NetCDF User's Guide for Fortran 90

An Access Interface for Self-Describing, Portable Data
Version 3.5
March 2002

Russ Rew, Unidata Program Center, and
Robert Pincus, NOAA/CIRES Climate Diagnostics Center

NetCDF User's Guide for Fortran 90

Foreword

Unidata is a National Science Foundation-sponsored program empowering U.S. universities, through innovative applications of computers and networks, to make the best use of atmospheric and related data for enhancing education and research. For analyzing and displaying such data, the Unidata Program Center offers universities several supported software packages developed by other organizations. Underlying these is a Unidata-developed system for acquiring and managing data in real time, making practical the Unidata principle that each university should acquire and manage its own data holdings as local requirements dictate. It is significant that the Unidata program has no data center—the management of data is a "distributed" function.

The Network Common Data Form (netCDF) software described in this guide was originally intended to provide a common data access method for the various Unidata applications. These deal with a variety of data types that encompass single-point observations, time series, regularly-spaced grids, and satellite or radar images.

The netCDF software functions as an I/O library, callable from C, FORTRAN, Fortran 90, C++, Java, Perl, Python, or other languages for which a netCDF library is available. The library stores and retrieves data in self-describing, machine-independent datasets. Each netCDF dataset can contain multidimensional, named variables (with differing types that include integers, reals, characters, bytes, etc.), and each variable may be accompanied by ancillary data, such as units of measure or descriptive text. The interface includes a method for appending data to existing netCDF datasets in prescribed ways, functionality that is not unlike a (fixed length) record structure. However, the netCDF library also allows direct-access storage and retrieval of data by variable name and index and therefore is useful only for disk-resident (or memory-resident) datasets.

NetCDF is designed to:

- Facilitate the use of common datasets by distinct applications.
- Permit datasets to be transported between or shared by dissimilar computers transparently, that is, without translation.
- Reduce the programming effort usually spent interpreting formats.
- Reduce errors arising from misinterpreting data and ancillary data.
- Facilitate using output from one application as input to another.
- Establish an interface standard that simplifies the design of new software for accessing geoscience data.

A measure of success has been achieved. NetCDF is now in use on computing platforms that range from personal computers to supercomputers and include most UNIX-based workstations. It can be used to create a complex dataset on one computer (say in FORTRAN) and retrieve that same self-describing dataset on another computer (say in C) without intermediate translations—netCDF datasets can be transferred across a network, or they can be accessed remotely using a suitable network file system or other remote access protocol.

Because we believe that the use of netCDF access in non-Unidata software will benefit Unidata's primary constituency—such use may result in more options for analyzing and displaying Unidata information—the netCDF library is distributed without licensing or other significant restrictions, and current versions can be obtained via anonymous FTP. Apparently the software has been well received by a wide range of institutions beyond the atmospheric science community, and a substantial number of open source and commercial data analysis systems can now accept netCDF datasets as input.

Several organizations have adopted netCDF as a data access standard, and there are efforts underway to support the netCDF programming interfaces as a means to store and retrieve data in other forms. We have encouraged and cooperated with these efforts.

Questions occasionally arise about the level of support provided for the netCDF software. Unidata's formal position, stated in the copyright notice which accompanies the netCDF library, is that the software is provided "as is". In practice, the software is updated from time to time, and Unidata intends to continue adapting the software to new platforms and development environments and maintaining the ability to access netCDF data for the foreseeable future. Because Unidata's mission is to serve geoscientists at U.S. universities, problems reported by that community necessarily receive the greatest attention.

We hope the reader will find the software useful and will give us feedback on its application as well as suggestions for its improvement.

David Fulker, Unidata Program Center Director

University Corporation for Atmospheric Research

Summary

The purpose of the Network Common Data Form (netCDF) interface is to allow you to create, access, and share array-oriented data in a form that is self-describing and portable. “Self-describing” means that a dataset includes information defining the data it contains. “Portable” means that the data in a dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers. Using the netCDF interface for creating new datasets makes the data portable. Using the netCDF interface in software for data access, management, analysis, and display can make the software more generally useful.

The netCDF software includes C, Fortran 77, and Fortran 90 interfaces for accessing netCDF data. These libraries are available for many common computing platforms.

Java, C++ and Perl interfaces for netCDF data access are also available from Unidata. The community of netCDF users has contributed ports of the software to additional platforms and interfaces for other programming languages as well. Source code for netCDF software libraries is freely available to encourage the sharing of both array-oriented data and the software that makes the data useful.

This User’s Guide presents the netCDF data model, but documents only the Fortran 90 interface. Separate documents are available for the other language interfaces; also see `http://www.unidata.ucar.edu/packages/netcdf/` for links to on-line versions of the C, FORTRAN, Fortran-90, Java, C++ and Perl documentation. Reference documentation in the form of UNIX ‘man’ pages for the C and FORTRAN interfaces and extensive additional information about netCDF, including pointers to other software that works with netCDF data, are also available from the netCDF home page.

1 Introduction

1.1 The NetCDF Interface

The Network Common Data Form, or netCDF, is an interface to a library of data access functions for storing and retrieving data in the form of arrays. An *array* is an n-dimensional (where n is 0, 1, 2, …) rectangular structure containing values of the same type (e.g., 8-bit character, 32-bit integer). A *scalar* (simple single value) is a 0-dimensional array.

NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable objects that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data. Generic utilities and application programs can access netCDF datasets and transform, combine, analyze, or display specified fields of the data. The development of such applications may lead to improved accessibility of data and improved reusability of software for array-oriented data management, analysis, and display.

The netCDF software implements an abstraction, which means that all operations to access and manipulate data in a netCDF dataset must use only the set of functions provided by the interface. The representation of the data is hidden from applications that use the interface, so that how the data are stored could be changed without affecting existing programs. The physical representation of netCDF data is designed to be independent of the computer on which the data were written.

Unidata supports the netCDF interfaces for C, FORTRAN, Fortran 90, Java, C++, and Perl and for various UNIX operating systems. The software is also ported and tested on a few other operating systems, with assistance from users with access to these systems, before each major release. Unidata's netCDF software is freely available to encourage its widespread use.

1.2 NetCDF Is Not a Database Management System

Why not use an existing database management system for storing array-oriented data? Relational database software has not proven to be ideally suited for the kinds of data access supported by the netCDF interface.

First, existing database systems that support the relational model do not support multidimensional objects (arrays) as a basic unit of data access. Representing arrays as relations makes some useful kinds of data access awkward and provides little support for the abstractions of multidimensional data and coordinate systems. A quite different data model is needed for array-oriented data to facilitate its retrieval, modification, mathematical manipulation and visualization.

Related to this is a second problem with general-purpose database systems: their poor performance on large arrays. Collections of satellite images, scientific model outputs and long-term global weather observations are beyond the capabilities of most database systems to organize and index for efficient retrieval.

Finally, general-purpose database systems provide, at significant cost in terms of both resources and access performance, many facilities that are not needed in the analysis, management, and display of array-oriented data. For example, elaborate update facilities, audit trails, report formatting, and mechanisms designed for transaction-processing are unnecessary for most scientific applications.

1.3 File Format

To achieve network-transparency (machine-independence), netCDF is implemented in terms of an extended version of XDR (eXternal Data Representation, see `http://www.faqs.org/rfcs/rfc1832.html`), a standard for describing and encoding data. This representation provides encoding of data into machine-independent sequences of bits. It has been implemented on a wide variety of computers, by assuming only that eight-bit bytes can be encoded and decoded in a consistent way. The IEEE 754 floating-point standard is used for floating-point data representation.

The overall structure of netCDF files is described in Chapter 9 “NetCDF File Structure and Performance,” page 83.

The details of the format are described in Appendix B “File Format Specification,” page 107. However, users are discouraged from using the format specification to develop independent low-level software for reading and writing netCDF files, because this could lead to compatibility problems if the format is ever modified.

1.4 What about Performance?

One of the goals of netCDF is to support efficient access to small subsets of large datasets. To support this goal, netCDF uses direct access rather than sequential access. This can be much more efficient when the order in which data is read is different from the order in which it was written, or when it must be read in different orders for different applications.

The amount of overhead for a portable external representation depends on many factors, including the data type, the type of computer, the granularity of data access, and how well the implementation has been tuned to the computer on which it is run. This overhead is typically small in comparison to the overall resources used by an application. In any case, the overhead of the external representation layer is usually a reasonable price to pay for portable data access.

Although efficiency of data access has been an important concern in designing and implementing netCDF, it is still possible to use the netCDF interface to access data in inefficient ways: for example, by requesting a slice of data that requires a single value from each record. Advice on how to use the interface efficiently is provided in Chapter 9 “NetCDF File Structure and Performance,” page 83.

1.5 Is NetCDF a Good Archive Format?

NetCDF can be used as a general-purpose archive format for storing arrays. Compression of data is possible with netCDF (e.g., using arrays of eight-bit or 16-bit integers to encode low-resolution floating-point numbers instead of arrays of 32-bit numbers), but the current version of netCDF was not designed to achieve optimal compression of data. Hence, using netCDF may require more space than special-purpose archive formats that exploit knowledge of particular characteristics of specific datasets.

1.6 Creating Self-Describing Data conforming to Conventions

The mere use of netCDF is not sufficient to make data “self-describing” and meaningful to both humans and machines. The names of variables and dimensions should be meaningful and conform to any relevant conventions. Dimensions should have corresponding coordinate variables where sensible.

Attributes play a vital role in providing ancillary information. It is important to use all the relevant standard attributes using the relevant conventions. Section 8.1 “Attribute Conventions,” page 69, describes reserved attributes (used by the netCDF library) and attribute conventions for generic application software.

A number of groups have defined their own additional conventions and styles for netCDF data. Descriptions of these conventions, as well as examples incorporating them can be accessed from the netCDF Conventions site, `http://www.unidata.ucar.edu/packages/netcdf/conventions.html`.

These conventions should be used where suitable. Additional conventions are often needed for local use. These should be contributed to the above netCDF conventions site if likely to interest other users in similar areas.

1.7 Background and Evolution of the NetCDF Interface

The development of the netCDF interface began with a modest goal related to Unidata’s needs: to provide a common interface between Unidata applications and real-time meteorological data. Since Unidata software was intended to run on multiple hardware platforms with access from both C and FORTRAN, achieving Unidata’s goals had the potential for providing a package that was useful in a broader context. By making the package widely available and collaborating with other organizations with similar needs, we hoped to improve the then current situation in which software for scientific data access was only rarely reused by others in the same discipline and almost never reused between disciplines (Fulker, 1988).

Important concepts employed in the netCDF software originated in a paper (Treinish and Gough, 1987) that described data-access software developed at the NASA Goddard National Space Science Data Center (NSSDC). The interface provided by this software was called the Common Data Format (CDF). The NASA CDF was originally developed as a platform-specific FORTRAN library to support an abstraction for storing arrays.

The NASA CDF package had been used for many different kinds of data in an extensive collection of applications. It had the virtues of simplicity (only 13 subroutines), independence from storage format, generality, ability to support logical user views of data, and support for generic applications.

Unidata held a workshop on CDF in Boulder in August 1987. We proposed exploring the possibility of collaborating with NASA to extend the CDF FORTRAN interface, to define a C interface, and to permit the access of data aggregates with a single call, while maintaining compatibility with the existing NASA interface.

Independently, Dave Raymond at the New Mexico Institute of Mining and Technology had developed a package of C software for UNIX that supported sequential access to self-describing array-oriented data and a “pipes and filters” (or “data flow”) approach to processing, analyzing, and displaying the data. This package also used the “Common Data Format” name, later changed to C-Based Analysis and Display System (CANDIS). Unidata learned of Raymond’s work (Raymond, 1988), and incorporated some of his ideas, such as the use of named dimensions and variables with differing shapes in a single data object, into the Unidata netCDF interface.

In early 1988, Glenn Davis of Unidata developed a prototype netCDF package in C that was layered on XDR. This prototype proved that a single-file, XDR-based implementation of the CDF interface could be achieved at acceptable cost and that the resulting programs could be implemented on both UNIX and VMS systems. However, it also demonstrated that providing a small, portable, and NASA CDF-compatible FORTRAN interface with the desired generality was not practical. NASA’s CDF and Unidata’s netCDF have since evolved separately, but recent CDF versions share many characteristics with netCDF.

In early 1988, Joe Fahle of SeaSpace, Inc. (a commercial software development firm in San Diego, California), a participant in the 1987 Unidata CDF workshop, independently developed a CDF package in C that extended the NASA CDF interface in several important ways (Fahle, 1989). Like Raymond’s package, the SeaSpace CDF software permitted variables with unrelated shapes to be included in the same data object and permitted a general form of access to multidimensional arrays. Fahle’s implementation was used at SeaSpace as the intermediate form of storage for a variety of steps in their image-processing system. This interface and format have subsequently evolved into the Terascan data format.

After studying Fahle’s interface, we concluded that it solved many of the problems we had identified in trying to stretch the NASA interface to our purposes. In August 1988, we convened a small workshop to agree on a Unidata netCDF interface, and to resolve remaining open issues. Attending were Joe Fahle of SeaSpace, Michael Gough of Apple (an author of the NASA CDF software), Angel Li of the University of Miami (who had implemented our prototype netCDF software on VMS and was a potential user), and Unidata systems development staff. Consensus was reached at the workshop after some further simplifications were discovered. A document incorporating the results of the workshop into a proposed Unidata netCDF interface specification was distributed widely for comments before Glenn Davis and Russ Rew implemented the first version of the software. Comparison with other data-access interfaces and experience using netCDF are discussed in Rew and Davis (1990a), Rew and Davis (1990b), Jenter and Signell (1992), and Brown, Folk, Goucher, and Rew (1993).

In October 1991, we announced version 2.0 of the netCDF software distribution. Slight modifications to the C interface (declaring dimension lengths to be `long` rather than `int`) improved the usability of netCDF on inexpensive platforms such as MS-DOS computers, without requiring recompilation on other platforms. This change to the interface required no changes to the associated file format.

Release of netCDF version 2.3 in June 1993 preserved the same file format but added single call access to records, optimizations for accessing cross-sections involving non-contiguous data, subsampling along specified dimensions (using ‘strides’), accessing non-contiguous data (using ‘mapped array sections’), improvements to the ncdump and ncgen utilities, and an experimental C++ interface.

In version 2.4, released in February 1996, support was added for new platforms and for the C++ interface, and significant optimizations were implemented for supercomputer architectures.

FAN (File Array Notation), software providing a high-level interface to netCDF data, was made available in May 1996. The capabilities of the FAN utilities include extracting and manipulating array data from netCDF datasets, printing selected data from netCDF arrays, copying ASCII data into netCDF arrays, and performing various operations (sum, mean, maximum, minimum, product,…) on netCDF arrays. More information about FAN is available from the FAN Utilities document, `http://www.unidata.ucar.edu/packages/netcdf/fan_utils.html`.

1.8 What’s New Since the Previous Release?

This Guide documents netCDF 3, which preserves the same file format as earlier versions but includes some major changes from version 2:

- complete rewrite of the netCDF library in ANSI C;
- new type-safe C and FORTRAN interfaces;
- automatic type conversion facilities;
- significant changes in the internal architecture, resulting in higher performance and easier optimization on new platforms;
- support for all netCDF 2 function interfaces, globals variables, and behavior, for backward compatibility;
- revised documentation; and fixes for reported bugs.

1.9 Limitations of NetCDF

The netCDF data model is widely applicable to data that can be organized into a collection of named array variables with named attributes, but there are some important limitations to the model and its implementation in software. Some of these limitations are inherent in the trade-offs among conflicting requirements that netCDF embodies, but other limitations may be addressed in a future version of the software.

Currently, netCDF offers a limited number of external numeric data types: 8-, 16-, 32-bit integers, or 32- or 64-bit floating-point numbers. This limited set of sizes may use file space inefficiently compared to packing data in bit fields. For example, arrays of 9-bit values must be stored in 16-bit short integers. Storing arrays of 1- or 2-bit values in 8-bit values is even less optimal.

With the current netCDF file format, there are constraints that limit how a dataset is structured to store more than 2 gigabytes of data in a single netCDF dataset. This limitation is a result of 32-bit offsets currently used for storing relative offsets within a file. Since one of the goals of netCDF is portable data and there are still many computing platforms that can't deal with files larger than 2 Gbytes, it is best to keep files that must be portable below this limit. Neveretheless, it is possible to store terabytes of data in a single netCDF file, as discussed in 9.3 “Large File Support,” page 84.

