` function will write a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_put_vars_ubyte (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const unsigned char *op);
int nc_put_vars_ushort(int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const unsigned short *op);
int nc_put_vars_uint   (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const unsigned int *op);
int nc_put_vars_longlong (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const long long *op);
int nc_put_vars_ulonglong(int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const unsigned long long *op);
int nc_put_vars_string (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const char **op);
int nc_put_vars        (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const void *op);

```

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	<p>A vector of <code>size_t</code> 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.</p> <p>Note: setting any element of the count array to zero causes the function to exit without error, and without doing anything.</p>
stride	<p>A vector of <code>ptrdiff_t</code> 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).</p>
tp up cp sp ip lp fp dp	<p>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 <i>The NetCDF Users Guide</i>.</p>

Return Codes

- NC_NOERR No error.
- NC_EHDFERR Error reported by HDF5 layer.
- NC_ENOTVAR The variable ID is invalid for the specified netCDF dataset.
- NC_EINVALCOORDS 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.
- NC_EEDGE 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.
- NC_ERANGE One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_put_vars()` function).
- NC_EINDEFINE The specified netCDF dataset is in define mode rather than data mode.

- NC_EBADID The specified netCDF ID does not refer to an open netCDF dataset.
- NC_ECHAR Attempt to convert to or from char.
- NC_ENOMEM Out of memory.
- NC_EBADTYPE Bad type.

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);
```

6.22 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_put_varm()` function will write a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_put_varm_ubyte (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      const ptrdiff_t *imap, const unsigned char *op);
int nc_put_varm_ushort (int ncid, int varid, const size_t *startp,
                       const size_t *countp, const ptrdiff_t *stridep,
                       const ptrdiff_t *imap, const unsigned short *op);
int nc_put_varm_uint (int ncid, int varid, const size_t *startp,
                     const size_t *countp, const ptrdiff_t *stridep,
                     const ptrdiff_t *imap, const unsigned int *op);
int nc_put_varm_longlong (int ncid, int varid, const size_t *startp,
                          const size_t *countp, const ptrdiff_t *stridep,
                          const ptrdiff_t *imap, const long long *op);
int nc_put_varm_ulonglong (int ncid, int varid, const size_t *startp,
                           const size_t *countp, const ptrdiff_t *stridep,
                           const ptrdiff_t *imap, const unsigned long long *op);
int nc_put_varm_string (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_put_varm                (int ncid, int varid,  const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t *imapp, const void *op);
n

```

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.

Note: setting any element of the `count` array to zero causes the function to exit without error, and without doing anything.

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*.

Return Codes

- NC_NOERR No error.
- NC_EHDFERR Error reported by HDF5 layer.
- NC_ENOTVAR The variable ID is invalid for the specified netCDF dataset.
- NC_EINVALCOORDS 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.
- NC_EEDGE 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.
- NC_ERANGE One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_put_vars()` function).
- NC_EINDEFINE The specified netCDF dataset is in define mode rather than data mode.
- NC_EBADID The specified netCDF ID does not refer to an open netCDF dataset.
- NC_ECHAR Attempt to convert to or from char.
- NC_ENOMEM Out of memory.

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);

```

6.23 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_get_var1()` function will read a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_get_var1_ubyte (int ncid, int varid, const size_t *indexp,
                      unsigned char *ip);

```

```

int nc_get_var1_ushort(int ncid, int varid, const size_t *indexp,
                      unsigned short *ip);
int nc_get_var1_uint  (int ncid, int varid, const size_t *indexp,
                      unsigned int *ip);
int nc_get_var1_longlong (int ncid, int varid, const size_t *indexp,
                          long long *ip);
int nc_get_var1_ulonglong(int ncid, int varid, const size_t *indexp,
                          unsigned long long *ip);
int nc_get_var1_string(int ncid, int varid, const size_t *indexp,
                      char **ip);
int nc_get_var1        (int ncid, int varid, const size_t *indexp,
                      void *ip);

```

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*](#).

Return Codes

- **NC_NOERR** No error.
- **NC_EHDFERR** Error reported by HDF5 layer.
- **NC_ENOTVAR** The variable ID is invalid for the specified netCDF dataset.
- **NC_EINVALCOORDS** 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.
- **NC_EEDGE** 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.
- **NC_ERANGE** One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_put_vars()` function).

- NC_EINDEFINE The specified netCDF is in define mode rather than data mode.
- NC_EBADID The specified netCDF ID does not refer to an open netCDF dataset.
- NC_ECHAR Attempt to convert to or from char.
- NC_ENOMEM Out of memory.

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);
```

6.24 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_get_var()` function will read a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_get_var_ubyte  (int ncid, int varid, unsigned char *ip);
int nc_get_var_ushort (int ncid, int varid, unsigned short *ip);
int nc_get_var_uint   (int ncid, int varid, unsigned int *ip);
int nc_get_var_longlong (int ncid, int varid, long long *ip);
int nc_get_var_ulonglong (int ncid, int varid, unsigned long long *ip);
int nc_get_var_string (int ncid, int varid, char **ip);
int nc_get_var        (int ncid, int varid, void *ip);

```

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*](#).

Return Codes

- **NC_NOERR** No error.
- **NC_EHDFERR** Error reported by HDF5 layer.
- **NC_ENOTVAR** The variable ID is invalid for the specified netCDF dataset.
- **NC_EINVALCOORDS** 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.
- **NC_EEDGE** 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.
- **NC_ERANGE** One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_put_vars()` function).

- NC_EINDEFINE The specified netCDF is in define mode rather than data mode.
- NC_EBADID The specified netCDF ID does not refer to an open netCDF dataset.
- NC_ECHAR Attempt to convert to or from char.
- NC_ENOMEM Out of memory.

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);
```