Another limitation of the current model is that only one unlimited (changeable) dimension is permitted for each netCDF data set. Multiple variables can share an unlimited dimension, but then they must all grow together. Hence the netCDF model does not permit variables with several unlimited dimensions or the use of multiple unlimited dimensions in different variables within the same dataset. Hence variables that have non-rectangular shapes (for example, ragged arrays) cannot be represented conveniently.

The extent to which data can be completely self-describing is limited: there is always some assumed context without which sharing and archiving data would be impractical. NetCDF permits storing meaningful names for variables, dimensions, and attributes; units of measure in a form that can be used in computations; text strings for attribute values that apply to an entire data set; and simple kinds of coordinate system information. But for more complex kinds of metadata (for example, the information necessary to provide accurate georeferencing of data on unusual grids or from satellite images), it is often necessary to develop conventions.

Specific additions to the netCDF data model might make some of these conventions unnecessary or allow some forms of metadata to be represented in a uniform and compact way. For example, adding explicit georeferencing to the netCDF data model would simplify elaborate georeferencing conventions at the cost of complicating the model. The problem is finding an appropriate trade-off between the richness of the model and its generality (i.e., its ability to encompass many kinds of data). A data model tailored to capture the shared context among researchers within one discipline may not be appropriate for sharing or combining data from multiple disciplines.

The netCDF data model does not support nested data structures such as trees, nested arrays, or other recursive structures, primarily because the current FORTRAN interface must be able to read and write any netCDF data set. Through use of indirection and conventions it is possible to represent some kinds of nested structures, but the result may fall short of the netCDF goal of self-describing data.

Finally, the current implementation limits concurrent access to a netCDF dataset. One writer and multiple readers may access data in a single dataset simultaneously, but there is no support for multiple concurrent writers.

1.10 Future Plans for NetCDF

Current plans are to add transparent data packing, improved concurrency support, and the ability to access datasets larger than 2 Gigabytes. Other desirable extensions that may be added, if practical, include access to data by key or coordinate value, support for efficient structure changes (e.g., new variables and attributes), support for pointers to data cross-sections in other datasets, nested arrays (allowing representation of ragged arrays, trees and other recursive data structures), and multiple unlimited dimensions.

References

1. Brown, S. A, M. Folk, G. Goucher, and R. Rew, "Software for Portable Scientific Data Management," *Computers in Physics*, American Institute of Physics, Vol. 7, No. 3, May/June 1993.
2. Davies, H. L., "FAN - An array-oriented query language," Second Workshop on Database Issues for Data Visualization (Visualization 1995), Atlanta, Georgia, IEEE, October 1995.
3. Fahle, J., *TeraScan Applications Programming Interface*, SeaSpace, San Diego, California, 1989.
4. Fulker, D. W., "The netCDF: Self-Describing, Portable Files---a Basis for 'Plug-Compatible' Software Modules Connectable by Networks," *ICSU Workshop on Geophysical Informatics*, Moscow, USSR, August 1988.
5. Fulker, D. W., "Unidata Strawman for Storing Earth-Referencing Data," *Seventh International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, New Orleans, La., American Meteorology Society, January 1991.
6. Gough, M. L., *NSSDC CDF Implementer's Guide (DEC VAX/VMS) Version 1.1*, National Space Science Data Center, 88-17, NASA/Goddard Space Flight Center, 1988.
7. Jenter, H. L. and R. P. Signell, "NetCDF: A Freely-Available Software-Solution to Data-Access Problems for Numerical Modelers," *Proceedings of the American Society of Civil Engineers Conference on Estuarine and Coastal Modeling*, Tampa, Florida, 1992.
8. Raymond, D. J., "A C Language-Based Modular System for Analyzing and Displaying Gridded Numerical Data," *Journal of Atmospheric and Oceanic Technology*, **5**, 501-511, 1988.
9. Rew, R. K. and G. P. Davis, "The Unidata netCDF: Software for Scientific Data Access," *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, Anaheim, California, American Meteorology Society, February 1990.
10. Rew, R. K. and G. P. Davis, "NetCDF: An Interface for Scientific Data Access," *Computer Graphics and Applications*, IEEE, pp. 76-82, July 1990.
11. Rew, R. K. and G. P. Davis, "Unidata's netCDF Interface for Data Access: Status and Plans," *Thirteenth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, Anaheim, California, American Meteorology Society, February 1997.
12. Treinish, L. A. and M. L. Gough, "A Software Package for the Data Independent Management of Multi-Dimensional Data," *EOS Transactions*, American Geophysical Union, **68**, 633-635, 1987.

2 Components of a NetCDF Dataset

2.1 The NetCDF Data Model

A netCDF dataset contains *dimensions*, *variables*, and *attributes*, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset. The netCDF library allows simultaneous access to multiple netCDF datasets which are identified by dataset ID numbers, in addition to ordinary file names.

A netCDF dataset contains a symbol table for variables containing their name, data type, rank (number of dimensions), dimensions, and starting disk address. Each element is stored at a disk address which is a linear function of the array indices (subscripts) by which it is identified. Hence, these indices need not be stored separately (as in a relational database). This provides a fast and compact storage method.

2.1.1 Naming Conventions

The names of dimensions, variables and attributes consist of arbitrary sequences of alphanumeric characters (as well as underscore '_' and hyphen '-'), beginning with a letter or underscore. (However names commencing with underscore are reserved for system use.) Case is significant in netCDF names.

2.1.2 Network Common Data Form Language (CDL)

We will use a small netCDF example to illustrate the concepts of the netCDF data model. This includes dimensions, variables, and attributes. The notation used to describe this simple netCDF object is called CDL (network Common Data form Language), which provides a convenient way of describing netCDF datasets. The netCDF system includes utilities for producing human-oriented CDL text files from binary netCDF datasets and vice versa.

```
netcdf example_1 {  // example of CDL notation for a netCDF dataset

dimensions:         // dimension names and lengths are declared first
        lat = 5, lon = 10, level = 4, time = unlimited;

variables:          // variable types, names, shapes, attributes
        float   temp(time,level,lat,lon);
                    temp:long_name     = "temperature";
                    temp:units         = "celsius";
        float   rh(time,lat,lon);
                    rh:long_name = "relative humidity";
                    rh:valid_range = 0.0, 1.0;      // min and max
        int     lat(lat), lon(lon), level(level);
                    lat:units       = "degrees_north";
                    lon:units       = "degrees_east";
```

```
                  level:units     = "millibars";
        short   time(time);
                  time:units      = "hours since 1996-1-1";
        // global attributes
                  :source = "Fictional Model Output";

data:                  // optional data assignments
        level   = 1000, 850, 700, 500;
        lat     = 20, 30, 40, 50, 60;
        lon     = -160,-140,-118,-96,-84,-52,-45,-35,-25,-15;
        time    = 12;
        rh      =.5,.2,.4,.2,.3,.2,.4,.5,.6,.7,
                 .1,.3,.1,.1,.1,.1,.5,.7,.8,.8,
                 .1,.2,.2,.2,.2,.5,.7,.8,.9,.9,
                 .1,.2,.3,.3,.3,.3,.7,.8,.9,.9,
                  0,.1,.2,.4,.4,.4,.4,.7,.9,.9;
}
```

The CDL notation for a netCDF dataset can be generated automatically by using `ncdump`, a utility program described later (see Section 10.5 "`ncdump`," page 94). Another netCDF utility, `ncgen`, generates a netCDF dataset (or optionally C or FORTRAN source code containing calls needed to produce a netCDF dataset) from CDL input (see Section 10.4 "`ncgen`," page 93).

The CDL notation is simple and largely self-explanatory. It will be explained more fully as we describe the components of a netCDF dataset. For now, note that CDL statements are terminated by a semicolon. Spaces, tabs, and newlines can be used freely for readability. Comments in CDL follow the characters '`//`' on any line. A CDL description of a netCDF dataset takes the form

```
netCDF name {
  dimensions: …
  variables: …
  data: …
}
```

where the *name* is used only as a default in constructing file names by the `ncgen` utility. The CDL description consists of three optional parts, introduced by the keywords `dimensions`, `variables`, and `data`. NetCDF dimension declarations appear after the `dimensions` keyword, netCDF variables and attributes are defined after the `variables` keyword, and variable data assignments appear after the `data` keyword.

2.2 Dimensions

A dimension may be used to represent a real physical dimension, for example, time, latitude, longitude, or height. A dimension might also be used to index other quantities, for example station or model-run-number.

A netCDF dimension has both a *name* and a *length*. A dimension length is an arbitrary positive integer, except that one dimension in a netCDF dataset can have the length `UNLIMITED`.

Such a dimension is called the *unlimited dimension* or the *record dimension*. A variable with an unlimited dimension can grow to any length along that dimension. The unlimited dimension index is like a record number in conventional record-oriented files. A netCDF dataset can have at most one unlimited dimension, but need not have any. If a variable has an unlimited dimension, that dimension must be the most significant (slowest changing) one. Thus any unlimited dimension must be the first dimension in a CDL shape and the last dimension in corresponding Fortran-90 array declarations.

CDL dimension declarations may appear on one or more lines following the CDL keyword `dimensions`. Multiple dimension declarations on the same line may be separated by commas. Each declaration is of the form *name* = *length*.

There are four dimensions in the above example: `lat`, `lon`, `level`, and `time`. The first three are assigned fixed lengths; `time` is assigned the length `UNLIMITED`, which means it is the *unlimited* dimension.

The basic unit of named data in a netCDF dataset is a *variable*. When a variable is defined, its *shape* is specified as a list of dimensions. These dimensions must already exist. The number of dimensions is called the *rank* (a.k.a. *dimensionality*). A scalar variable has rank 0, a vector has rank 1 and a matrix has rank 2.

It is possible to use the same dimension more than once in specifying a variable shape (but this was not possible in previous netCDF versions). For example, `correlation(instrument, instrument)` could be a matrix giving correlations between measurements using different instruments. But data whose dimensions correspond to those of physical space/time should have a shape comprising different dimensions, even if some of these have the same length.

2.3 Variables

Variables are used to store the bulk of the data in a netCDF dataset. A *variable* represents an array of values of the same type. A scalar value is treated as a 0-dimensional array. A variable has a name, a data type, and a shape described by its list of dimensions specified when the variable is created. A variable may also have associated attributes, which may be added, deleted or changed after the variable is created.

A variable external data type is one of a small set of netCDF *types* that have the names `NF90_BYTE` (with synonym `NF90_INT1`), `NF90_CHAR`,`NF90_SHORT` (with synonym `NF90_INT2`), `NF90_INT (with synonym NF90_INT4)`, `NF90_FLOAT` (with synonyms `NF90_REAL and NF90_REAL4`), and `NF90_DOUBLE` (with synonym `NF90_REAL8`) in the Fortran-90 interface.

In the CDL notation, these types are given the simpler names `byte`, `char`, `short`, `int`, `float`, and `double`. `real` may be used as a synonym for `float` in the CDL notation. `long` is a deprecated synonym for `int`. The exact meaning of each of the types is discussed in Section 3.1 “NetCDF external data types,” page 15.

CDL variable declarations appear after the `variable` keyword in a CDL unit. They have the form *type variable_name* (*dim_name_1, dim_name_2, …*);

for variables with dimensions, or

type variable_name;

for scalar variables.

In the above CDL example there are six variables. As discussed below, four of these are coordinate variables. The remaining variables (sometimes called *primary variables*), `temp` and `rh`, contain what is usually thought of as the data. Each of these variables has the unlimited dimension `time` as its first dimension, so they are called *record variables*. A variable that is not a record variable has a fixed length (number of data values) given by the product of its dimension lengths. The length of a record variable is also the product of its dimension lengths, but in this case the product is variable because it involves the length of the unlimited dimension, which can vary. The length of the unlimited dimension is the number of records.

2.3.1 Coordinate Variables

It is legal for a variable to have the same name as a dimension. Such variables have no special meaning to the netCDF library. However there is a convention that such variables should be treated in a special way by software using this library.

A variable with the same name as a dimension is called a *coordinate variable*. It typically defines a physical coordinate corresponding to that dimension. The above CDL example includes the coordinate variables `lat`, `lon`, `level` and `time`, defined as follows:

```
        int     lat(lat), lon(lon), level(level);
        short   time(time);
…
data:
        level   = 1000, 850, 700, 500;
        lat     = 20, 30, 40, 50, 60;
        lon     = -160,-140,-118,-96,-84,-52,-45,-35,-25,-15;
        time    = 12;
```

These define the latitudes, longitudes, barometric pressures and times corresponding to positions along these dimensions. Thus there is data at altitudes corresponding to 1000, 850, 700 and 500 millibars; and at latitudes 20, 30, 40, 50 and 60 degrees north. Note that each coordinate variable is a vector and has a shape consisting of just the dimension with the same name.

A position along a dimension can be specified using an *index*. This is an integer with a minimum value of 1 for Fortran-90 programs. Thus the 700 millibar level would have an index value of 3 in the example above.

If a dimension has a corresponding coordinate variable, then this provides an alternative, and often more convenient, means of specifying position along it. Current application packages that make use of coordinate variables commonly assume they are numeric vectors and strictly monotonic (all values are different and either increasing or decreasing).

2.4 Attributes

NetCDF *attributes* are used to store data about the data (*ancillary data* or *metadata*), similar in many ways to the information stored in data dictionaries and schema in conventional database systems. Most attributes provide information about a specific variable. These are identified by the name (or ID) of that variable, together with the name of the attribute.

Some attributes provide information about the dataset as a whole and are called *global* attributes. These are identified by the attribute name together with a blank variable name (in CDL) or a special null "global variable" ID (in C or Fortran).

An attribute has an associated variable (the null "global variable" for a global attribute), a name, a data type, a length, and a value. The current version treats all attributes as vectors; scalar values are treated as single-element vectors.

Conventional attribute names should be used where applicable. New names should be as meaningful as possible.

The external type of an attribute is specified when it is created. The types permitted for attributes are the same as the netCDF external data types for variables. Attributes with the same name for different variables should sometimes be of different types. For example, the attribute `valid_max` specifying the maximum valid data value for a variable of type `int` should be of type `int`, whereas the attribute `valid_max` for a variable of type `double` should instead be of type `double`.

Attributes are more dynamic than variables or dimensions; they can be deleted and have their type, length, and values changed after they are created, whereas the netCDF interface provides no way to delete a variable or to change its type or shape.

The CDL notation for defining an attribute is

variable_name:attribute_name = *list_of_values*;

for a variable attribute, or

:attribute_name = *list_of_values*;

for a global attribute. The type and length of each attribute are not explicitly declared in CDL; they are derived from the values assigned to the attribute. All values of an attribute must be of the same type. The notation used for constant values of the various netCDF types is discussed later (see Section 10.3 "CDL Notation for Data Constants," page 92).

In the netCDF example (see Section 2.1.2 "Network Common Data Form Language (CDL)," page 9), `units` is an attribute for the variable `lat` that has a 13-character array value '`degrees_north`'. And `valid_range` is an attribute for the variable `rh` that has length 2 and values '`0.0`' and '`1.0`'.

One global attribute---`source`---is defined for the example netCDF dataset. This is a character array intended for documenting the data. Actual netCDF datasets might have more global attributes to document the origin, history, conventions, and other characteristics of the dataset as a whole.

Most generic applications that process netCDF datasets assume standard attribute conventions and it is strongly recommended that these be followed unless there are good reasons for not doing so. See Section 8.1 “Attribute Conventions,” page 69, for information about `units`, `long_name`, `valid_min`, `valid_max`, `valid_range`, `scale_factor`, `add_offset`, `_FillValue`, and other conventional attributes.

Attributes may be added to a netCDF dataset long after it is first defined, so you don’t have to anticipate all potentially useful attributes. However adding new attributes to an existing dataset can incur the same expense as copying the dataset. See Chapter 9 “NetCDF File Structure and Performance,” page 83, for a more extensive discussion.

2.5 Differences between Attributes and Variables

In contrast to variables, which are intended for bulk data, attributes are intended for ancillary data, or information about the data. The total amount of ancillary data associated with a netCDF object, and stored in its attributes, is typically small enough to be memory-resident. However variables are often too large to entirely fit in memory and must be split into sections for processing.

Another difference between attributes and variables is that variables may be multidimensional. Attributes are all either scalars (single-valued) or vectors (a single, fixed dimension).

Variables are created with a name, type, and shape before they are assigned data values, so a variable may exist with no values. The value of an attribute must be specified when it is created, so no attribute ever exists without a value.

A variable may have attributes, but an attribute cannot have attributes. Attributes assigned to variables may have the same units as the variable (for example, `valid_range`) or have no units (for example, `scale_factor`). If you want to store data that requires units different from those of the associated variable, it is better to use a variable than an attribute. More generally, if data require ancillary data to describe them, are multidimensional, require any of the defined netCDF dimensions to index their values, or require a significant amount of storage, that data should be represented using variables rather than attributes.

3 Data

This chapter discusses the six primitive netCDF external data types, the kinds of data access supported by the netCDF interface, and how data structures other than arrays may be implemented in a netCDF dataset.

3.1 NetCDF external data types

The external types supported by the netCDF interface are:

`char`	8-bit characters intended for representing text.
`byte`	8-bit signed or unsigned integers (see discussion below).
`short`	16-bit signed integers.
`int`	32-bit signed integers.
`float or real`	32-bit IEEE floating-point.
`double`	64-bit IEEE floating-point.

These types were chosen to provide a reasonably wide range of trade-offs between data precision and number of bits required for each value. These external data types are independent from whatever internal data types are supported by a particular machine and language combination.

These types are called “external”, because they correspond to the portable external representation for netCDF data. When a program reads external netCDF data into an internal variable, the data is converted, if necessary, into the specified internal type. Similarly, if you write internal data into a netCDF variable, this may cause it to be converted to a different external type, if the external type for the netCDF variable differs from the internal type.

The separation of external and internal types and automatic type conversion have several advantages. You need not be aware of the external type of numeric variables, since automatic conversion to or from any desired numeric type is available. You can use this feature to simplify code, by making it independent of external types, using a sufficiently wide internal type, e.g., double precision, for numeric netCDF data of several different external types. Programs 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 a future version of netCDF, when new external types will be added for packed data for which there may be no natural corresponding internal type, for example, packed 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. For example, an internal short integer type may not be able to hold data stored externally as an integer. When accessing an array of values, a range 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 a single-precision floating-point variable, for example, no error results unless the magnitude of the double precision value exceeds the representable range of single-precision floating point numbers 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 adequate precision.

The names for the primitive external data types (`byte`, `char`, `short`, `int`, `float` or `real`, and `double`) are reserved words in CDL, so the names of variables, dimensions, and attributes must not be type names.

It is possible to interpret `byte` data as either signed (-128 to 127) or unsigned (0 to 255). However, when reading byte data to be converted into other numeric types, it is interpreted as signed.

See Section 2.3 “Variables,” page 11, for the correspondence between netCDF external data types and the data types of a language.

3.2 Data Access

To access (read or write) netCDF data you specify an open netCDF dataset, a netCDF variable, and information (e.g., indices) identifying elements of the variable. The name of the access function corresponds to the internal type of the data. If the internal type has a different representation from the external type of the variable, a conversion between the internal type and external type will take place when the data is read or written.

Access to data is *direct*, which means you can access a small subset of data from a large dataset efficiently, without first accessing all the data that precedes it. Reading and writing data by specifying a variable, instead of a position in a file, makes data access independent of how many other variables are in the dataset, making programs immune to data format changes that involve adding more variables to the data.

In the C, FORTRAN, and Fortran 90 interfaces, datasets are not specified by name every time you want to access data, but instead by a small integer called a dataset ID, obtained when the dataset is first created or opened.

Similarly, a variable is not specified by name for every data access either, but by a variable ID, a small integer used to identify each variable in a netCDF dataset.

3.2.1 Forms of Data Access

The netCDF interface supports several forms of direct access to data values in an open netCDF dataset. We describe each of these forms of access in order of increasing generality:

- access to all elements;
- access to individual elements, specified with an *index vector*;
- access to array sections, specified with an *index vector*, and *count vector*;
- access to subsampled array sections, specified with an *index vector*, *count vector*, and *stride vector*; and
- access to mapped array sections, specified with an *index vector*, *count vector*, *stride vector*, and an *index mapping vector.*

The four types of vector (*index vector*, *count vector*, *stride vector* and *index mapping vector*) each have one element for each dimension of the variable. Thus, for an n-dimensional variable (rank = n), n-element vectors are needed. If the variable is a scalar (no dimensions), these vectors are ignored.

An *array section* is a “slab” or contiguous rectangular block that is specified by two vectors. The *index vector* gives the indices of the element in the corner closest to the origin. The *count vector* gives the lengths of the edges of the slab along each of the variable’s dimensions, in order. The number of values accessed is the product of these edge lengths.

A *subsampled array section* is similar to an *array section*, except that an additional *stride vector* is used to specify sampling. This vector has an element for each dimension giving the length of the strides to be taken along that dimension. For example, a stride of 4 means every fourth value along the corresponding dimension. The total number of values accessed is again the product of the elements of the *count vector.*

A *mapped array section* is similar to a *subsampled array section* except that an additional *index mapping vector* allows one to specify how data values associated with the netCDF variable are arranged in memory. The offset of each value from the reference location, is given by the sum of the products of each index (of the imaginary internal array which would be used if there were no mapping) by the corresponding element of the index mapping vector. The number of values accessed is the same as for a *subsampled array section*.

The use of mapped array sections is discussed more fully below, but first we present an example of the more commonly used array-section access.

3.2.2 An Example of Array-Section Access

Assume that in our earlier example of a netCDF dataset (see Section 2.1.2 “Network Common Data Form Language (CDL),” page 9), we wish to read a cross-section of all the data for the `temp` variable at one level (say, the second), and assume that there are currently three records (`time` values) in the netCDF dataset. Recall that the dimensions are defined as

```
    lat = 5, lon = 10, level = 4, time = unlimited;
```

and the variable `temp` is declared as

```
     float   temp(time, level, lat, lon);
```

in the CDL notation.

In Fortran-90, the dimensions are reversed from the CDL declaration with the first dimension varying fastest and the record dimension as the last dimension of a record variable. Thus the Fortran-90 declaration for a variable that holds data for only one level is

```
   INTEGER, PARAMETER :: LATS = 5, LONS = 10, LEVELS = 1, TIMES = 3
   …
   REAL, DIMENSION(LONS, LATS, LEVELS, TIMES) :: TEMP
```

To specify the block of data that represents just the second level, all times, all latitudes, and all longitudes, we need to provide a start index and some edge lengths. The start index should be (1, 1, 2, 1) in Fortran-90, because we want to start at the beginning of each of the `time`, `lon`, and `lat` dimensions, but we want to begin at the second value of the `level` dimension. The edge lengths should be (10, 5, 1, 3) in Fortran-90, since we want to get data for all three `time` values, only one `level` value, all five `lat` values, and all 10 `lon` values. We should expect to get a total of 150 floating-point values returned (3 * 1 * 5 * 10), and should provide enough space in our array for this many. The order in which the data will be returned is with the first dimension, `LON`, varying fastest:

```
        TEMP( 1, 1, 2, 1)
        TEMP( 2, 1, 2, 1)
        TEMP( 3, 1, 2, 1)
        TEMP( 4, 1, 2, 1)

              …

        TEMP( 8, 5, 2, 3)
        TEMP( 9, 5, 2, 3)
        TEMP(10, 5, 2, 3)
```

Different dimension orders for the C, FORTRAN, or other language interfaces do not reflect a different order for values stored on the disk, but merely different orders supported by the procedural interfaces to the languages. In general, it does not matter whether a netCDF dataset is written using the C, FORTRAN, or another language interface; netCDF datasets written from any supported language may be read by programs written in other supported languages.

3.2.3 More on General Array Section Access

The use of mapped array sections allows non-trivial relationships between the disk addresses of variable elements and the addresses where they are stored in memory. For example, a matrix in memory could be the transpose of that on disk, giving a quite different order of elements. In a regular array section, the mapping between the disk and memory addresses is trivial: the structure of the in-memory values (i.e., the dimensional lengths and their order) is identical to that of the array section. In a mapped array section, however, an *index mapping vector* is used to define the mapping between indices of netCDF variable elements and their memory addresses.

With mapped array access, the offset (number of array elements) from the origin of a memory-resident array to a particular point is given by the *inner product*[1] of the index mapping vector with the point's *coordinate offset vector.* A point's *coordinate offset vector* gives, for each dimension, the offset from the origin of the containing array to the point. In Fortran-90, the values of a point's coordinate offset vector are one less than the corresponding values of the point's coordinate vector, e.g., the array element A(3,5) has coordinate offset vector [2, 4].

The index mapping vector for a regular array section would have—in order from most rapidly varying dimension to most slowly—a constant 1, the product of that value with the edge length of the most rapidly varying dimension of the array section, then the product of that value with the edge length of the next most rapidly varying dimension, and so on. In a mapped array, however, the correspondence between netCDF variable disk locations and memory locations can be different.

A detailed example of mapped array access is presented in the description of the interfaces for mapped array access. See Section 7.6 "Reading Data Values: NF90_GET_VAR," page 61.

Note that, although the netCDF abstraction allows the use of subsampled or mapped array-section access, their use is not required. If you do not need these more general forms of access, you may ignore these capabilities and use single value access or regular array section access instead.

3.3 Type Conversion

Each netCDF variable has an external type, specified when the variable is first defined. This external type determines whether the data is intended for text or numeric values, and if numeric, the range and precision of numeric values.

If the netCDF external type for a variable is `char`, only character data representing text strings can be written to or read from the variable. No automatic conversion of text data to a different representation is supported.

If the type is numeric, however, the netCDF library allows you to access the variable data as a different type and provides automatic conversion between the numeric data in memory and the data in the netCDF variable. For example, if you write a program that deals with all numeric data as double-precision floating point values, you can read netCDF data into double-precision arrays without knowing or caring what the external type of the netCDF variables are. On reading netCDF data, integers of various sizes and single-precision floating-point values will all be converted to double-precision, if you use the data access interface for double-precision values. Of course, you can avoid automatic numeric conversion by using the netCDF interface for a value type that corresponds to the external data type of each netCDF variable, where such value types exist.

1. The *inner product* of two vectors [x0, x1, …, xn] and [y0, y1, …, yn] is just x0*y0 + x1*y1 + … + xn*yn.

The automatic numeric conversions performed by netCDF are easy to understand, because they behave just like assignment of data of one type to a variable of a different type. For example, if you read floating-point netCDF data as integers, the result is truncated towards zero, just as it would be if you assigned a floating-point value to an integer variable. Such truncation is an example of the loss of precision that can occur in numeric conversions.

Converting from one numeric type to another may result in an error if the target type is not capable of representing the converted value. For example, an integer may not be able to hold data stored externally as an IEEE floating-point number. When accessing an array of values, a range 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 result in an error. For example, if you read double precision values into an integer, no error results unless the magnitude of the double precision value exceeds the representable range of integers 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.

Whether a range error occurs in writing a large floating-point value near the boundary of representable values may be depend on the platform. The largest floating-point value you can write to a netCDF float variable is the largest floating-point number representable on your system that is less than 2 to the 128th power. The largest double precision value you can write to a double variable is the largest double-precision number representable on your system that is less than 2 to the 1024th power.

This automatic conversion and separation of external data representation from internal data types will become even more important in a future version of netCDF, 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.

3.4 Data Structures

The only kind of data structure directly supported by the netCDF abstraction is a collection of named arrays with attached vector attributes. NetCDF is not particularly well-suited for storing linked lists, trees, sparse matrices, ragged arrays or other kinds of data structures requiring pointers.

It is possible to build other kinds of data structures from sets of arrays by adopting various conventions regarding the use of data in one array as pointers into another array. The netCDF library won't provide much help or hindrance with constructing such data structures, but netCDF provides the mechanisms with which such conventions can be designed.

The following example stores a ragged array `ragged_mat` using an attribute `row_index` to name an associated index variable giving the index of the start of each row. In this example, the first row contains 12 elements, the second row contains 7 elements (19 - 12), and so on.

```
        float   ragged_mat(max_elements);
                ragged_mat:row_index = "row_start";
        int     row_start(max_rows);
data:
        row_start   = 0, 12, 19, …
```

As another example, netCDF variables may be grouped within a netCDF dataset by defining attributes that list the names of the variables in each group, separated by a conventional delimiter such as a space or comma. Using a naming convention for attribute names for such groupings permits any number of named groups of variables. A particular conventional attribute for each variable might list the names of the groups of which it is a member. Use of attributes, or variables that refer to other attributes or variables, provides a flexible mechanism for representing some kinds of complex structures in netCDF datasets.

4 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, see 10.4 “ncgen,” page 93, 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.

4.1 Creating a NetCDF Dataset

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

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

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

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

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

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

4.2 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 calls to read data from those variables in a netCDF dataset is:

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

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

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

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

4.3 Reading a netCDF Dataset with Unknown Names

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

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

```
NF90_OPEN                  ! open existing netCDF dataset
  …
NF90_Inquire               ! find out what is in it
     …
   NF90_Inquire_Dimension ! get dimension names, lengths
     …
   NF90_Inquire_Variable  ! get variable names, types, shapes
        …
      NF90_INQ_ATTNAME    ! get attribute names
        …
      NF90_Inquire_Attribute ! get attribute values
        …
      NF90_GET_ATT        ! get attribute values
        …
   NF90_GET_VAR           ! get values of variables
     …
NF90_CLOSE                 ! close netCDF dataset
```

As in the previous example, a single call opens the existing netCDF dataset, returning a netCDF ID. This netCDF ID is given to the `NF90_Inquire` routine, which returns the number of dimensions, the number of variables, the number of global attributes, and the ID of the unlimited dimension, if there is one.

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

Dimension IDs use consecutive integers, beginning at 1. Also dimensions, once created, cannot be deleted. Therefore, knowing the number of dimension IDs in a netCDF dataset means knowing all the dimension IDs: they are the integers 1, 2, 3, …up to the number of dimensions. For each dimension ID, a call to the inquire function `NF90_Inquire_Dimension` returns the dimension name and length.

Variable IDs are also assigned from consecutive integers 1, 2, 3, … up to the number of variables. These can be used in `NF90_Inquire_Variable` calls to find out the names, types, shapes, and the number of attributes assigned to each variable.

Once the number of attributes for a variable is known, successive calls to `NF90_INQ_ATTNAME` return the name for each attribute given the netCDF ID, variable ID, and attribute number. Armed with the attribute name, a call to NF90_Inquire_Variablereturns its type and length. Given the type and length, you can allocate enough space to hold the attribute values. Then a call to `NF90_GET_ATT` returns the attribute values.

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

4.4 Writing Data in an Existing NetCDF Dataset

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

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

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

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

Next, calls to `NF90_INQ_VARID` get the variable ID from the name, for each variable you want to write. Then each call to `NF90_PUT_VAR` writes data into a specified variable, either a single value at a time, or a whole set of values at a time, depending on which variant of the interface is used. The calls used to overwrite values of non-record variables are the same as are used to overwrite values of record variables or append new data to record variables. The difference is that, with record variables, the record dimension is extended by writing values that don’t yet exist in the dataset. This extends all record variables at once, writing “fill values” for record variables for which the data has not yet been written (but see 7.8 “Fill Values,” page 67 for how to specify different behavior).