6.25 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_get_vara()` function will write a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_get_vara_ubyte (int ncid, int varid, const size_t *startp,
                    const size_t *countp, unsigned char *ip);
int nc_get_vara_ushort(int ncid, int varid, const size_t *startp,
                    const size_t *countp, unsigned short *ip);
int nc_get_vara_uint (int ncid, int varid, const size_t *startp,
                    const size_t *countp, unsigned int *ip);
int nc_get_vara_longlong(int ncid, int varid, const size_t *startp,
                    const size_t *countp, long long *ip);
int nc_get_vara_ulonglong(int ncid, int varid, const size_t *startp,
                    const size_t *countp, unsigned long long *ip);
int nc_get_vara_string(int ncid, int varid, const size_t *startp,
                    const size_t *countp, char **ip);
int nc_get_vara (int ncid, int varid, const size_t start[],
                const size_t count[], void *ip);

```

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.

Note: setting any element of the `count` array to zero causes the function to exit without error, and without doing anything.

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 <i>The NetCDF Users Guide</i> .

Return Codes

- NC_NOERR No error.
- NC_ENOTVAR The variable ID is invalid for the specified netCDF dataset.
- NC_EINVALCOORDS 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.
- NC_EEDGE 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.
- NC_ERANGE One or more of the values are out of the range of values representable by the desired type. (Does not apply to `nc_get_vara()` function).
- NC_EINDEFINE The specified netCDF is in define mode rather than data mode.
- NC_EBADID 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);

```

6.26 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_get_vars()` function will read a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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)
int nc_get_vars_ubyte (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      unsigned char *ip);
int nc_get_vars_ushort(int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      unsigned short *ip);
int nc_get_vars_uint  (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      unsigned int *ip);
int nc_get_vars_longlong (int ncid, int varid, const size_t *startp,
                          const size_t *countp, const ptrdiff_t *stridep,
                          long long *ip);
int nc_get_vars_ulonglong(int ncid, int varid, const size_t *startp,
                          const size_t *countp, const ptrdiff_t *stridep,
                          unsigned long long *ip);
int nc_get_vars_string(int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      char **ip);
int nc_get_vars      (int ncid, int varid, const size_t *startp,
                      const size_t *countp, const ptrdiff_t *stridep,
                      void *ip);

```

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.

Note: setting any element of the `count` array to zero causes the function to exit without error, and without doing anything.

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 <i>The NetCDF Users Guide</i> .

Return Codes

- NC_NOERR No error.
- NC_EHDFERR Error reported by HDF5 layer.
- NC_ENOTVAR The variable ID is invalid for the specified netCDF dataset.
- NC_EINVALCOORDS 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.
- NC_EEDGE 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.
- NC_ERANGE One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_get_vars()` function).
- NC_EINDEFINE The specified netCDF is in define mode rather than data mode.
- NC_EBADID The specified netCDF ID does not refer to an open netCDF dataset.
- NC_ECHAR Attempt to convert to or from char.
- NC_ENOMEM Out of memory.

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);
...

```

6.27 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.

The functions for types `ubyte`, `ushort`, `uint`, `longlong`, `ulonglong`, and `string` are only available for netCDF-4/HDF5 files.

The `nc_get_varm()` function will read a variable of any type, including user defined type. For this function, the type of the data in memory must match the type of the variable - no data conversion is done.

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);
int nc_get_varm_ubyte (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_get_varm_ushort(int ncid, int varid, const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t * imapp, unsigned short *ip);
int nc_get_varm_uint  (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t * imapp, unsigned int *ip);
int nc_get_varm_longlong (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t * imapp, long long *ip);
int nc_get_varm_ulonglong(int ncid, int varid, const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t * imapp, unsigned long long *ip);
int nc_get_varm_string(int ncid, int varid, const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t * imapp, char **ip);
int nc_get_varm      (int ncid, int varid, const size_t *startp,
                                const size_t *countp, const ptrdiff_t *stridep,
                                const ptrdiff_t *imapp, void *ip);

```

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.

Note: setting any element of the count array to zero causes the function to exit without error, and without doing anything.

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*.

Return Codes

- `NC_NOERR` No error.
- `NC_EHDFERR` Error reported by HDF5 layer.
- `NC_ENOTVAR` The variable ID is invalid for the specified `netCDF` dataset.
- `NC_EINVALCOORDS` 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.
- `NC_EEDGE` 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.

- **NC_ERANGE** One or more of the specified values are out of the range of values representable by the external data type of the variable. (Does not apply to the `nc_get_vars()` function).
- **NC_EINDEFINE** The specified netCDF is in define mode rather than data mode.
- **NC_EBADID** The specified netCDF ID does not refer to an open netCDF dataset.
- **NC_ECHAR** Attempt to convert to or from char.
- **NC_ENOMEM** Out of memory.

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);

```

6.28 Reading and Writing Character String Values

Prior to version 4.0, strings could only be stored as simple arrays of characters. Users may still wish to store strings this way, as it ensures maximum compatibility with other software.

Starting in netCDF-4.0, the atomic string type allows a new way to store strings, as a variable length array in the underlying HDF5 layer. This allows arrays of strings to be stored compactly.

For more information of classic models strings [Section 6.28.1 \[Classic Strings\]](#), page 130. For more information on the netCDF-4.0 string type [Section 6.28.2 \[Arrays of Strings\]](#), page 131.

6.28.1 Reading and Writing Character String Values in the Classic Model

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);

```

6.28.2 Reading and Writing Arrays of Strings

In netCDF-4, the NC_STRING type is introduced. It can store arrays of strings compactly.

By using the NC_STRING type, arrays of strings (char **) can be read and written to the file.

This allows attributes to hold more than one string. Since attributes are one-dimensional, using the classic model, an attribute could only hold one string, as an array of char. With the NC_STRING type, an array of strings can be stored in one attribute.

When reading data of type NC_STRING, the HDF5 layer will allocate memory to hold the data. It is up to the user to free this memory with the nc_free_string function. See [Section 6.29 \[nc-free-string\]](#), page 132.