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

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

4.5 Adding New Dimensions, Variables, Attributes

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

```
NF90_OPEN              ! open existing netCDF dataset
  …
NF90_REDEF             ! put it into define mode
    …
  NF90_DEF_DIM         ! define additional dimensions (if any)
    …
  NF90_DEF_VAR         ! define additional variables (if any)
    …
  NF90_PUT_ATT         ! define other attributes (if any)
    …
NF90_ENDDEF            ! check definitions, leave define mode
    …
  NF90_PUT_VAR         ! provide new variable values
    …
NF90_CLOSE             ! close netCDF dataset
```

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

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

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

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

4.6 Error Handling

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

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

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

4.7 Compiling and Linking with the NetCDF Library

Details of how to compile and link a program that uses the netCDF C or FORTRAN interfaces differ, depending on the operating system, the available compilers, and where the netCDF library and include files are installed. 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 Fortran 90 procedure or module which references netCDF constants or procedures must have access to the module information created when the netCDF module was compiled. The suffix for this file depends on the compiler, but is often `.MOD`. Most Fortran 90 compilers do not allow you to specify an alternative location for this file as you might the location of external libraries. The simplest solution, therefore, is to create a symbolic link from the directory in which your code resides to the location of the pre-compiled netCDF module. For example:

```
ln -s /usr/local/netcdf/src/f90/netcdf.mod .
```

You may then compile source files which reference netCDF constants or procedures.

```
f90 -c mymodule.f90
```

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:

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

5 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 nonnegative integer returned when you create or open the dataset. A netCDF ID is much like a file descriptor in C or a logical unit number in FORTRAN. In any single program, the netCDF IDs of distinct open netCDF datasets are distinct. A single netCDF dataset may be opened multiple times and will then have multiple distinct netCDF IDs; however at most one of the open instances of a single netCDF dataset should permit writing. When an open netCDF dataset is closed, the ID is no longer associated with a netCDF dataset.

Functions that deal with the netCDF library include:

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

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

- Create, given dataset name and whether to overwrite or not.
- Open for access, given dataset name and read or write intent.
- Put into define mode, to add dimensions, variables, or attributes.
- Take out of define mode, checking consistency of additions.
- Close, writing to disk if required.
- Inquire about the number of dimensions, number of variables, number of global attributes, and ID of the unlimited dimension, if any.
- Synchronize to disk to make sure it is current.
- Set and unset *nofill* mode for optimized sequential writes.

After a summary of conventions used in describing the netCDF interfaces, the rest of this chapter presents a detailed description of the interfaces for these operations.

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

The examples follow a simple convention for error handling, always checking the error status returned from each netCDF function call and calling a `HANDLE_ERR` subroutine in case an error was detected. For an example of such a subroutine, see Section 5.2 "Get error message corresponding to error status: `NF90_STRERROR`," page 30.

5.2 Get error message corresponding to error status: `NF90_STRERROR`

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

Usage

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

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

Errors

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

Example

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

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

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

5.3 Get netCDF library version: `NF90_INQ_LIBVERS`

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

Usage

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

Errors

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

Example

Here is an example using `NF90_INQ_LIBVERS` to print the version of the netCDF library with which the program is linked:

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

5.4 Create a NetCDF dataset: `NF90_CREATE`

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

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

Usage

```
function nf90_create(path, cmode, ncid)
  character (len = *), intent(in   ) :: path
  integer,             intent(in   ) :: cmode
  integer, optional,   intent(in   ) :: initialsize
  integer, optional,   intent(inout) :: chunksize
  integer,             intent(  out) :: ncid
  integer                            :: nf90_create
```

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

`cmode`	The creation mode. A zero value (or `NF90_CLOBBER`) specifies the default behavior: overwrite any existing dataset with the same file name and buffer and cache accesses for efficiency. Otherwise, the creation mode is `NF90_NOCLOBBER`, `NF90_SHARE`, or `IOR(NF90_NOCLOBBER, NF90_SHARE)`. Setting the `NF90_NOCLOBBER` flag means you do not want to clobber (overwrite) an existing dataset; an error (`NF90_EEXIST`) is returned if the specified dataset already exists. The `NF90_SHARE` flag is appropriate when one process may be writing the dataset and one or more other processes reading the dataset concurrently; it means that dataset accesses are not buffered and caching is limited. Since the buffering scheme is optimized for sequential access, programs that do not access data sequentially may see some performance improvement by setting the `NF_SHARE` flag.
`ncid`	Returned netCDF ID.

The following optional arguments allow additional performance tuning.

`initialsize`	The initial size of the file (in bytes) at creation time. A value of 0 causes the file size to be computed when `nf90_enddef` is called.
`chunksize`	Controls a space versus time trade-off, memory allocated in the netcdf library versus number of system calls. Because of internal requirements, the value may not be set to exactly the value requested. The actual value chosen is returned. The library chooses a system-dependent default value if `NF90_SIZEHINT_DEFAULT` is supplied as input. If the "preferred I/O block size" is available from the `stat()` system call as member `st_blksize` this value is used. Lacking that, twice the system pagesize is used. Lacking a call to discover the system pagesize, the default chunksize is set to 8192 bytes. The chunksize is a property of a given open netcdf descriptor ncid, it is not a persistent property of the netcdf dataset.

Errors

If no errors were detected, `NF90_CREATE` returns the value `NF90_NOERR`. Possible causes of errors include:

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

Example

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

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

5.5 Open a NetCDF Dataset for Access: `NF90_OPEN`

The function `NF90_OPEN` opens an existing netCDF dataset for access in data mode.

Usage

```
function nf90_open(path, mode, ncid, chunksize)
  character (len = *), intent(in   ) :: path
  integer,             intent(in   ) :: mode
  integer,             intent(  out) :: ncid
  integer, optional,   intent(inout) :: chunksize
  integer                            :: nf90_open
```

`path`	File name for netCDF dataset to be opened.
`mode`	A zero value (or `NF90_NOWRITE`) specifies the default behavior: open the dataset with read-only access, buffering and caching accesses for efficiency Otherwise, the creation mode is `NF90_WRITE`, `NF90_SHARE`, or `IOR(NF90_WRITE, NF90_SHARE)`. Setting the `NF90_WRITE` flag opens the dataset with read-write access. (“Writing” means any kind of change to the dataset, including appending or changing data, adding or renaming dimensions, variables, and attributes, or deleting attributes.) The `NF_SHARE` flag is appropriate when one process may be writing the dataset and one or more other processes reading the dataset concurrently; it means that dataset accesses are not buffered and caching is limited. Since the buffering scheme is optimized for sequential access, programs that do not access data sequentially may see some performance improvement by setting the `NF90_SHARE` flag.
`ncid`	Returned netCDF ID.

The following optional argument allows additional performance tuning.

`chunksize`	Controls a space versus time trade-off, memory allocated in the netcdf library versus number of system calls. Because of internal requirements, the value may not be set to exactly the value requested. The actual value chosen is returned. The library chooses a system-dependent default value if `NF90_SIZEHINT_DEFAULT` is supplied as input. If the "preferred I/O block size" is available from the `stat()` system call as member `st_blksize` this value is used. Lacking that, twice the system pagesize is used. Lacking a call to discover the system pagesize, the default chunksize is set to 8192 bytes. The chunksize is a property of a given open netcdf descriptor ncid, it is not a persistent property of the netcdf dataset.

Errors

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

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

Example

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

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

5.6 Put Open NetCDF Dataset into Define Mode: `NF90_REDEF`

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

Usage

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

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

Errors

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

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

Example

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

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

5.7 Leave Define Mode: NF90_ENDDEF

The function NF90_ENDDEF takes an open netCDF dataset out of define mode. The changes made to the netCDF dataset while it was in define mode are checked and committed to disk if no problems occurred. Non-record variables may be initialized to a “fill value” as well (see Section 5.12 “Set Fill Mode for Writes: NF90_SET_FILL,” page 41). 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. See Chapter 9 “NetCDF File Structure and Performance,” page 83, for a more extensive discussion.

Usage

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

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

The following arguments allow additional performance tuning. Note: these arguments expose internals of the netcdf version 1 file format, and may not be available in future netcdf implementations.
The current netcdf file format has three sections: the "header" section, the data section for fixed size variables, and the data section for variables which have an unlimited dimension (record variables). The header begins at the beginning of the file. The index (offset) of the beginning of the other two sections is contained in the header. Typically, there is no space between the sections. This causes copying overhead to accrue if one wishes to change the size of the sections, as may happen when changing the names of things, text attribute values, adding attributes or adding variables. Also, for buffered i/o, there may be advantages to aligning sections in certain ways.

	The following parameters allow one to control costs of future calls to `nf90_redef` or `nf90_enddef` by requesting that some space be available at the end of the section. The default value for both arguments is 0.
`h_minfree`	Size of the pad (in bytes) at the end of the "header" section.
`v_minfree`	Size of the pad (in bytes) at the end of the data section for fixed size variables.
	The align parameters allow one to set the alignment of the beginning of the corresponding sections. The beginning of the section is rounded up to an index which is a multiple of the align parameter. The flag value `NF90_ALIGN_CHUNK` tells the library to use the chunksize (see above) as the align parameter. The default value for both arguments is 4 bytes.
`v_align`	The alignment of the beginning of the data section for fixed size variables.
`r_align`	The alignment of the beginning of the data section for variables which have an unlimited dimension (record variables).

Errors

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

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

Example

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

```
use netcdf
implicit none
integer :: ncid, status
…
```

```
   status = nf90_create("foo.nc", nf90_noclobber, ncid)
   if (status /= nf90_noerr) call handle_err(status)
   …  !  create dimensions, variables, attributes
   status = nf90_enddef(ncid)
   if (status /= nf90_noerr) call handle_err(status)
```

5.8 Close an Open NetCDF Dataset: `NF90_CLOSE`

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

Usage

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

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

Errors

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

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

Example

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

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

5.9 Inquire about an Open NetCDF Dataset: NF90_Inquire

The `NF90_Inquire` subroutine returns information about an open netCDF dataset, given its netCDF ID. The subroutine can be called from either define mode or data mode, and returns values for any or all of the following: the number of dimensions, the number of variables, the number of global attributes, and the dimension ID of the dimension defined with unlimited length, if any.

No I/O is performed when `NF90_Inquire` is called, since the required information is available in memory for each open netCDF dataset.

Usage

```
function nf90_Inquire(ncid, nDimensions, nVariables, nAttributes, &
                      unlimitedDimId)
  integer,            intent( in) :: ncid
  integer, optional, intent(out) :: nDimensions, nVariables, nAttributes, &
                                    unlimitedDimId
  integer                        :: nf90_Inquire
```

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`nDimensions`	Returned number of dimensions defined for this netCDF dataset.
`nVariables`	Returned number of variables defined for this netCDF dataset.
`nAttributes`	Returned number of global attributes defined for this netCDF dataset.
`unlimited-DimID`	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.

Errors

Function `NF90_Inquire` returns the value `NF90_NOERR` if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

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

Example

Here is an example using `Nf90_Inquire` to find out about a netCDF dataset named `foo.nc`:

```
   use netcdf
   implicit none
   integer :: ncid, status, nDims, nVars, nGlobalAtts, unlimDimID
   …
   status = nf90_open(“foo.nc”, nf90_nowrite, ncid)
   if (status /= nf90_noerr) call handle_err(status)
   …
   status = Nf90_Inquire(ncid, nDims, nVars, nGlobalAtts, unlimdimid)
```

```
if (status /= nf90_noerr) call handle_err(status)
status = Nf90_Inquire(ncid, nDimensions = nDims, &
                      unlimitedDimID = unlimdimid)
if (status /= nf90_noerr) call handle_err(status)
```

5.10 Synchronize an Open NetCDF Dataset to Disk: `NF90_SYNC`

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

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

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

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

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

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

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

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

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90 _CREATE`.

Errors

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

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

Example

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

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

5.11 Back Out of Recent Definitions: `NF90_ABORT`

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

Usage

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

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

Errors

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

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

Example

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

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

5.12 Set Fill Mode for Writes: `NF90_SET_FILL`

This function is intended for advanced usage, to optimize writes under some circumstances described below. The function `NF90_SET_FILL` sets the *fill mode* for a netCDF dataset open for writing and returns the current fill mode in a return parameter. The fill mode can be specified as either `NF90_FILL` or `NF90_NOFILL`. The default behavior corresponding to `NF90_FILL` is that data is pre-filled with fill values, that is fill values are written when you create non-record variables or when you write a value beyond data that has not yet been written. This makes it possible to detect attempts to read data before it was written. See Section 7.8 "Fill Values," page 67, for more information on the use of fill values. See Section 8.1 "Attribute Conventions," page 69, for information about how to define your own fill values.

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

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

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

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

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

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

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`fillmode`	Desired fill mode for the dataset, either `NF90_NOFILL` or `NF90_FILL`.
`old_mode`	Returned current fill mode of the dataset before this call, either `NF90_NOFILL` or `NF90_FILL`.

Errors

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

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

Example

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

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

6 Dimensions

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

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

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

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

Operations supported on dimensions are:

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

6.1 Create a Dimension: `NF90_DEF_DIM`

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

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`name`	Dimension name. Must begin with an alphabetic character, followed by zero or more alphanumeric characters including the underscore (‘_’). Case is significant.
`len`	Length of dimension; that is, number of values for this dimension as an index to variables that use it. This should be either a positive integer or the predefined constant `NF90_UNLIMITED`.
`dimid`	Returned dimension ID.

Errors

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

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

Example

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

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

6.2 Get a Dimension ID from Its Name: `NF90_INQ_DIMID`

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

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`name`	Dimension name, a character string beginning with a letter and followed by any sequence of letters, digits, or underscore ('_') characters. Case is significant in dimension names.
`dimid`	Returned dimension ID.

Errors

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

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

Example

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

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

6.3 Inquire about a Dimension: NF90_Inquire_Dimension

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

Usage

```
function nf90_Inquire_Dimension(ncid, dimid, name, len)
  integer,                       intent( in) :: ncid, dimid
```

```
 character (len = *), optional, intent(out) :: name
 integer,             optional, intent(out) :: len
 integer                                    :: nf90_Inquire_Dimension
```

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`dimid`	Dimension ID, for example from a previous call to `NF90_INQ_DIMID` or `NF90_DEF_DIM`.
`name`	Returned dimension name. The caller must allocate space for the returned name. The maximum possible length, in characters, of a dimension name is given by the predefined constant `NF90_MAX_NAME`.
`len`	Returned length of dimension. For the unlimited dimension, this is the current maximum value used for writing any variables with this dimension, that is the maximum record number.

Errors

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

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

Example

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

```
use netcdf
implicit none
integer :: ncid, status, LatDimID, RecordDimID
integer :: nLats, nRecords
character(len = nf90_max_name) :: RecordDimName
…
status = nf90_open(“foo.nc”, nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
! Get ID of unlimited dimension
status = nf90_Inquire(ncid, unlimitedDimId = RecordDimID)
if (status /= nf90_noerr) call handle_err(status)
…
status = nf90_inq_dimid(ncid, “Lat”, LatDimID)
if (status /= nf90_noerr) call handle_err(status)
! How many values of “lat” are there?
status = nf90_Inquire_Dimension(ncid, LatDimID, len = nLats)
if (status /= nf90_noerr) call handle_err(status)
! What is the name of the unlimited dimension, how many records are there?
status = nf90_Inquire_Dimension(ncid, RecordDimID, &
                                name = RecordDimName, len = Records)
```

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

6.4 Rename a Dimension: `NF90_RENAME_DIM`

The function `NF90_`renames an existing dimension in a netCDF dataset open for writing. If the new name is longer than the old name, the netCDF dataset must be in define mode. You cannot rename a dimension to have the same name as another dimension.

Usage

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

ncid	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
dimid	Dimension ID, from a previous call to `NF90_INQ_DIMID` or `NF90_DEF_DIM`.
name	New name for the dimension.

Errors

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

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

Example

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

```
   use netcdf
   implicit none
   integer :: ncid, status, LatDimID
   …
   status = nf90_open(“foo.nc”, nf90_write, ncid)
   if (status /= nf90_noerr) call handle_err(status)
   …
   ! Put in define mode so we can rename the dimension
   status = nf90_redef(ncid)
   if (status /= nf90_noerr) call handle_err(status)
   ! Get the dimension ID for “Lat”...
   status = nf90_inq_dimid(ncid, “Lat”, LatDimID)
```

```
if (status /= nf90_noerr) call handle_err(status)
! ... and change the name to “Latitude”.
status = nf90_rename_dim(ncid, LatDimID, “Latitude”)
if (status /= nf90_noerr) call handle_err(status)
! Leave define mode
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

7 Variables

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

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

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

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

Attributes (see Chapter 8 “Attributes,” page 69) may be associated with a variable to specify such properties as units.

Operations supported on variables are:

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

7.1 Language Types Corresponding to NetCDF External Data Types

The following table gives the netCDF external data types and the corresponding type constants for defining variables in the Fortran 90 interface:

netCDF/CDL Data Type	Fortran 90 API Mnemonic	Bits
`byte`	`NF90_BYTE`	8
`char`	`NF90_CHAR`	8
`short`	`NF90_SHORT`	16
`int`	`NF90_INT`	32
`float`	`NF90_FLOAT`	32
`double`	`NF90_DOUBLE`	64

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

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

7.2 Create a Variable: `NF90_DEF_VAR`

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

Usage

```
function nf90_def_var(ncid, name, xtype, dimids, varid)
  integer,               intent( in) :: ncid
  character (len = *),   intent( in) :: name
  integer,               intent( in) :: xtype
  integer, dimension(:), intent( in) :: dimids
  integer                            :: nf90_def_var
```

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`name`	Name for this variable. Must begin with an alphabetic character, which is followed by zero or more alphanumeric characters including the underscore (‘_’). Case is significant.
`xtype`	The external type for this variable, one of the set of predefined netCDF external data types: `NF90_BYTE`, `NF90_CHAR`, `NF90_SHORT`, `NF90_INT`, `NF90_FLOAT`, or `NF90_DOUBLE`.

`dimids`	Dimension ID(s) corresponding to this variable's dimension(s). If the ID of the unlimited dimension is included, it must be last. Optional argument `dimids` may be a vector or, if the variable has only one dimension, a scalar; if the argument is omitted the netCDF variable is defined as a scalar.
`varid`	Returned variable ID

Errors

`NF90_DEF_VAR` returns the value `NF90_NOERR` if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

- The netCDF dataset is not in define mode.
- The specified variable name is the name of another existing variable.
- The specified type is not a valid netCDF type.
- The specified number of dimensions is negative or more than the constant `NF90_MAX_VAR_DIMS`, the maximum number of dimensions permitted for a netCDF variable.
- One or more of the dimension IDs in the list of dimensions is not a valid dimension ID for the netCDF dataset.
- The number of variables would exceed the constant `NF90_MAX_VARS`, the maximum number of variables permitted in a netCDF dataset.
- The specified netCDF ID does not refer to an open netCDF dataset.

Example

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

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

7.3 Get a Variable ID from Its Name: NF90_INQ_VARID

The function `NF90_INQ_VARID` returns the ID of a netCDF variable, given its name.

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`name`	Variable name for which ID is desired.
`varid`	Returned variable ID.

Errors

`NF90_INQ_VARID` returns the value `NF90_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 `NF90_INQ_VARID` to find out the ID of a variable named `rh` in an existing netCDF dataset named `foo.nc`:

```
use netcdf
implicit none
integer :: status, ncid, RhVarId
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, RhVarId)
if(status /= nf90_NoErr) call handle_err(status)
```

7.4 Get Information about a Variable from Its ID: NF90_Inquire_Variable

`NF90_Inquire_Variable` returns information about a netCDF variable given its ID. Information about a variable includes its name, type, number of dimensions, a list of dimension IDs describing the shape of the variable, and the number of variable attributes that have been assigned to the variable.

Usage

```
function nf90_Inquire_Variable(ncid, varid, name, xtype, ndims, dimids, nAtts)
  integer,                             intent( in) :: ncid, varid
  character (len = *),   optional, intent(out) :: name
  integer,               optional, intent(out) :: xtype, ndims
  integer, dimension(*), optional, intent(out) :: dimids
  integer,               optional, intent(out) :: nAtts
  integer                                      :: nf90_Inquire_Variable
```

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`varid`	Variable ID.
`name`	Returned variable name. The caller must allocate space for the returned name. The maximum possible length, in characters, of a variable name is given by the predefined constant `NF90_MAX_NAME`.
`xtype`	Returned external type for this variable, one of the set of predefined netCDF external data types. The valid netCDF external data types are `NF90_BYTE`, `NF90_CHAR`, `NF90_SHORT`, `NF90_INT`, `NF90_FLOAT`, and `NF90_DOUBLE`.
`ndims`	Returned number of dimensions for this variable. 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 `NDIMS` dimension IDs corresponding to the variable dimensions. The caller must allocate enough space for a vector of at least `NDIMS` integers to be returned. The maximum possible number of dimensions for a variable is given by the predefined constant `NF90_MAX_VAR_DIMS`.
`natts`	Returned number of variable attributes assigned to this variable. Note that you can get the number of global attributes by using the `NF90_GLOBAL` pseudo-variable ID

Errors

Function `NF90_Inquire_Variable` returns the value `NF90_NOERR` if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

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

Example

Here is an example using `NF90_Inquire_Variable` to find out about a variable named `rh` in an existing netCDF dataset named `foo.nc`:

```
use netcdf
implicit none
```

```
integer                                 :: status, ncid, &
                                           RhVarId       &
                                           numDims, numAtts
integer, dimension(nf90_max_var_dims) :: rhDimIds
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_error(status)
…
status = nf90_inq_varid(ncid, “rh”, RhVarId)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Var(ncid, RhVarId, ndims = numDims, natts = numAtts)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Var(ncid, RhVarId, dimids = rhDimIds(:numDims))
if(status /= nf90_NoErr) call handle_err(status)
```

7.5 Writing Data Values: NF90_PUT_VAR

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

Take care when using the simplest forms of this interface with record variables when you don’t specify how many records are to be written. If you try to write all the values of a record variable into a netCDF file that has no record data yet (hence has 0 records), nothing will be written. Similarly, if you try to write all of a record variable but there are more records in the file than you assume, more data may be written to the file than you supply, which may result in a segmentation violation.

Usage

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

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

`values`
: The data value(s) to be written. The data may be of any type, and may be a scalar or an array of any rank.
 You cannot put `CHARACTER` data into a numeric variable or numeric data into a text variable. For numeric data, if the type of data differs from the netCDF variable type, type conversion will occur. See Section 3.3 “Type Conversion,” page 24, for details.

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

`count`
: A vector of integers specifying the number of indices selected along each dimension. To write a single value, for example, specify `count` as `(1, 1, …, 1)`. The elements of `count` correspond, in order, to the variable’s dimensions. Hence, if the variable is a record variable, the last element of `count` corresponds to a count of the number of records to write.
By default, `count(:numDims) = shape(values)` and
`count(numDims + 1:) = 1`, where `numDims = size(shape(values))`.

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

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

```
map = (/ 1, (product(edgeLengths(:i)), &
             i = 1, size(edgeLengths) - 1) /),
```

that is, there is no mapping.
Use of Fortran 90 intrinsic functions (including `reshape`, `transpose`, and `spread`) may let you avoid using this argument.

Errors

NF90_PUT_VAR returns the value NF90_NOERR if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

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

(As noted above, another possible source of error is using this interface to write all the values of a record variable without specifying the number of records. If there are a different number of records in the file than you assume, the amount of data written may be different from what you expect!)

Example

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

```
use netcdf
implicit none
integer :: ncId, rhVarId, status
…
status = nf90_open(“foo.nc”, nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…ß
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_put_var(ncid, rhVarId, 0.5, start = (/ 4, 3, 2 /) )
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
integer                              :: ncId, rhVarId,status,          &
                                        lonDimID, latDimId, timeDimId, &
                                        numLons, numLats, numTimes,    &
                                        i
integer, dimension(nf90_max_var_dims) :: dimIDs
```

```
…
status = nf90_open(“foo.nc”, nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
! How big is the netCDF variable, that is, what are the lengths of
!   its constituent dimensions?
status = nf90_Inquire_Variable(ncid, rhVarId, dimids = dimIDs)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Dimension(ncid, dimIDs(1), len = numLons)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Dimension(ncid, dimIDs(2), len = numLats)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Dimension(ncid, dimIDs(3), len = numTimes)
if(status /= nf90_NoErr) call handle_err(status)
…
! Make a temporary array the same shape as the netCDF variable.
status = nf90_put_var(ncid, rhVarId, &
                      reshape( &
                       (/ (0.5, i = 1, numLons * numLats * numTimes) /) , &
                       shape = (/ numLons, numLats, numTimes /) )
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 10, numLats = 5, numTimes = 3
real, dimension(numLons, numLats) &
                   :: rhValues
…
status = nf90_open(“foo.nc”, nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
! Fill in all values at the last time
rhValues(:, :) = 0.5
status = nf90_put_var(ncid, rhVarId,rhvalues,       &
                      start = (/ 1, 1, numTimes /), &
                      count = (/ numLats, numLons, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
```

```
integer             :: ncId, rhVarId, status
integer, parameter :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) &
                    :: rhValues = 0.5
…
status = nf90_open(“foo.nc”, nf90_Write, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
…
! Fill in every other value using an array section
status = nf90_put_var(ncid, rhVarId, rhValues(::2, ::2), &
                       stride = (/ 2, 2 /))
if(status /= nf90_NoErr) call handle_err(status)
```

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

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

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

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

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

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

```
use netcdf
```

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

7.6 Reading Data Values: NF90_GET_VAR

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

Take care when using the simplest forms of this interface with record variables when you don’t specify how many records are to be read. If you try to read all the values of a record variable into an array but there are more records in the file than you assume, more data will be read than you expect, which may cause a segmentation violation.

Usage

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

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

`values`	The data value(s) to be read. The data may be of any type, and may be a scalar or an array of any rank. You cannot read `CHARACTER` data from a numeric variable or numeric data from a text variable. For numeric data, if the type of data differs from the netCDF variable type, type conversion will occur. See Section 3.3 “Type Conversion,” page 24, for details.
`start`	A vector of integers specifying the index in the variable from which the first (or only) of the data values will be read. The indices are relative to 1, so for example, the first data value of a variable would have index `(1, 1, …, 1)`. The elements of `start` correspond, in order, to the variable’s dimensions. Hence, if the variable is a record variable, the last index would correspond to the starting record number for writing the data values. By default, `start(:) = 1`.
`count`	A vector of integers specifying the number of indices selected along each dimension. To read a single value, for example, specify `count` as `(1, 1, …, 1)`. The elements of `count` correspond, in order, to the variable’s dimensions. Hence, if the variable is a record variable, the last element of `count` corresponds to a count of the number of records to read. By default, `count(:numDims) = shape(values)` and `count(numDims + 1:) = 1`, where `numDims = size(shape(values))`.
`stride`	A vector of integers that specifies the sampling interval along each dimension of the netCDF variable. The elements of the stride vector correspond, in order, to the netCDF variable’s dimensions (`stride(1)` gives the sampling interval along the most rapidly varying dimension of the netCDF variable). Sampling intervals are specified in type-independent units of elements (a value of 1 selects consecutive elements of the netCDF variable along the corresponding dimension, a value of 2 selects every other element, etc.). By default, `stride(:) = 1`.
`map`	A vector of integers that specifies the mapping between the dimensions of a netCDF variable and the in-memory structure of the internal data array. The elements of the index mapping vector correspond, in order, to the netCDF variable’s dimensions (`map(1)` gives the distance between elements of the internal array corresponding to the most rapidly varying dimension of the netCDF variable). Distances between elements are specified in units of elements. By default, `edgeLengths = shape(values)`, and `map = (/ 1, (product(edgeLengths(:i)), &` `             i = 1, size(edgeLengths) - 1) /)`, that is, there is no mapping. Use of Fortran 90 intrinsic functions (including `reshape`, `transpose`, and `spread`) may let you avoid using this argument.

Errors

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

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

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

Example

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

```
use netcdf
implicit none
integer :: ncId, rhVarId, status
real    :: rhValue
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
-
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_get_var(ncid, rhVarId, rhValue, start = (/ 4, 3, 2 /) )
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
integer                               :: ncId, rhVarId, &
                                         lonDimID, latDimId, timeDimId, &
                                         numLons, numLats, numTimes,    &
                                         status
integer, dimension(nf90_max_var_dims) :: dimIDs
real, dimension(:, :, :), allocatable :: rhValues
```

```
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
! How big is the netCDF variable, that is, what are the lengths of
!   its constituent dimensions?
status = nf90_Inquire_Variable(ncid, rhVarId, dimids = dimIDs)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Dimension(ncid, dimIDs(1), len = numLons)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Dimension(ncid, dimIDs(2), len = numLats)
if(status /= nf90_NoErr) call handle_err(status)
status = nf90_Inquire_Dimension(ncid, dimIDs(3), len = numTimes)
if(status /= nf90_NoErr) call handle_err(status)
allocate(rhValues(numLons, numLats, numTimes))
…
status = nf90_get_var(ncid, rhVarId, rhValues)
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 10, numLats = 5, numTimes = 3
real, dimension(numLons, numLats, numTimes) &
                   :: rhValues
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
!Read the values at the last time by passing an array section
status = nf90_get_var(ncid, rhVarId, rhValues(:, :, 3), &
                      start = (/ 1, 1, numTimes /),     &
                      count = (/ numLats, numLons, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
integer            :: ncId, rhVarId, status
integer, parameter :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) &
                   :: rhValues
```

```
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
…
! Read every other value into an array section
status = nf90_get_var(ncid, rhVarId, rhValues(::2, ::2) &
                      stride = (/ 2, 2 /))
if(status /= nf90_NoErr) call handle_err(status)
```

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

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

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

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

```
use netcdf
implicit none
integer                            :: ncId, rhVarId, status
integer, parameter                 :: numLons = 6, numLats = 4
real, dimension(numLons, numLats) :: rhValues
! netCDF variable has dimensions (numLats, numLons)
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
…
! Read transposed values: map vector would be (/ 1, numLats /) for
!   no transposition
status = nf90_get_var(ncid, rhVarId,rhValues, map = (/ numLons, 1 /))
if(status /= nf90_NoErr) call handle_err(status)
```

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

```
use netcdf
implicit none
integer                            :: ncId, rhVarId, status
integer, parameter                 :: numLons = 6, numLats = 4
```