```

int ncid, varid, i, dimids[NDIMS];
char *data[DIM_LEN] = {"Let but your honour know",
    "Whom I believe to be most strait in virtue",
    "That, in the working of your own affections",
    "Had time cohered with place or place with wishing",
    "Or that the resolute acting of your blood",
    "Could have attain'd the effect of your own purpose",
    "Whether you had not sometime in your life",
    "Err'd in this point which now you censure him",

```

```

                                "And pull'd the law upon you."};
char *data_in[DIM_LEN];

printf("*** testing string attribute...");
{
    size_t att_len;
    int ndims, nvars, natts, unlimdimid;
    nc_type att_type;

    if (nc_create(FILE_NAME, NC_NETCDF4, &ncid)) ERR;
    if (nc_put_att(ncid, NC_GLOBAL, ATT_NAME, NC_STRING, DIM_LEN, data)) ERR;
    if (nc_inq(ncid, &ndims, &nvars, &natts, &unlimdimid)) ERR;
    if (ndims != 0 || nvars != 0 || natts != 1 || unlimdimid != -1) ERR;
    if (nc_inq_att(ncid, NC_GLOBAL, ATT_NAME, &att_type, &att_len)) ERR;
    if (att_type != NC_STRING || att_len != DIM_LEN) ERR;
    if (nc_close(ncid)) ERR;
    nc_exit();

    /* Check it out. */
    if (nc_open(FILE_NAME, NC_NOWRITE, &ncid)) ERR;
    if (nc_inq(ncid, &ndims, &nvars, &natts, &unlimdimid)) ERR;
    if (ndims != 0 || nvars != 0 || natts != 1 || unlimdimid != -1) ERR;
    if (nc_inq_att(ncid, NC_GLOBAL, ATT_NAME, &att_type, &att_len)) ERR;
    if (att_type != NC_STRING || att_len != DIM_LEN) ERR;
    if (nc_get_att(ncid, NC_GLOBAL, ATT_NAME, data_in)) ERR;
    for (i=0; i<att_len; i++)
        if (strcmp(data_in[i], data[i])) ERR;
    if (nc_free_string(att_len, (char **)data_in)) ERR;
    if (nc_close(ncid)) ERR;
    nc_exit();
}

```

6.29 Releasing Memory for a NC_STRING: nc_free_string

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

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

To save the user the trouble calling `free()` on each element of the NC_STRING array (i.e. the array of arrays), the `nc_free_string` function is provided.

Usage

```
int nc_free_string(size_t len, char **data);
```

`len` The number of character arrays in the array.

****data** The pointer to the data array.

Return Codes

NC_NOERR No error.

Example

```
if (nc_get_att(ncid, NC_GLOBAL, ATT_NAME, data_in)) ERR;
...
if (nc_free_string(att_len, (char **)data_in)) ERR;
```

6.30 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.

6.31 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);
```

6.32 Copy a Variable from One File to Another: `nc_copy_var`

This function will copy a variable from one file to another.

It works even if the files are different formats, (i.e. classic vs. netCDF-4/HDF5.)

If you're copying into a netcdf-3 file, from a netcdf-4 file, you must be copying a var of one of the six netcdf-3 types. Similarly for the attributes.

Usage

```
nc_copy_var(int ncid_in, int varid_in, int ncid_out)
```

- `ncid_in` The file ID for the file that contains the variable to be copied.
- `varid_in` The variable ID for the variable to be copied.
- `ncid_out` The file ID for the file where the variable should be copied to.

Return Codes

- `NC_NOERR` No error.
- `NC_EBADID` Bad `ncid`.
- `NC_EBADVAR` Bad `varid`.
- `NC_EHDFERR` HDF5 layer error.
- `NC_ENOMEM` Out of memory.
- `NC_ERANGE` One or more values out of range.

Example

6.33 Change between Collective and Independent Parallel Access: `nc_var_par_access`

The function `nc_var_par_access` changes whether read/write operations on a parallel file system are performed collectively (the default) or independently on the variable. This function can only be called if the file was created with `nc_create_par` (see [Section 2.7 \[nc_create_par\]](#), [page 18](#)) or opened with `nc_open_par` (see [Section 2.10 \[nc_open_par\]](#), [page 22](#)).

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

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

Usage

```
int nc_var_par_access(int ncid, int varid, int access);
```

ncid NetCDF ID, from a previous call to `nc_open_par` (see [Section 2.10 \[nc_open_par\]](#), [page 22](#)) or `nc_create_par` (see [Section 2.7 \[nc_create_par\]](#), [page 18](#)).

varid Variable ID.

access `NC_INDEPENDENT` to set this variable to independent operations. `NC_COLLECTIVE` to set it to collective operations.

Return Values

`NC_NOERR` No error.

Example

Here is an example using `nc_var_par_access`:

```
#include <netcdf.h>

...
int ncid, v1id, dimids[NDIMS];
int data[DIMSIZE*DIMSIZE], j, i, res;
...

/* Create a parallel netcdf-4 file. */
if ((res = nc_create_par(FILE, NC_MPIIO, comm, info, &ncid)))
    BAIL(res);

/* Create two dimensions. */
if ((res = nc_def_dim(ncid, "d1", DIMSIZE, dimids)))
    BAIL(res);
if ((res = nc_def_dim(ncid, "d2", DIMSIZE, &dimids[1])))
    BAIL(res);

/* Create one var. */
if ((res = nc_def_var(ncid, "v1", NC_INT, NDIMS, dimids, &v1id)))
    BAIL(res);

if ((res = nc_enddef(ncid)))
    BAIL(res);

/* Tell HDF5 to use independent parallel access for this var. */
if ((res = nc_var_par_access(ncid, v1id, NC_INDEPENDENT)))
    BAIL(res);

/* Write slabs of phony data. */
if ((res = nc_put_vara_int(ncid, v1id, start, count,
                           &data[mpi_rank*QTR_DATA])))
    BAIL(res);
```

7 Attributes

7.1 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.