```
real, dimension(numLons, numLats) :: rhValues
! netCDF variable has dimensions (numLats, numLons)
real, dimension(numLons, numLats) :: tempValues
…
status = nf90_open(“foo.nc”, nf90_NoWrite, ncid)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_inq_varid(ncid, “rh”, rhVarId)
if(status /= nf90_NoErr) call handle_err(status)
…
status = nf90_get_var(ncid, rhVarId, tempValues))
if(status /= nf90_NoErr) call handle_err(status)
rhValues(:, :) = transpose(tempValues)
```

7.7 Reading and Writing Character String Values

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

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

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

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

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

7.8 Fill Values

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

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

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

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

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

7.9 Rename a Variable: `NF90_RENAME_VAR`

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

Usage

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

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

`varid`	Variable ID.
`newname`	New name for the specified variable.

Errors

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

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

Example

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

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

8 Attributes

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

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

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

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.

8.1 Attribute Conventions

Names commencing with underscore ('_') are reserved for use by the netCDF library. Most generic applications that process netCDF datasets assume standard attribute conventions and it is strongly recommended that these be followed unless there are good reasons for not doing so. Below we list the names and meanings of recommended standard attributes that have proven useful. Note that some of these (e.g. `units`, `valid_range`, `scale_factor`) assume numeric data and should not be used with character data.

`units`	A character string that specifies the units used for the variable's data. Unidata has developed a freely-available library of routines to convert between character string and binary forms of unit specifications and to perform various useful operations on the binary forms. This library is used in some netCDF applications. Using the recommended units syntax permits data represented in conformable units to be automatically converted to common units for arithmetic operations. See Appendix A "Units," page 105, for more information.
`long_name`	A long descriptive name. This could be used for labeling plots, for example. If a variable has no `long_name` attribute assigned, the variable name should be used as a default.
`valid_min`	A scalar specifying the minimum valid value for this variable.
`valid_max`	A scalar specifying the maximum valid value for this variable.
`valid_range`	A vector of two numbers specifying the minimum and maximum valid values for this variable, equivalent to specifying values for both `valid_min` and `valid_max` attributes. Any of these attributes define the *valid range*. The attribute `valid_range` must not be defined if either `valid_min` or `valid_max` is defined. Generic applications should treat values outside the *valid range* as missing. The type of each `valid_range`, `valid_min` and `valid_max` attribute should match the type of its variable (except that for `byte` data, these can be of a signed integral type to specify the intended range). If neither `valid_min`, `valid_max` nor `valid_range` is defined then generic applications should define a valid range as follows. If the data type is byte and `_FillValue` is not explicitly defined, then the valid range should include all possible values. Otherwise, the valid range should exclude the `_FillValue` (whether defined explicitly or by default) as follows. If the `_FillValue` is positive then it defines a valid maximum, otherwise it defines a valid minimum. For integer types, there should be a difference of 1 between the `_FillValue` and this valid minimum or maximum. For floating point types, the difference should be twice the minimum possible (1 in the least significant bit) to allow for rounding error.
`scale_factor`	If present for a variable, the data are to be multiplied by this factor after the data are read by the application that accesses the data.

`add_offset`
: If present for a variable, this number is to be added to the data after it is read by the application that accesses the data. If both `scale_factor` and `add_offset` attributes are present, the data are first scaled before the offset is added. The attributes `scale_factor` and `add_offset` can be used together to provide simple data compression to store low-resolution floating-point data as small integers in a netCDF dataset. When scaled data are written, the application should first subtract the offset and then divide by the scale factor.

  When `scale_factor` and `add_offset` are used for packing, the associated variable (containing the packed data) is typically of type byte or short, whereas the unpacked values are intended to be of type float or double. The attributes `scale_factor` and `add_offset` should both be of the type intended for the unpacked data, e.g. float or double.

`_FillValue`
: The `_FillValue` attribute specifies the *fill value* used to pre-fill disk space allocated to the variable. Such pre-fill occurs unless *nofill mode* is set using `NF90_SET_FILL`. See Section 5.12 “Set Fill Mode for Writes: `NF90_SET_FILL`,” page 41, for details. The *fill value* is returned when reading values that were never written. If `_FillValue` is defined then it should be scalar and of the same type as the variable. It is not necessary to define your own `_FillValue` attribute for a variable if the default *fill value* for the type of the variable is adequate. However, use of the default fill value for data type byte is not recommended. Note that if you change the value of this attribute, the changed value applies only to subsequent writes; previously written data are not changed.

  Generic applications often need to write a value to represent undefined or missing values. The *fill value* provides an appropriate value for this purpose because it is normally outside the *valid range* and therefore treated as missing when read by generic applications. It is legal (but not recommended) for the *fill value* to be within the *valid range*.

  See Section 7.8 “Fill Values,” page 67, for more information.

`missing_value`
: This attribute is not treated in any special way by the library or conforming generic applications, but is often useful documentation and may be used by specific applications. The `missing_value` attribute can be a scalar or vector containing values indicating missing data. These values should all be outside the *valid range* so that generic applications will treat them as missing.

`signedness`
: Deprecated attribute, originally designed to indicate whether byte values should be treated as signed or unsigned. The attributes `valid_min` and `valid_max` may be used for this purpose. For example, if you intend that a byte variable store only nonnegative values, you can use `valid_min = 0` and `valid_max = 255`. This attribute is ignored by the netCDF library.

`FORTRAN_format`
: A character array providing the format that should be used by FORTRAN or Fortran 90 applications to print values for this variable. For example, if you know a variable is only accurate to three significant digits, it would be appropriate to define the `FORTRAN_format` attribute as `"(G10.3)"`.

`title`
: A global attribute that is a character array providing a succinct description of what is in the dataset.

`history`
: A global attribute for an audit trail. This is a character array with a line for each invocation of a program that has modified the dataset. Well-behaved generic netCDF applications should append a line containing: date, time of day, user name, program name and command arguments.

`Conventions`
: If present, ‘`Conventions`’ is a global attribute that is a character array for the name of the conventions followed by the dataset, in the form of a string that is interpreted as a directory name relative to a directory that is a repository of documents describing sets of discipline-specific conventions. This permits a hierarchical structure for conventions and provides a place where descriptions and examples of the conventions may be maintained by the defining institutions and groups. The conventions directory name is currently interpreted relative to the directory `pub/netcdf/Conventions/` on the host machine `ftp.unidata.ucar.edu`. Alternatively, a full URL specification may be used to name a WWW site where documents that describe the conventions are maintained.

  For example, if a group named NUWG agrees upon a set of conventions for dimension names, variable names, required attributes, and netCDF representations for certain discipline-specific data structures, they may store a document describing the agreed-upon conventions in a dataset in the `NUWG/` subdirectory of the Conventions directory. Datasets that followed these conventions would contain a global `Conventions` attribute with value `"NUWG"`.

  Later, if the group agrees upon some additional conventions for a specific subset of NUWG data, for example time series data, the description of the additional conventions might be stored in the `NUWG/Time_series/` subdirectory, and datasets that adhered to these additional conventions would use the global `Conventions` attribute with value `"NUWG/Time_series"`, implying that this dataset adheres to the NUWG conventions and also to the additional NUWG time-series conventions.

8.2 Create an Attribute: `NF90_PUT_ATT`

The function `NF90_PUT_ATT`adds or changes a variable attribute or global attribute of an open netCDF dataset. If this attribute is new, or if the space required to store the attribute is greater than before, the netCDF dataset must be in define mode.

Usage

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

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`varid`	Variable ID.
`name`	Attribute name. Must begin with an alphabetic character, followed by zero or more alphanumeric characters including the underscore ('_'). Case is significant. 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. A table of conventional attribute names is presented in the earlier chapter on the netCDF interface.
`values`	An array of attribute values. Values may be supplied as scalars or as arrays of rank one (one dimensional vectors). The external data type of the attribute is set to match the internal representation of the argument, that is if `values` is a two byte integer array, the attribute will be of type `NF90_INT2`. Fortran 90 intrinsic functions can be used to convert attributes to the desired type.

Errors

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

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

Example

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

```
use netcdf
implicit none
integer :: ncid, status, RHVarID
…
status = nf90_open(“foo.nc”, nf90_write, ncid)
if (status /= nf90_noerr) call handle_err(status)
…
! Enter define mode so we can add the attribute
status = nf90_redef(ncid)
if (status /= nf90_noerr) call handle_err(status)
! Get the variable ID for “rh”...
status = nf90_inq_varid(ncid, “rh”, RHVarID)
if (status /= nf90_noerr) call handle_err(status)
! ...  put the range attribute, setting it to eight byte reals...
status = nf90_put_att(ncid, RHVarID, “valid_range”, real((/ 0, 100 /))
! ... and the title attribute.
if (status /= nf90_noerr) call handle_err(status)
status = nf90_put_att(ncid, RHVarID, “title”, “example netCDF dataset”) )
if (status /= nf90_noerr) call handle_err(status)
! Leave define mode
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

8.3 Get Information about an Attribute: `NF90_Inquire_Att and NF90_INQ_ATTNAME`

The function `NF90_Inquire_att` returns information about a netCDF attribute given the variable ID and attribute name. Information about an attribute includes its type, length, name, and number. See `NF90_GET_ATT` for getting attribute values.

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

Usage

```
function nf90_Inquire_Attribute(ncid, varid, name, xtype, len, attnum)
  integer,             intent( in)           :: ncid, varid
   character (len = *), intent( in)           :: name
   integer,             intent(out), optional :: xtype, len, attnum
   integer                                    :: nf90_Inquire_Attribute

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

```
 integer                          :: nf90_inq_attname
```

`ncid`	NetCDF ID, from a previous call to `NF_OPEN` or `NF90_CREATE`.
`varid`	Variable ID of the attribute’s variable, or `NF90_GLOBAL` for a global attribute.
`name`	Attribute name, input except that for `NF90_INQ_ATTNAME`, this is where the attribute name is returned.
`xtype`	Returned attribute type, one of the set of predefined netCDF external data types. The valid netCDF external data types are `NF90_BYTE`, `NF90_CHAR`, `N90_SHORT`, `NF90_INT`, `NF90_FLOAT`, and `NF90_DOUBLE`.
`len`	Returned number of values currently stored in the attribute. For a string-valued attribute, this is the number of characters in the string.
`attnum`	For `NF90_INQ_ATTNAME`, the input attribute number; for `NF90_Inquire_Attribute`, the returned attribute number. The attributes for each variable are numbered from 1 (the first attribute) to `NATTS`, where `NATTS` is the number of attributes for the variable, as returned from a call to `NF90_Inquire_Variable.` (If you already know an attribute name, knowing its number is not very useful, because accessing information about an attribute requires its name.)

Errors

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

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

Example

Here is an example using `NF90_Inquire_Att` to inquire about the lengths of an attribute named `valid_range` for a netCDF variable named `rh` and a global attribute named `title` in an existing netCDF dataset named `foo.nc`:

```
use netcdf
implicit none
integer :: ncid, status
integer :: RHVarID                       ! Variable ID
integer :: validRangeLength, titleLength ! Attribute lengths
…
status = nf90_open(“foo.nc”, nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
…
```

```
   ! Get the variable ID for “rh”...
   status = nf90_inq_varid(ncid, “rh”, RHVarID)
   if (status /= nf90_noerr) call handle_err(status)
   ! ...  get the length of the “valid_range” attribute...
   status = nf90_Inquire_Att(ncid, RHVarID, “valid_range”, &
                             len = validRangeLength)
   if (status /= nf90_noerr) call handle_err(status)
   ! ... and the global title attribute.
   status = nf90_Inquire_Att(ncid, nf90_global, “title”, len = titleLength)
   if (status /= nf90_noerr) call handle_err(status)
```

8.4 Get Attribute’s Values: NF90_GET_ATT

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

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`varid`	Variable ID of the attribute’s variable, or `NF90_GLOBAL` for a global attribute.
`name`	Attribute name.
`values`	Returned attribute values. All elements of the vector of attribute values are returned, so you must provide enough space to hold them. If you don’t know how much space to reserve, call `NF90_Inquire_Att` first to find out the length of the attribute. If there is only a single attribute `values` may be a scalar. If the attribute is of type character `values` should be a variable of type `character` with the `len` Fortran 90 attribute set to an appropriate value (i.e. `character (len = 80) :: values`). You cannot read character data from a numeric variable or numeric data from a text variable. For numeric data, if the type of data differs from the netCDF variable type, type conversion will occur (see Section 3.3 “Type Conversion,” page 24, for details).

Errors

`NF90_GET_ATT` returns the value `NF90_NOERR` if no errors occurred. Otherwise, the returned status indicates an error. Possible causes of errors include:

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

- One or more of the attribute values are out of the range of values representable by the desired type.

Example

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

```
use netcdf
implicit none
integer              :: ncid, status
integer              :: RHVarID                       ! Variable ID
integer              :: validRangeLength, titleLength ! Attribute lengths
real, dimension(:), allocatable, &
                     :: validRange
character (len = 80) :: title
…
status = nf90_open(“foo.nc”, nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
…
! Find the lengths of the attributes
status = nf90_inq_varid(ncid, “rh”, RHVarID)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_Inquire_Att(ncid, RHVarID, “valid_range”, &
                          len = validRangeLength)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_Inquire_Att(ncid, nf90_global, “title”, len = titleLength)
if (status /= nf90_noerr) call handle_err(status)
…
!Allocate space to hold attribute values, check string lengths
allocate(validRange(validRangeLength), stat = status)
if(status /= 0 .or. len(title) < titleLength)
  print *, “Not enough space to put attribute values.”
  exit
end if
! Read the attributes.
status = nf90_get_att(ncid, RHVarID, “valid_range”, validRange)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_get_att(ncid, nf90_global, “title”, title)
if (status /= nf90_noerr) call handle_err(status)
```

8.5 Copy Attribute from One NetCDF to Another: `NF90_COPY_ATT`

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

Usage

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

`ncid_in`	The netCDF ID of an input netCDF dataset from which the attribute will be copied, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`varid_in`	ID of the variable in the input netCDF dataset from which the attribute will be copied, or `NF90_GLOBAL` for a global attribute.
`name`	Name of the attribute in the input netCDF dataset to be copied.
`ncid_out`	The netCDF ID of the output netCDF dataset to which the attribute will be copied, from a previous call to `NF90_OPEN` or `NF90_CREATE`. It is permissible for the input and output netCDF IDs to be the same. The output netCDF dataset should be in define mode if the attribute to be copied does not already exist for the target variable, or if it would cause an existing target attribute to grow.
`varid_out`	ID of the variable in the output netCDF dataset to which the attribute will be copied, or `NF90_GLOBAL` to copy to a global attribute.

Errors

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

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

Example

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

```
use netcdf
implicit none
integer :: ncid1, ncid2, status
integer :: RHVarID, avgRHVarID    ! Variable ID
…
status = nf90_open("foo.nc", nf90_nowrite, ncid1)
```

```
  if (status /= nf90_noerr) call handle_err(status)
  status = nf90_open(“bar.nc”, nf90_write, ncid2)
  if (status /= nf90_noerr) call handle_err(status)
  …
  ! Find the IDs of the variables
  status = nf90_inq_varid(ncid1, “rh”, RHVarID)
  if (status /= nf90_noerr) call handle_err(status)
  status = nf90_inq_varid(ncid1, “avgrh”, avgRHVarID)
  if (status /= nf90_noerr) call handle_err(status)
  …
  status = nf90_redef(ncid2)   ! Enter define mode
  if (status /= nf90_noerr) call handle_err(status)
  ! Copy variable attribute from “rh” in file 1 to “avgrh” in file 1
  status = nf90_copy_att(ncid1, RHVarID, “units”, ncid2, avgRHVarID)
  if (status /= nf90_noerr) call handle_err(status)
  status = nf90_enddef(ncid2)
  if (status /= nf90_noerr) call handle_err(status)
```

8.6 Rename an Attribute: `NF90_RENAME_ATT`

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

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`
`varid`	ID of the attribute’s variable, or `NF90_GLOBAL` for a global attribute
`curname`	The current attribute name.
`newname`	The new name to be assigned to the specified attribute. If the new name is longer than the current name, the netCDF dataset must be in define mode.

Errors

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

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

Example

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

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

8.7 Delete an Attribute: `NF90_DEL_ATT`

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

Usage

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

`ncid`	NetCDF ID, from a previous call to `NF90_OPEN` or `NF90_CREATE`.
`varid`	ID of the attribute’s variable, or `NF90_GLOBAL` for a global attribute.
`name`	The original attribute name.

Errors

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

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

Example

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