7.2 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

With netCDF-4 files, `nc_put_att` will notice if you are writing a `_Fill_Value_` attribute, and will tell the HDF5 layer to use the specified fill value for that variable.

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

Use the `nc_put_att` function to create attributes of any type, including user-defined types. We recommend using the type safe versions of this function whenever possible.

```

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);
int nc_put_att_ubyte     (int ncid, int varid, const char *name, nc_type xtype,
                        size_t len, const unsigned char *op);
int nc_put_att_ushort    (int ncid, int varid, const char *name, nc_type xtype,
                        size_t len, const unsigned short *op);
int nc_put_att_uint      (int ncid, int varid, const char *name, nc_type xtype,
                        size_t len, const unsigned int *op);
int nc_put_att_longlong  (int ncid, int varid, const char *name, nc_type xtype,
                        size_t len, const long long *op);
int nc_put_att_ulonglong (int ncid, int varid, const char *name, nc_type xtype,
                        size_t len,
                        const unsigned long long *op);
int nc_put_att_string    (int ncid, int varid, const char *name, size_t len,
                        const char **op);
int nc_put_att           (int ncid, int varid, const char *name, nc_type xtype,
                        size_t len, const void *op);

```

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. 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.

Return Codes

NC_NOERR No error.

NC_EINVAL

Trying to set global _FillValue. (NetCDF-4 files only).

NC_ENOTVAR

Couldn't find varid.

NC_EBADTYPE

Fill value must be same type as variable. (NetCDF-4 files only).

NC_ENOMEM

Out of memory

NC_EFILLVALUE

Fill values must be written while the file is still in initial define mode, that is, after the file is created, but before it leaves define mode for the first time. NC_EFILLVALUE is returned when the user attempts to set the fill value after it's too late.

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);

```

7.3 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 *attnum);

```

ncid	NetCDF ID, from a previous call to <code>nc_open</code> or <code>nc_create</code> .
varid	Variable ID of the attribute's variable, or <code>NC_GLOBAL</code> for a global attribute.
name	Attribute name. For <code>nc_inq_attname</code> , 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, <code>nc_type</code> , is defined in the netCDF header file. The valid netCDF external data types are <code>NC_BYTE</code> , <code>NC_CHAR</code> , <code>NC_SHORT</code> , <code>NC_INT</code> , <code>NC_FLOAT</code> , and <code>NC_DOUBLE</code> . If this parameter is given as <code>'0'</code> (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 <code>NC_CHAR</code> , 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 <code>'0'</code> (a null pointer), no length will be returned so no variable to hold this information needs to be declared.
attnum	For <code>nc_inq_attname</code> , attribute number. The attributes for each variable are numbered from 0 (the first attribute) to <code>natts-1</code> , where <code>natts</code> is the number of attributes for the variable, as returned from a call to <code>nc_inq_varnatts</code> .
attnump	For <code>nc_inq_attid</code> , 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 */
size_t 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);

```

7.4 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.

The `nc_get_att()` functions works for any type of attribute, and must be used to get attributes of user-defined type. We recommend that they type safe versions of this function be used where possible.

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);
int nc_get_att_ubyte     (int ncid, int varid, const char *name, unsigned char *ip);
int nc_get_att_ushort    (int ncid, int varid, const char *name, unsigned short *ip);
int nc_get_att_uint      (int ncid, int varid, const char *name, unsigned int *ip);
int nc_get_att_longlong  (int ncid, int varid, const char *name, long long *ip);
int nc_get_att_ulonglong (int ncid, int varid, const char *name, unsigned long long *ip);
int nc_get_att_string    (int ncid, int varid, const char *name, char **ip);
int nc_get_att           (int ncid, int varid, const char *name, void *ip);

```

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	<p>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.</p>
--	--

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 */
...

```

7.5 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 <code>nc_open</code> or <code>nc_create</code> . |
| varid_in | ID of the variable in the input netCDF dataset from which the attribute will be copied, or <code>NC_GLOBAL</code> 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 <code>nc_open</code> or <code>nc_create</code> . 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 <code>NC_GLOBAL</code> 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);
```

7.6 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);
```

7.7 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 *imapp, 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 3 to NetCDF 4 Transition Guide

B.1 Introduction

The release of netCDF-4 represents a substantial increase in the capabilities of the netCDF C and Fortran APIs.

The netCDF-4.0 release (June, 2008) allows the use of the popular HDF5 data format as a storage layer. The HDF5 format has many features, and only a subset of them are exposed in the netCDF-4 API. This represents a deliberate selection process by netCDF-4 developers. We strove to chose the most useful features of the HDF5 model, while retaining the simplicity of the netCDF APIs.

Although there a many new features, full backward compatibility is assured (and extensively tested). Existing software will continue to work, existing netCDF data files will continue to work with netCDF-4.0, just as with previous releases of the netCDF library.

The use of netCDF-4 files allows the use of the expanded data model, including user-defined types, groups, the new unsigned, 64-bit, and string types.

Using netCDF-4 files also allows the use of such features as endianness control, per-variable data compression, chunking, parallel I/O, and checksums. These features fit neatly within the classic netCDF data model.

Although the expanded data model offers many exciting new features, we expect and encourage users to proceed with care - it also allows the creation of needlessly, even horribly complex files. This would decrease interoperability and increase the work of the poor programmers trying to use the data file.

There are many netCDF-4 features which fit comfortably within the classic netCDF model. Existing programs can be very quickly converted to use features such as compression, endianness control, and chunking. This allows users to gain immediate performance pay off, with minimal software development effort.

B.2 NetCDF-4 and HDF5

NetCDF-4 depends on HDF5 to deliver the new features of the expanded data model, as well as the features required to support the classic netCDF data model.

NetCDF-4 users must have at least HDF5 version 1.8.1 (and at lease zlib-1.2.3) to use HDF5 with netCDF-4.0. If these packages are not found when netCDF is built, then the netCDF library may still be build (without the `-enable-netcdf-4` option), but will not allow users to create netCDF-4/HDF5 files, or use the expanded data model. Only classic and 64-bit offset format netCDF files will be created or readable. (see [Section “Configure” in *The NetCDF Installation and Porting Guide*](#)).

The HDF5 files created by netCDF-4 will be readable (and writable) by any HDF5 application. However, netCDF-4.0 cannot read any HDF5 file, only those created by netCDF-4.

B.3 Backward Compatibility

In the context of netCDF, backward compatibility has several meanings.

Data Compatibility

NetCDF-4 provides backward compatibility for existing data. All netCDF data files remain readable to the netCDF library. When a file is opened, the library detects the underlying format of the file; this is transparent to the programmer and user.

Code Compatibility

NetCDF-4 provides backward compatibility for existing software. Programs using the 4.0 release can use it as a drop-in replacement for netCDF-3.x. Existing programs will continue to create netCDF classic or 64-bit offset files.

Model Compatibility

NetCDF-4 introduces an expanded model of a netCDF data file (include such new elements as groups, user-defined types, multiple-unlimited dimensions, etc.) This expanded model is a super-set of the classic netCDF model. Everything that works in the classic model works in the expanded model as well. (The reverse is not true - code using the expanded data model will fail if run on classic model netCDF files.)

B.4 The Classic and the Expanded NetCDF Data Models

The classic netCDF data model consists of variables, dimensions, and attributes.

The netCDF-4.0 release introduces an expanded data model, which offers many new features. These features will only work on files which have been created with the NC_NETCDF4 flag, and without the NC_CLASSIC_MODEL flag (see [Section 2.5 \[nc_create\]](#), page 13).

New Types New data types are introduced: NC_UBYTE, NC_USHORT, NC_UINT, NC_INT64, NC_UINT64, and NC_STRING. These types may be used for attributes and variables. See [Section 6.5 \[nc_def_var\]](#), page 85.

Groups NetCDF objects may now be organized into a hierarchical set of groups. Groups are organized much like a UNIX file system, with each group capable of containing more groups. Within each group, a classic model netCDF “file” exists, with its own dimensions, variables, and attributes. See [Section 3.9 \[nc_def_grp\]](#), page 44.

User Defined Types

NetCDF-4 allows the user to define new data types, including a compound type (see [Section 5.6 \[nc_def_compound\]](#), page 56), a variable length array type (see [Section 5.21 \[nc_def_vlen\]](#), page 70), an enumerated type (see [Section 5.28 \[nc_def_enum\]](#), page 75), and an opaque type (see [Section 5.25 \[nc_def_opaque\]](#), page 73).

Multiple Unlimited Dimensions

NetCDF-4/HDF5 data files may use multiple unlimited dimensions with a file, and even within a variable.

B.5 Using NetCDF-4.0 with the Classic and 64-bit Offset Formats

Prior to the 4.0 release, two underlying data formats were available for the netCDF user, the classic, and the 64-bit offset format. (The 64-bit offset format was introduced in the 3.6.0 release, and allows the use of larger variables).

Software using netCDF, relinked against the netCDF-4.0 library, will continue to work exactly as before. Since the default create mode in `nc_create` is to create a classic format file, using unmodified netCDF-3 code with the netCDF-4 library will result in the exact same output - a classic netCDF file or 64-bit offset file.

When writing or reading classic and 64-bit offset files, the netCDF-4.0 library relies on the core netCDF-3.x code.

B.6 Creating a NetCDF-4/HDF5 File

The extra features of netCDF-4 can only be accessed by adding the `NC_NETCDF4` flag to the create mode of `nc_create`. Files created with the `NC_NETCDF4` flag can have multiple unlimited dimensions, use the new atomic types, use compound and opaque types, and take advantage of the other features of netCDF-4. (see [Section 2.5 \[nc_create\]](#), page 13).

B.7 Using NetCDF-4.0 with the Classic Model

By changing your `nc_create` call to create a netCDF-4/HDF5 file you gain access to many new features - perhaps too many! Using groups or user-defined types will make the file unreadable to existing netCDF applications, until they are updated to handle the new netCDF-4 model.

Using the `NC_CLASSIC_MODEL` flag with the `NC_NETCDF4` flag tells the library to create a netCDF-4/HDF5 file which must abide by the rules of the classic netCDF data model. Such a file may not contain groups, user defined types, multiple unlimited dimensions, etc.

But a classic model file is guaranteed to be compatible with existing netCDF software, once they relink to the netCDF 4.0 library.

Some features of netCDF-4 are transparent to the user when the file is read. For example, a netCDF-4/HDF5 file may contain compressed data. When such a file is read, the decompression of the data takes place transparently. This means that data may use the data compression feature, and still conform to the classic netCDF data model, and thus retain compatibility with existing netCDF software (see [Section 6.10 \[nc_def_var_deflate\]](#), page 93). The same applies for control of endianness (see [Section 6.14 \[nc_def_var_endian\]](#), page 96), chunking (see [Section 6.6 \[nc_def_var_chunking\]](#), page 87), checksums (see [Section 6.12 \[nc_def_var_fletcher32\]](#), page 94), and parallel I/O, if netCDF-4 was built on a system with the MPI libraries.

To use these feature, change your `nc_create` calls to use the `NC_NETCDF4` and `NC_CLASSIC_MODEL` flags. Then call the appropriate `nc_def_var_*` function after the variable is defined, but before the next call to `nc_enddef`.

B.8 Use of the Expanded Model Impacts Fortran Portability

Using expanded model features severely impacts portability for Fortran programmers.

Fortran compilers do not always agree as to how data should be laid out in memory. This makes handling compound and variable length array types compiler and platform dependant.

(This is also true for C, but the clever HDF5 configuration has solved this problem for C. Alas, not for Fortran.)

Despite this, Fortran programs can take advantage of the new data model. The portability challenge is no different from that which Fortran programmers already deal with when doing data I/O.

B.9 The C++ API Does Not Handle Expanded Model in this Release

Unfortunately, the C++ API does not support the netCDF-4 expanded data model. A new C++ API is being developed and may be built by adventurous users using the `-enable-cxx4` option to configure (see [Section “Configure”](#) in *The NetCDF Installation and Porting Guide*).

Appendix C NetCDF 2 to NetCDF 3 C Transition Guide

C.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.

C.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.

C.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`:

<code>var</code>	entire variable access
<code>var1</code>	single value access
<code>vara</code>	array or array section access
<code>vars</code>	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.

C.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.

C.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.

C.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.

C.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. The `ncdump` and `ncgen` utilities now properly escape and handle escaped special characters in names, so that all valid netCDF names are representable in CDL.

C.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 auto-correlation 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	<code>nc_abort</code>
ncattcopy	<code>nc_copy_att</code>
ncattdel	<code>nc_del_att</code>
ncattget	<code>nc_get_att_double</code> , <code>nc_get_att_float</code> , <code>nc_get_att_int</code> , <code>nc_get_att_long</code> , <code>nc_get_att_schar</code> , <code>nc_get_att_short</code> , <code>nc_get_att_text</code> , <code>nc_get_att_uchar</code>
ncattinq	<code>nc_inq_att</code> , <code>nc_inq_attid</code> , <code>nc_inq_attlen</code> , <code>nc_inq_atttype</code>
ncattname	<code>nc_inq_attname</code>
ncattput	<code>nc_put_att_double</code> , <code>nc_put_att_float</code> , <code>nc_put_att_int</code> , <code>nc_put_att_long</code> , <code>nc_put_att_schar</code> , <code>nc_put_att_short</code> , <code>nc_put_att_text</code> , <code>nc_put_att_uchar</code>

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 D NetCDF-3 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_EMAXATTRS      (-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 */

```


Appendix E NetCDF-4 Error Codes

NetCDF-4 uses all error codes from NetCDF-3 (see [Appendix D \[NetCDF-3 Error Codes\]](#), [page 167](#)). The following additional error codes were added for new errors unique to netCDF-4.

```
#define NC_EHDFERR          (-101)
#define NC_ECANTREAD        (-102)
#define NC_ECANTWRITE       (-103)
#define NC_ECANTCREATE      (-104)
#define NC_EFILEMETA        (-105)
#define NC_EDIMMETA         (-106)
#define NC_EATTMETA         (-107)
#define NC_EVARMETA         (-108)
#define NC_ENOCOMPOUND      (-109)
#define NC_EATTEXISTS       (-110)
#define NC_ENOTNC4          (-111) /* Attempting netcdf-4 operation on netcdf-3 file. */
#define NC_ESTRICNC3        (-112) /* Attempting netcdf-4 operation on strict nc3 netcdf-
#define NC_EBADGRPID        (-113) /* Bad group id. Bad! */
#define NC_EBADTYPEID       (-114) /* Bad type id. */
#define NC_EBADFIELDID      (-115) /* Bad field id. */
#define NC_EUNKNAME         (-116)
```


8 Index

A

abnormal termination	3
aborting define mode	6
aborting definitions	6
adding attributes	6
adding attributes using nc_redef	23
adding dimensions	6
adding dimensions using nc_redef	23
adding variables	6
adding variables using nc_redef	23
API, C summary	149
appending data to variable	83
array section, reading mapped	119
array section, reading subsampled	122
array section, writing	104
array section, writing mapped	119
array section, writing subsampled	122
array, writing mapped	110
attnum	141
attnump	141
attributes, adding	6
attributes, array of strings	129
attributes, character string	130
attributes, copying	144
attributes, creating	137
attributes, deleting	146
attributes, deleting, introduction	6
attributes, finding length	140
attributes, getting information about	140
attributes, ID	140
attributes, inquiring about	140
attributes, introduction	137
attributes, number of	28
attributes, operations on	137
attributes, reading	142
attributes, renaming	145
attributes, writing	137

B

backing out of definitions	31
backward compatibility with v2 API	159
big-endian	96
bit lengths of data types	83
bit lengths of netcdf-3 data types	84
bit lengths of netcdf-4 data types	84
byte vs. char fill values	133
byte, zero	130

C

C API summary	149
call sequence, typical	3
canceled definitions	31

character-string data, writing	130
checksum	94
chunking	87
code templates	3
compiling with netCDF library	7
compound types, overview	56
compression, setting parameters	93
contiguous	87
copying attributes	144
create flag, setting default	34
creating a dataset	3
creating variables	85

D

datasets, overview	9
deflate	93
deleting attributes	146
dimensions, adding	6
dimensions, number of	28

E

endianness	96
entire variable, reading	117
entire variable, writing	102
enum type	74
error codes	12
error codes, netcdf-3	167
error codes, netcdf-4	169
error handling	7

F

fill	90
fill values	133
fletcher32	94
format version	28

G

groups, overview	37
------------------------	----

H

handle_err	12
HDF5 errors, first create	13, 19

I

inquiring about attributes	140
inquiring about variables	98
interface descriptions	9

L

length of attributes	140
lenp	141
linking to netCDF library	7
little-endian	96

M

mapped array section, writing	125
mapped array, writing	110

N

name	141
nc__create	16
nc__create, example	16
nc__create, flags	16
nc__enddef	26
nc__enddef, example	26
nc__open	21
nc__open, example	21
NC_64BIT_OFFSET	13, 16
nc_abort	31
nc_abort, example	31
NC_CLOBBER	13, 16, 18
nc_close	27
nc_close, example	27
nc_close, root group	27
nc_close, typical use	3
nc_copy_att	144
nc_copy_att, example	144
nc_copy_var	134
nc_create	13
nc_create, example	13
nc_create, flags	13
nc_create, typical use	3
nc_create_par	18
nc_create_par, example	18
nc_create_par, flags	18
nc_def_compound	56
nc_def_dim	47
nc_def_dim, example	47
nc_def_dim, typical use	3, 6
nc_def_enum	75
nc_def_grp	44
nc_def_opaque	73
nc_def_var	85
nc_def_var, example	85
nc_def_var, typical use	3
nc_def_var_chunking	87
nc_def_var_deflate	93
nc_def_var_endian	96
nc_def_var_fill	90
nc_def_var_fletcher32	94
nc_def_vlen	70, 71
nc_del_att	146
nc_del_att, example	146
nc_enddef	25

nc_enddef, example	25
nc_enddef, typical use	3
nc_free_string	132
nc_free_vlen	72
nc_get_att, typical use	4, 5
nc_get_att_ type	142
nc_get_att_ type, example	142
nc_get_var	117
nc_get_var, typical use	4, 5
nc_get_var_ type	117
nc_get_var_ type, example	117
nc_get_var_double	117
nc_get_var_float	117
nc_get_var_int	117
nc_get_var_long	117
nc_get_var_longlong	117
nc_get_var_schar	117
nc_get_var_short	117
nc_get_var_string	117
nc_get_var_text	117
nc_get_var_ubyte	117
nc_get_var_uchar	117
nc_get_var_uint	117
nc_get_var_ulonglong	117
nc_get_var_ushort	117
nc_get_var1	115
nc_get_var1_ type	115
nc_get_var1_ type, example	115
nc_get_var1_double	115
nc_get_var1_float	115
nc_get_var1_int	115
nc_get_var1_long	115
nc_get_var1_longlong	115
nc_get_var1_schar	115
nc_get_var1_short	115
nc_get_var1_string	115
nc_get_var1_text	115
nc_get_var1_ubyte	115
nc_get_var1_uchar	115
nc_get_var1_uint	115
nc_get_var1_ulonglong	115
nc_get_var1_ushort	115
nc_get_vara	119
nc_get_vara_ type	119
nc_get_vara_ type, example	119
nc_get_vara_double	119
nc_get_vara_float	119
nc_get_vara_int	119
nc_get_vara_long	119
nc_get_vara_longlong	119
nc_get_vara_schar	119
nc_get_vara_short	119
nc_get_vara_string	119
nc_get_vara_text	119
nc_get_vara_ubyte	119
nc_get_vara_uchar	119
nc_get_vara_uint	119
nc_get_vara_ulonglong	119

nc_get_vara_ushort	119	nc_inq_dimid, typical use	4
nc_get_varm	125	nc_inq_dimids	40, 49
nc_get_varm_type	125	nc_inq_dimlen	49
nc_get_varm_type, example	125	nc_inq_dimname	49
nc_get_varm_double	125	nc_inq_enum	78
nc_get_varm_float	125	nc_inq_enum_ident	81
nc_get_varm_int	125	nc_inq_enum_member	80
nc_get_varm_long	125	nc_inq_format	28
nc_get_varm_longlong	125	nc_inq_grp_parent	44
nc_get_varm_schar	125	nc_inq_grpname	41
nc_get_varm_short	125	nc_inq_grpname_full	42
nc_get_varm_string	125	nc_inq_grpname_len	43
nc_get_varm_text	125	nc_inq_grps	38
nc_get_varm_ubyte	125	nc_inq_libvers	13
nc_get_varm_uchar	125	nc_inq_libvers, example	13
nc_get_varm_uint	125	nc_inq_natts	28
nc_get_varm_ulonglong	125	nc_inq_ncid	37
nc_get_vars	122	nc_inq_ndims	28
nc_get_vars_type	122	nc_inq_nvars	28
nc_get_vars_type, example	122	nc_inq_opaque	74
nc_get_vars_double	122	nc_inq_type	54
nc_get_vars_float	122	nc_inq_typeids	53
nc_get_vars_int	122	nc_inq_unlimdim	28
nc_get_vars_long	122	nc_inq_unlimdims	51
nc_get_vars_longlong	122	nc_inq_user_type	55
nc_get_vars_schar	122	nc_inq_var	98
nc_get_vars_short	122	nc_inq_var, example	98
nc_get_vars_string	122	nc_inq_var, typical use	5
nc_get_vars_text	122	nc_inq_var_chunking	89
nc_get_vars_ubyte	122	nc_inq_var_deflate	94
nc_get_vars_uchar	122	nc_inq_var_endian	97
nc_get_vars_uint	122	nc_inq_var_fill	92
nc_get_vars_ulonglong	122	nc_inq_var_fletcher32	95
nc_get_vars_ushort	122	nc_inq_varid	98
nc_inq Family	28	nc_inq_varid, example	98
nc_inq Family, example	28	nc_inq_varid, typical use	4
nc_inq, typical use	5	nc_inq_varids	39
nc_inq_att Family	140	nc_inq_varname	98
nc_inq_att Family, example	140	nc_inq_varnatts	98
nc_inq_att, typical use	5	nc_inq_varndims	98
nc_inq_compound	61	nc_inq_vartype	98
nc_inq_compound_field	65	nc_insert_array_compound	59
nc_inq_compound_fielddim_sizes	69	nc_insert_compound	58
nc_inq_compound_fieldindex	66	nc_insert_enum	77
nc_inq_compound_fieldname	66	NC_MPIO	18
nc_inq_compound_fieldndims	68	NC_MPIPOSIX	18
nc_inq_compound_fieldoffset	67	NC_NETCDF4	22
nc_inq_compound_fielddtype	67	NC_NOCLOBBER	13, 16, 18
nc_inq_compound_name	63	NC_NOWRITE	19, 21, 22
nc_inq_compound_nfields	64	nc_open	19
nc_inq_compound_size	64	nc_open, example	19
nc_inq_dim	49	nc_open_par	22
nc_inq_dim Family	49	nc_put_att, typical use	3, 6
nc_inq_dim Family, example	49	nc_put_att_type	137
nc_inq_dim, typical use	5	nc_put_att_type, example	137
nc_inq_dimid	48, 49	nc_put_var	102
nc_inq_dimid, example	48	nc_put_var, typical use	3
		nc_put_var_type	102

nc_put_var_ type, example.....	102	nc_put_varm_short	110
nc_put_var_double.....	102	nc_put_varm_string.....	110
nc_put_var_float.....	102	nc_put_varm_text	110
nc_put_var_int.....	102	nc_put_varm_ubyte.....	110
nc_put_var_long.....	102	nc_put_varm_uchar.....	110
nc_put_var_longlong.....	102	nc_put_varm_uint	110
nc_put_var_schar.....	102	nc_put_varm_ulonglong.....	110
nc_put_var_short.....	102	nc_put_varm_ushort.....	110
nc_put_var_string.....	102	nc_put_vars	107
nc_put_var_text	102	nc_put_vars_ type	107
nc_put_var_ubyte.....	102	nc_put_vars_ type, example.....	107
nc_put_var_uchar.....	102	nc_put_vars_double.....	107
nc_put_var_uint	102	nc_put_vars_float.....	107
nc_put_var_ulonglong.....	102	nc_put_vars_int.....	107
nc_put_var_ushort.....	102	nc_put_vars_long.....	107
nc_put_var1.....	100	nc_put_vars_longlong.....	107
nc_put_var1_ type.....	100	nc_put_vars_schar	107
nc_put_var1_ type, example.....	100	nc_put_vars_short	107
nc_put_var1_double.....	100	nc_put_vars_string.....	107
nc_put_var1_float.....	100	nc_put_vars_text	107
nc_put_var1_int	100	nc_put_vars_ubyte.....	107
nc_put_var1_long.....	100	nc_put_vars_uchar.....	107
nc_put_var1_longlong.....	100	nc_put_vars_uint	107
nc_put_var1_schar.....	100	nc_put_vars_ulonglong.....	107
nc_put_var1_short.....	100	nc_put_vars_ushort.....	107
nc_put_var1_string.....	100	nc_redef.....	23
nc_put_var1_text	100	nc_redef, example.....	23
nc_put_var1_ubyte.....	100	nc_redef, typical use.....	6
nc_put_var1_uchar.....	100	nc_rename_att.....	145
nc_put_var1_uint	100	nc_rename_att, example.....	145
nc_put_var1_ulonglong.....	100	nc_rename_dim.....	50
nc_put_var1_ushort.....	100	nc_rename_dim, example.....	50
nc_put_vara.....	104	nc_rename_var	133
nc_put_vara_ type.....	104	nc_rename_var, example.....	133
nc_put_vara_ type, example.....	104	nc_set_default_format.....	34
nc_put_vara_double.....	104	nc_set_default_format, example.....	34
nc_put_vara_float.....	104	nc_set_fill.....	32
nc_put_vara_int	104	nc_set_fill, example.....	32
nc_put_vara_long.....	104	NC.SHARE.....	6, 13, 16
nc_put_vara_longlong.....	104	NC.SHARE, and buffering.....	3
nc_put_vara_schar.....	104	NC.SHARE, in nc_open.....	21
nc_put_vara_short.....	104	NC.SHARE, in nc_open.....	19
nc_put_vara_string.....	104	nc_strerror	12
nc_put_vara_text	104	nc_strerror, example.....	12
nc_put_vara_ubyte.....	104	nc_strerror, introduction.....	7
nc_put_vara_uchar.....	104	NC.STRING, freeing.....	132
nc_put_vara_uint	104	NC.STRING, using.....	131
nc_put_vara_ulonglong.....	104	nc_sync	30
nc_put_vara_ushort.....	104	nc_sync, example.....	30
nc_put_varm	110	nc_var_par_access	135
nc_put_varm_ type.....	110	nc_var_par_access, example.....	135
nc_put_varm_ type, example.....	110	NC.WRITE.....	19, 21, 22
nc_put_varm_double.....	110	ncid	141
nc_put_varm_float.....	110	netCDF 2 transition guide.....	159
nc_put_varm_int.....	110	netCDF library version.....	13
nc_put_varm_long.....	110	netcdf-3 error codes.....	167
nc_put_varm_longlong.....	110	netcdf-4 error codes.....	169
nc_put_varm_schar.....	110		

O

opaque type 73

P

parallel access 10

parallel example 10

R

reading attributes 142

reading entire variable 117

reading netCDF dataset with known names 4

reading netCDF dataset with unknown names... 5

reading single value 115

renaming attributes 145

renaming variable 133

S

single value, reading 115

string arrays 129

strings in classic model 130

subsamped array, writing 107

T

templates, code 3

transition guide, netCDF 2 159

U

user defined types 53

user defined types, overview 53

V

variable length array type, overview 53

variable length arrays 70

variable, copying 134

variable, renaming 133

variable, writing entire 102

variables, adding 6

variables, checksum 94

variables, chunking 87

variables, contiguous 87

variables, creating 85

variables, endian 96

variables, fill 90

variables, fletcher32 94

variables, getting name 98

variables, inquiring about 98

variables, number of 28

variables, setting deflate 93

varid 141

version of netCDF, discovering 13

version, format 28

VLEN 70**VLEN**, defining 70, 71, 72**W**

write errors 7

write fill mode, setting 32

writing array section 104

writing attributes 137

writing character-string data 130

writing entire variable 102

writing mapped array 110

writing mapped array section 125

writing single value 100

writing subsampled array 107

X**XDR** library 159**xtypep** 141**Z**

zero byte 130

zero length edge 159

zero valued count vector 159