```
use netcdf
implicit none
integer :: ncid1, status
integer :: RHVarID         ! Variable ID
…
status = nf90_open(“foo.nc”, nf90_nowrite, ncid)
if (status /= nf90_noerr) call handle_err(status)
…
! Find the IDs of the variables
status = nf90_inq_varid(ncid, “rh”, RHVarID)
if (status /= nf90_noerr) call handle_err(status)
…
status = nf90_redef(ncid)   ! Enter define mode
if (status /= nf90_noerr) call handle_err(status)
status = nf90_del_att(ncid, RHVarID, “Units”)
if (status /= nf90_noerr) call handle_err(status)
status = nf90_enddef(ncid)
if (status /= nf90_noerr) call handle_err(status)
```

9 NetCDF File Structure and Performance

This chapter describes the file structure of a netCDF dataset in enough detail to aid in understanding netCDF performance issues.

NetCDF is a data abstraction for array-oriented data access and a software library that provides a concrete implementation of the interfaces that support that abstraction. The implementation provides a machine-independent format for representing arrays. Although the netCDF file format is hidden below the interfaces, some understanding of the current implementation and associated file structure may help to make clear why some netCDF operations are more expensive than others.

For a detailed description of the netCDF format, see Appendix B "File Format Specification," page 107. Knowledge of the format is not needed for reading and writing netCDF data or understanding most efficiency issues. Programs that use only the documented interfaces and that make no assumptions about the format will continue to work even if the netCDF format is changed in the future, because any such change will be made below the documented interfaces and will support earlier versions of the netCDF file format.

9.1 Parts of a NetCDF File

A netCDF dataset is stored as a single file comprising two parts:

- a *header*, containing all the information about dimensions, attributes, and variables except for the variable data;
- a *data* part, comprising *fixed-size data*, containing the data for variables that don't have an unlimited dimension; and *variable-size data*, containing the data for variables that have an unlimited dimension.

Both the header and data parts are represented in a machine-independent form. This form is very similar to XDR (eXternal Data Representation), extended to support efficient storage of arrays of non-byte data.

The header at the beginning of the file contains information about the dimensions, variables, and attributes in the file, including their names, types, and other characteristics. The information about each variable includes the offset to the beginning of the variable's data for fixed-size variables or the relative offset of other variables within a record. The header also contains dimension lengths and information needed to map multidimensional indices for each variable to the appropriate offsets.

This header has no usable extra space; it is only as large as it needs to be for the dimensions, variables, and attributes (including all the attribute values) in the netCDF dataset. This has the advantage that netCDF files are compact, requiring very little overhead to store the ancillary data that makes the datasets self-describing. A disadvantage of this organization is that any operation on a netCDF dataset that requires the header to grow (or, less likely, to shrink), for example adding new dimensions or new variables, requires moving the data by copying it. This expense is incurred when `NF90_ENDDEF` is called, after a previous call to `NF90_REDEF`. If you create all necessary dimensions, variables, and attributes *before* writing data, and avoid later additions and renamings of netCDF components that require more space in the header part of the file, you avoid the cost associated with later changing the header.

When the size of the header is changed, data in the file is moved, and the location of data values in the file changes. If another program is reading the netCDF dataset during redefinition, its view of the file will be based on old, probably incorrect indexes. If netCDF datasets are shared across redefinition, some mechanism external to the netCDF library must be provided that prevents access by readers during redefinition, and causes the readers to call `NF90_SYNC` before any subsequent access.

The fixed-size data part that follows the header contains all the variable data for variables that do not employ an unlimited dimension. The data for each variable is stored contiguously in this part of the file. If there is no unlimited dimension, this is the last part of the netCDF file.

The record-data part that follows the fixed-size data consists of a variable number of fixed-size records, each of which contains data for all the record variables. The record data for each variable is stored contiguously in each record.

The order in which the variable data appears in each data section is the same as the order in which the variables were defined, in increasing numerical order by netCDF variable ID. This knowledge can sometimes be used to enhance data access performance, since the best data access is currently achieved by reading or writing the data in sequential order.

9.2 The Extended XDR Layer

XDR is a standard for describing and encoding data and a library of functions for external data representation, allowing programmers to encode data structures in a machine-independent way. NetCDF employs an extended form of XDR for representing information in the header part and the data parts. This extended XDR is used to write portable data that can be read on any other machine for which the library has been implemented.

The cost of using a canonical external representation for data varies according to the type of data and whether the external form is the same as the machine’s native form for that type.

For some data types on some machines, the time required to convert data to and from external form can be significant. The worst case is reading or writing large arrays of floating-point data on a machine that does not use IEEE floating-point as its native representation.

9.3 Large File Support

It is possible to write netCDF files that exceed 2 GB on platforms that have "Large File Support" (LFS). Such files would be platform-independent to other LFS platforms, but if you call `nc_open` to access data from such a file on an older platform without LFS, you would expect a "file too large" error.

There are important constraints on the structure of large netCDF files that result from the 32-bit relative offsets that are part of the netCDF file format:

- If you don’t use the unlimited dimension, only one variable can exceed 2 Gbytes in size, but it can be as large as the underlying file system permits. It must be the last variable in the dataset, and the offset to the beginning of this variable must be less than about 2 Gbytes. For example, the structure of the data might be something like:

  ```
  netcdf bigfile1 {
     dimensions:
        x=2000;
        y=5000;
        z=10000;
     variables:
        double x(x);         // coordinate variables
        double y(y);
        double z(z);
        double var(x, y, z); // 800 Gbytes
  }
  ```

- If you use the unlimited dimension, any number of record variables may exceed 2 Gbytes in size, as long as the offset of the start of each record variable within a record is less than about 2 Gbytes. For example, the structure of the data in a 2.4 Tbyte file might be something like:

  ```
  netcdf bigfile2 {
     dimensions:
        x=2000;
        y=5000;
        z=10;
        t=UNLIMITED;         // 1000 records, for example
     variables:
        double x(x);         // coordinate variables
        double y(y);
        double z(z);
        double t(t);
                             // 3 record variables, 2.4 Gbytes per record
        double var1(t, x, y, z);
        double var2(t, x, y, z);
        double var3(t, x, y, z);
  }
  ```

9.4 The I/O Layer

An I/O layer implemented much like the C standard I/O (stdio) library is used by netCDF to read and write portable data to netCDF datasets. Hence an understanding of the standard I/O library provides answers to many questions about multiple processes accessing data concurrently, the use of I/O buffers, and the costs of opening and closing netCDF files. In particular, it is possible to have one process writing a netCDF dataset while other processes read it. Data reads and writes are no more atomic than calls to stdio `fread()` and `fwrite()`. An `NF90_SYNC` call is analogous to the `fflush` call in the C standard I/O library, writing unwritten buffered data so other processes can read it; `NF90_SYNC` also brings header changes up-to-date (for example, changes to attribute values). `NF90_SHARE` is analogous to setting a stdio stream to be unbuffered with `the _IONBF` flag to `setvbuf.`

As in the stdio library, flushes are also performed when “seeks” occur to a different area of the file. Hence the order of read and write operations can influence I/O performance significantly. Reading data in the same order in which it was written within each record will minimize buffer flushes.

You should not expect netCDF data access to work with multiple writers having the same file open for writing simultaneously.

It is possible to tune an implementation of netCDF for some platforms by replacing the I/O layer with a different platform-specific I/O layer. This may change the similarities between netCDF and standard I/O, and hence characteristics related to data sharing, buffering, and the cost of I/O operations.

The distributed netCDF implementation is meant to be portable. Platform-specific ports that further optimize the implementation for better I/O performance are practical in some cases.

9.5 UNICOS Optimization

As was mentioned in the previous section, it is possible to replace the I/O layer in order to increase I/O efficiency. This has been done for UNICOS, the operating system of Cray computers similar to the Cray Y-MP.

Additionally, it is possible for the user to obtain even greater I/O efficiency through appropriate setting of the `NETCDF_FFIOSPEC` environment variable. This variable specifies the Flexible File I/O buffers for netCDF I/O when executing under the UNICOS operating system (the variable is ignored on other operating systems). An appropriate specification can greatly increase the efficiency of netCDF I/O—to the extent that it can surpass default FORTRAN binary I/O. Possible specifications include the following:

`bufa:336:2`	2, asynchronous, I/O buffers of 336 blocks each (i.e., double buffering). This is the default specification and favors sequential I/O.
`cache:256:8`	8, synchronous, 256-block buffers. This favors larger random accesses.
`cachea:256:8:2`	8, asynchronous, 256-block buffers with a 2 block read-ahead/write-behind factor. This also favors larger random accesses.
`cachea:8:256:0`	256, asynchronous, 8-block buffers without read-ahead/write-behind. This favors many smaller pages without read-ahead for more random accesses as typified by slicing netCDF arrays.

`cache:8:256,cachea.sds:1024:4:1`	This is a two layer cache. The first (synchronous) layer is composed of 256 8-block buffers in memory, the second (asynchronous) layer is composed of 4 1024-block buffers on the SSD. This scheme works well when accesses proceed through the dataset in random waves roughly 2x1024-blocks wide.

All of the options/configurations supported in CRI's FFIO library are available through this mechanism. We recommend that you look at CRI's I/O optimization guide for information on using FFIO to it's fullest. This mechanism is also compatible with CRI's EIE I/O library.

Tuning the `NETCDF_FFIOSPEC` variable to a program's I/O pattern can dramatically improve performance. Speedups of two orders of magnitude have been seen.

10 NetCDF Utilities

One of the primary reasons for using the netCDF interface for applications that deal with arrays is to take advantage of higher-level netCDF utilities and generic applications for netCDF data. Currently two netCDF utilities are available as part of the netCDF software distribution:

- `ncdump` reads a netCDF dataset and prints a textual representation of the information in the dataset
- `ncgen` reads a textual representation of a netCDF dataset and generates the corresponding binary netCDF file or a C or FORTRAN program to create the netCDF dataset

Users have contributed other netCDF utilities, and various visualization and analysis packages are available that access netCDF data. For an up-to-date list of freely-available and commercial software that can access or manipulate netCDF data, see the NetCDF Software list, `http://www.unidata.ucar.edu/packages/netcdf/software.html`.

This chapter describes the `ncgen` and `ncdump` utilities. These tools convert between binary netCDF datasets and a text representation of netCDF datasets. The output of `ncdump` and the input to `ncgen` is a text description of a netCDF dataset in a tiny language known as CDL (network Common data form Description Language).

10.1 CDL Syntax

Below is an example of CDL, describing a netCDF dataset with several named dimensions (`lat`, `lon`, `time`), variables (`z`, `t`, `p`, `rh`, `lat`, `lon`, `time`), variable attributes (`units`, `_FillValue`, `valid_range`), and some data.

```
netcdf foo {    // example netCDF specification in CDL

dimensions:
lat = 10, lon = 5, time = unlimited;

variables:
  int     lat(lat), lon(lon), time(time);
  float   z(time,lat,lon), t(time,lat,lon);
  double  p(time,lat,lon);
  int     rh(time,lat,lon);

  lat:units = "degrees_north";
  lon:units = "degrees_east";
  time:units = "seconds";
  z:units = "meters";
  z:valid_range = 0., 5000.;
  p:_FillValue = -9999.;
  rh:_FillValue = -1;

data:
  lat   = 0, 10, 20, 30, 40, 50, 60, 70, 80, 90;
```

```
     lon   = -140, -118, -96, -84, -52;
   }
```

All CDL statements are terminated by a semicolon. Spaces, tabs, and newlines can be used freely for readability. Comments may follow the double slash characters `//` on any line.

A CDL description consists of three optional parts: dimensions, variables, and data. The variable part may contain variable declarations and attribute assignments.

A dimension is used to define the shape of one or more of the multidimensional variables described by the CDL description. A dimension has a name and a length. At most one dimension in a CDL description can have the unlimited length, which means a variable using this dimension can grow to any length (like a record number in a file).

A variable represents a multidimensional array of values of the same type. A variable has a name, a data type, and a shape described by its list of dimensions. Each variable may also have associated attributes (see below) as well as data values. The name, data type, and shape of a variable are specified by its declaration in the variable section of a CDL description. A variable may have the same name as a dimension; by convention such a variable contains coordinates of the dimension it names.

An attribute contains information about a variable or about the whole netCDF dataset. Attributes may be used to specify such properties as units, special values, maximum and minimum valid values, and packing parameters. Attribute information is represented by single values or arrays of values. For example, `units` is an attribute represented by a character array such as `celsius`. An attribute has an associated variable, a name, a data type, a length, and a value. In contrast to variables that are intended for data, attributes are intended for ancillary data (data about data).

In CDL, an attribute is designated by a variable and attribute name, separated by a colon (‘`:`’). It is possible to assign global attributes to the netCDF dataset as a whole by omitting the variable name and beginning the attribute name with a colon (‘`:`’). The data type of an attribute in CDL is derived from the type of the value assigned to it. The length of an attribute is the number of data values or the number of characters in the character string assigned to it. Multiple values are assigned to non-character attributes by separating the values with commas (‘`,`’). All values assigned to an attribute must be of the same type.

CDL names for variables, attributes, and dimensions may be any combination of alphabetic or numeric characters as well as underscore (‘_’) , dash (‘`-`’), and dot (‘.’) characters, but names beginning with ‘_’ are reserved for use by the library. Case is significant in CDL names. The names for the primitive data types and their synonyms (`char`, `byte`, `short`, `int`, `long`, `float`, `real`, `double`) are reserved words in CDL, so the names of variables, dimensions, and attributes must not be type names.

The optional data section of a CDL description is where netCDF variables may be initialized. The syntax of an initialization is simple:

variable = value_1, value_2, …;

The comma-delimited list of constants may be separated by spaces, tabs, and newlines. For multi-dimensional arrays, the last dimension varies fastest. Thus, row-order rather than column order is used for matrices. If fewer values are supplied than are needed to fill a variable, it is extended with the fill value. The types of constants need not match the type declared for a variable; coercions are done to convert integers to floating point, for example. All meaningful type conversions are supported.

A special notation for fill values is supported: the _ character designates a fill value for variables.

10.2 CDL Data Types

The CDL data types are:

`char`	Characters.
`byte`	Eight-bit integers.
`short`	16-bit signed integers.
`int`	32-bit signed integers.
`long`	(Deprecated, currently synonymous with int)
`float`	IEEE single-precision floating point (32 bits).
`real`	(Synonymous with float).
`double`	IEEE double-precision floating point (64 bits).

Except for the added data-type `byte` and the lack of the type qualifier `unsigned`, CDL supports the same primitive data types as C. In declarations, type names may be specified in either upper or lower case.

The `byte` type differs from the `char` type in that it is intended for eight-bit data, and the zero byte has no special significance, as it may for character data. The `ncgen` utility converts `byte` declarations to `char` declarations in the output C code and to `BYTE`, `INTEGER*1`, or similar platform-specific declaration in output FORTRAN code.

The `short` type holds values between -32768 and 32767. The `ncgen` utility converts `short` declarations to `short` declarations in the output C code and to `INTEGER*2` declaration in output FORTRAN code.

The `int` type can hold values between -2147483648 and 2147483647. The `ncgen` utility converts `int` declarations to `int` declarations in the output C code and to `INTEGER` declarations in output FORTRAN code. In CDL declarations `integer` and `long` are accepted as synonyms for `int`.

The `float` type can hold values between about -3.4+38 and 3.4+38, with external representation as 32-bit IEEE normalized single-precision floating-point numbers. The `ncgen` utility converts `float` declarations to `float` declarations in the output C code and to `REAL` declarations in output

FORTRAN code. In CDL declarations `real` is accepted as a synonym for `float`.

The `double` type can hold values between about -1.7+308 and 1.7+308, with external representation as 64-bit IEEE standard normalized double-precision, floating-point numbers. The `ncgen` utility converts `double` declarations to `double` declarations in the output C code and to `DOUBLE PRECISION` declarations in output FORTRAN code.

10.3 CDL Notation for Data Constants

This section describes the CDL notation for constants.

Attributes are initialized in the `variables` section of a CDL description by providing a list of constants that determines the attribute's type and length. (In the C and FORTRAN procedural interfaces to the netCDF library, the type and length of an attribute must be explicitly provided when it is defined.) CDL defines a syntax for constant values that permits distinguishing among different netCDF types. The syntax for CDL constants is similar to C syntax, except that type suffixes are appended to `shorts` and `floats` to distinguish them from `ints` and `doubles`.

A byte constant is represented by a single character or multiple character escape sequence enclosed in single quotes. For example:

```
'a'     // ASCII a
'\0'    // a zero byte
'\n'    // ASCII newline character
'\33'   // ASCII escape character (33 octal)
'\x2b'  // ASCII plus (2b hex)
'\376'  // 377 octal = -127 (or 254) decimal
```

Character constants are enclosed in double quotes. A character array may be represented as a string enclosed in double quotes. Multiple strings are concatenated into a single array of characters, permitting long character arrays to appear on multiple lines. To support multiple variable-length string values, a conventional delimiter such as '`,`' may be used, but interpretation of any such convention for a string delimiter must be implemented in software above the netCDF library layer. The usual escape conventions for C strings are honored. For example:

```
"a"            // ASCII ‘a’
"Two\nlines\n" // a 10-character string with two embedded newlines
"a bell:\007"  // a string containing an ASCII bell
"ab","cde"     // the same as "abcde"
```

The form of a `short` constant is an integer constant with an '`s`' or '`S`' appended. If a `short` constant begins with '`0`', it is interpreted as octal. When it begins with '`0x`', it is interpreted as a hexadecimal constant. For example:

```
2s      // a short 2
0123s   // octal
0x7ffs  // hexadecimal
```

The form of an `int` constant is an ordinary integer constant. If an `int` constant begins with '`0`', it is interpreted as octal. When it begins with '0x', it is interpreted as a hexadecimal constant. Examples of valid `int` constants include:

```
-2
0123            // octal
0x7ff           // hexadecimal
1234567890L     // deprecated, uses old long suffix
```

The `float` type is appropriate for representing data with about seven significant digits of precision. The form of a `float` constant is the same as a C floating-point constant with an '`f`' or '`F`' appended. A decimal point is required in a CDL `float` to distinguish it from an integer. For example, the following are all acceptable `float` constants:

```
-2.0f
3.14159265358979f       // will be truncated to less precision
1.f
.1f
```

The `double` type is appropriate for representing floating-point data with about 16 significant digits of precision. The form of a `double` constant is the same as a C floating-point constant. An optional '`d`' or '`D`' may be appended. A decimal point is required in a CDL `double` to distinguish it from an `integer`. For example, the following are all acceptable double constants:

```
-2.0
3.141592653589793
1.0e-20
1.d
```

10.4 ncgen

The `ncgen` tool generates a netCDF file or a C or FORTRAN program that creates a netCDF dataset. If no options are specified in invoking `ncgen`, the program merely checks the syntax of the CDL input, producing error messages for any violations of CDL syntax.

UNIX syntax for invoking `ncgen`:

```
ncgen [-b] [-o netcdf-file] [-c] [-f] [-n] [input-file]
```

where:

`-b` Create a (binary) netCDF file. If the '`-o`' option is absent, a default file name will be constructed from the netCDF name (specified after the `netcdf` keyword in the input) by appending the '`.nc`' extension. **Warning: if a file already exists with the specified name it will be overwritten.**

`-o netcdf-file`	Name for the netCDF file created. If this option is specified, it implies the '`-b`' option. (This option is necessary because netCDF files are direct-access files created with seek calls, and hence cannot be written to standard output.)
`-c`	Generate C source code that will create a netCDF dataset matching the netCDF specification. The C source code is written to standard output. This is only useful for relatively small CDL files, since all the data is included in variable initializations in the generated program.
`-f`	Generate FORTRAN source code that will create a netCDF dataset matching the netCDF specification. The FORTRAN source code is written to standard output. This is only useful for relatively small CDL files, since all the data is included in variable initializations in the generated program.
`-n`	Deprecated. Like the '`-b`' option, except creates a netCDF file with a '`.cdf`' extension instead of an '`.nc`' extension, in the absence of an output filename specified by the '`-o`' option. This option is only supported for backward compatibility.

Examples

Check the syntax of the CDL file `foo.cdl`:

```
ncgen foo.cdl
```

From the CDL file `foo.cdl`, generate an equivalent binary netCDF file named `bar.nc`:

```
ncgen -o bar.nc foo.cdl
```

From the CDL file `foo.cdl`, generate a C program containing netCDF function invocations that will create an equivalent binary netCDF dataset:

```
ncgen -c foo.cdl > foo.c
```

10.5 ncdump

The `ncdump` tool generates the CDL text representation of a netCDF dataset on standard output, optionally excluding some or all of the variable data in the output. The output from `ncdump` is intended to be acceptable as input to `ncgen`. Thus `ncdump` and `ncgen` can be used as inverses to transform data representation between binary and text representations.

`ncdump` may also be used as a simple browser for netCDF datasets, to display the dimension names and lengths; variable names, types, and shapes; attribute names and values; and optionally, the values of data for all variables or selected variables in a netCDF dataset.

`ncdump` defines a default format used for each type of netCDF variable data, but this can be overridden if a `C_format` attribute is defined for a netCDF variable. In this case, `ncdump` will use the `C_format` attribute to format values for that variable. For example, if floating-point data for the netCDF variable `Z` is known to be accurate to only three significant digits, it might be appropriate to use this variable attribute:

```
Z:C_format = "%.3g"
```

`ncdump` uses ‘`_`’ to represent data values that are equal to the `_FillValue` attribute for a variable, intended to represent data that has not yet been written. If a variable has no `_FillValue` attribute, the default fill value for the variable type is used unless the variable is of byte type.

UNIX syntax for invoking `ncdump`:

```
ncdump  [ -c | -h]  [-v var1,…]  [-b lang]  [-f lang]
[-l len]  [ -p fdig[,ddig]]  [ -n name]  [input-file]
```

where:

`-c`	Show the values of *coordinate* variables (variables that are also dimensions) as well as the declarations of all dimensions, variables, and attribute values. Data values of non-coordinate variables are not included in the output. This is often the most suitable option to use for a brief look at the structure and contents of a netCDF dataset.
`-h`	Show only the *header* information in the output, that is, output only the declarations for the netCDF dimensions, variables, and attributes of the input file, but no data values for any variables. The output is identical to using the ‘`-c`’ option except that the values of coordinate variables are not included. (At most one of ‘`-c`’ or ‘`-h`’ options may be present.)
`-v var1,…`	The output will include data values for the specified variables, in addition to the declarations of all dimensions, variables, and attributes. One or more variables must be specified by name in the comma-delimited list following this option. The list must be a single argument to the command, hence cannot contain blanks or other white space characters. The named variables must be valid netCDF variables in the input-file. The default, without this option and in the absence of the ‘`-c`’ or ‘`-h`’ options, is to include data values for *all* variables in the output.
`-b lang`	A brief annotation in the form of a CDL comment (text beginning with the characters ‘`//`’) will be included in the data section of the output for each ‘row’ of data, to help identify data values for multidimensional variables. If *lang* begins with ‘`C`’ or ‘`c`’, then C language conventions will be used (zero-based indices, last dimension varying fastest). If *lang* begins with ‘`F`’ or ‘`f`’, then FORTRAN language conventions will be used (one-based indices, first dimension varying fastest). In either case, the data will be presented in the same order; only the annotations will differ. This option may be useful for browsing through large volumes of multidimensional data.

`-f lang` Full annotations in the form of trailing CDL comments (text beginning with the characters '`//`') for every data value (except individual characters in character arrays) will be included in the data section. If *lang* begins with '`C`' or '`c`', then C language conventions will be used (zero-based indices, last dimension varying fastest). If *lang* begins with '`F`' or '`f`', then FORTRAN language conventions will be used (one-based indices, first dimension varying fastest). In either case, the data will be presented in the same order; only the annotations will differ. This option may be useful for piping data into other filters, since each data value appears on a separate line, fully identified. (At most one of '`-b`' or '`-f`' options may be present.)

`-l len` Changes the default maximum line length (80) used in formatting lists of non-character data values.

`-p float_digits[,double_digits]`

Specifies default precision (number of significant digits) to use in displaying floating-point or double precision data values for attributes and variables. If specified, this value overrides the value of the `C_format` attribute, if any, for a variable. Floating-point data will be displayed with *float_digits* significant digits. If *double_digits* is also specified, double-precision values will be displayed with that many significant digits. In the absence of any '`-p`' specifications, floating-point and double-precision data are displayed with 7 and 15 significant digits respectively. CDL files can be made smaller if less precision is required. If both floating-point and double precisions are specified, the two values must appear separated by a comma (no blanks) as a single argument to the command.

`-n name` CDL requires a name for a netCDF dataset, for use by '`ncgen -b`' in generating a default netCDF dataset name. By default, `ncdump` constructs this name from the last component of the file name of the input netCDF dataset by stripping off any extension it has. Use the '`-n`' option to specify a different name. Although the output file name used by '`ncgen -b`' can be specified, it may be wise to have `ncdump` change the default name to avoid inadvertently overwriting a valuable netCDF dataset when using `ncdump`, editing the resulting CDL file, and using '`ncgen -b`' to generate a new netCDF dataset from the edited CDL file.

Examples

Look at the structure of the data in the netCDF dataset `foo.nc`:

```
ncdump -c foo.nc
```

Produce an annotated CDL version of the structure and data in the netCDF dataset `foo.nc`, using C-style indexing for the annotations:

```
ncdump -b c foo.nc > foo.cdl
```

Output data for only the variables `uwind` and `vwind` from the netCDF dataset `foo.nc`, and show the floating-point data with only three significant digits of precision:

```
ncdump -v uwind,vwind -p 3 foo.nc
```

Produce a fully-annotated (one data value per line) listing of the data for the variable `omega`, using FORTRAN conventions for indices, and changing the netCDF dataset name in the resulting CDL file to `omega`:

```
ncdump -v omega -f fortran -n omega foo.nc > Z.cdl
```

11 Answers to Some Frequently Asked Questions

This chapter contains answers to some of the most frequently asked questions about netCDF. A more comprehensive and up-to-date FAQ document for netCDF is maintained at `http://www.unidata.ucar.edu/packages/netcdf/faq.html`.

What Is netCDF?

NetCDF (network Common Data Form) is an interface for array-oriented data access and a freely-distributed collection of software libraries for C, FORTRAN, C++, and Perl that provide implementations of the interface. The netCDF software was developed by Glenn Davis, Russ Rew, and Steve Emmerson at the Unidata Program Center in Boulder, Colorado, and augmented by contributions from other netCDF users. The netCDF libraries define a machine-independent format for representing arrays. Together, the interface, libraries, and format support the creation, access, and sharing of array-oriented data.

NetCDF data is:

- Self-describing. A netCDF dataset includes information about the data it contains.
- portable. A netCDF dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- Direct-access. A small subset of a large dataset may be accessed efficiently, without first reading through all the preceding data.
- Appendable. Data can be appended to a netCDF dataset along one dimension for multiple variables without copying the dataset or redefining its structure. The structure of a netCDF dataset may also be changed, though in some cases this is implemented by copying the data.
- Sharable. One writer and multiple readers may simultaneously access the same netCDF dataset.

How do I get the netCDF software package?

Source distributions are available via anonymous FTP from the directory

`ftp://ftp.unidata.ucar.edu/pub/netcdf/`.

Files in that directory include:

`netcdf.tar.Z`	A compressed tar file of source code for the latest general release.
`netcdf-beta.tar.Z`	The current beta-test release.

Binary distributions for some platforms are available from the directory `ftp://ftp.unidata.ucar.edu/pub/binary/`

Source for the Perl interface is available as a separate package, via anonymous FTP from the directory

`ftp://ftp.unidata.ucar.edu/pub/netcdf-perl/`.

Is there any access to netCDF information on the World Wide Web?

Yes, the latest version of this FAQ document as well as a hypertext version of the NetCDF User's Guide and other information about netCDF are available from

`http://www.unidata.ucar.edu/packages/netcdf`.

What has changed since the previous release?

Version 3 keeps the same format, but introduces new interfaces for C and FORTRAN that provide automatic type conversion and improved type safety. For more details, see:

`http://www.unidata.ucar.edu/packages/netcdf/release-notes.html`.

Is there a mailing list for netCDF discussions and questions?

Yes. For information about the mailing list and how to subscribe or unsubscribe, send a message to `majordomo@unidata.ucar.edu` with no subject and with the following line in the body of the message:

```
   info netcdfgroup
```

Who else uses netCDF?

The netCDF mailing list has almost 500 addresses (some of which are aliases to more addresses) in fifteen countries. Several groups have adopted netCDF as a standard way to represent some forms of array-oriented data, including groups in the atmospheric sciences, hydrology, oceanography, environmental modeling, geophysics, chromatography, mass spectrometry, and neuro-imaging.

A description of some of the projects and groups that have used netCDF is available from

`http://www.unidata.ucar.edu/packages/netcdf/usage.html`.

What is the physical format for a netCDF files?

See Chapter 9 "NetCDF File Structure and Performance," page 83, for an explanation of the physical structure of netCDF data at a high enough level to make clear the performance implications of different data organizations. See Appendix B "File Format Specification," page 107, for a detailed specification of the file format.

Programs that access netCDF data should perform all access through the documented interfaces, rather than relying on the physical format of netCDF data. That way, any future changes to the format will not require changes to programs, since any such changes will be accompanied by changes in the library to support both the old and new versions of the format.

What does netCDF run on?

The current version of netCDF has been tested successfully on the following platforms:

- AIX-4.1
- HPUX-9.05
- IRIX-5.3
- IRIX64-6.1
- MSDOS (using gcc, f2c, and GNU make)
- OSF1-3.2
- OpenVMS-6.2
- OS/2 2.1
- SUNOS-4.1.4
- SUNOS-5.5
- ULTRIX-4.5
- UNICOS-8
- Windows NT-3.51

What other software is available for netCDF data?

Utilities available in the current netCDF distribution from Unidata are `ncdump`, for converting netCDF datasets to an ASCII human-readable form, and `ncgen` for converting from the ASCII human-readable form back to a binary netCDF file or a C or FORTRAN program for generating the netCDF dataset.

Several commercial and freely available analysis and data visualization packages have been adapted to access netCDF data. More information about these packages and other software that can be used to manipulate or display netCDF data is available from

`http://www.unidata.ucar.edu/packages/netcdf/software.html`.

What other formats are available for scientific data?

The *Scientific Data Format Information FAQ*, available from `http://fits.cv.nrao.edu/traffic/scidataformats/faq.html`, provides a good description of other access interfaces and formats for array-oriented data, including CDF and HDF.

How do I make a bug report?

If you find a bug, send a description to `support@unidata.ucar.edu`. This is also the address to use for questions or discussions about netCDF that are not appropriate for the entire `netcdfgroup` mailing list.

How do I search through past problem reports?

A search form is available at the bottom of the netCDF home page providing a full-text search of the support questions and answers about netCDF provided by Unidata support staff.

How does the C++ interface differ from the C interface?

It provides all the functionality of the C interface (except for the mapped array access of `nc_put_varm_`*type* and `nc_get_varm_`*type*). With the C++ interface (`http://www.unidata.ucar.edu/packages/netcdf/cxxdoc_toc.html`) no IDs are needed for netCDF components, there is no need to specify types when creating attributes, and less indirection is required for dealing with dimensions. However, the C++ interface is less mature and less-widely used than the C interface, and the documentation for the C++ interface is less extensive, assuming a familiarity with the netCDF data model and the C interface.

How does the FORTRAN interface differ from the C interface?

It provides all the functionality of the C interface. The FORTRAN interface uses FORTRAN conventions for array indices, subscript order, and strings. There is no difference in the on-disk format for data written from the different language interfaces. Data written by a C language program may be read from a FORTRAN program and vice-versa.

How does the Fortran 90 interface differ from the C interface?

The Fortran 90 interface provides the same functionality as the FORTRAN and C interfaces, but the interface is substantially smaller. We’ve done this by using optional arguments in the file, dimension, variable, and attribute inquire functions (`nf90_Inquire_`) and by using overloaded functions for the reading and writing of variables and attributes.

The Fortran 90 interface is currently implemented as a set of wrappers around the FORTRAN interface. Because there is almost no copying of information, the performance penalty should be very small.

The Fortran 90 interface is new as of February 2000, and we would appreciate any user feedback.

How does the Perl interface differ from the C interface?

It provides all the functionality of the C interface. The Perl interface (`http://www.uni-`

`data.ucar.edu/packages/netcdf-perl/`) uses Perl conventions for arrays and strings. There is no difference in the on-disk format for data written from the different language interfaces. Data written by a C language program may be read from a Perl program and vice-versa.

Appendix A Units

The Unidata Program Center has developed a units library to convert between formatted and binary forms of units specifications and perform unit algebra on the binary form. Though the units library is self-contained and there is no dependency between it and the netCDF library, it is nevertheless useful in writing generic netCDF programs and we suggest you obtain it. The library and associated documentation is available from `http://www.unidata.ucar.edu/packages/udunits/`.

The following are examples of units strings that can be interpreted by the `utScan()` function of the Unidata units library:

```
10 kilogram.meters/seconds2
10 kg-m/sec2
10 kg m/s^2
10 kilogram meter second-2
(PI radian)2
degF
100rpm
geopotential meters
33 feet water
milliseconds since 1992-12-31 12:34:0.1 -7:00
```

A unit is specified as an arbitrary product of constants and unit-names raised to arbitrary integral powers. Division is indicated by a slash '`/`'. Multiplication is indicated by white space, a period '`.`', or a hyphen '`-`'. Exponentiation is indicated by an integer suffix or by the exponentiation operators '`^`' and '`**`'. Parentheses may be used for grouping and disambiguation. The time stamp in the last example is handled as a special case.

Arbitrary Galilean transformations (i.e., $y = ax + b$) are allowed. In particular, temperature conversions are correctly handled. The specification:

```
degF @ 32
```

indicates a Fahrenheit scale with the origin shifted to thirty-two degrees Fahrenheit (i.e., to zero Celsius). Thus, the Celsius scale is equivalent to the following unit:

```
1.8 degF @ 32
```

Note that the origin-shift operation takes precedence over multiplication. In order of increasing precedence, the operations are division, multiplication, origin-shift, and exponentiation.

`utScan()` understands all the SI prefixes (e.g. “mega” and “milli”) plus their abbreviations (e.g. “M” and “m”)

The function `utPrint()` always encodes a unit specification one way. To reduce misunderstandings, it is recommended that this encoding style be used as the default. In general, a unit is encoded in terms of basic units, factors, and exponents. Basic units are separated by spaces, and any exponent directly appends its associated unit. The above examples would be encoded as follows:

```
10 kilogram meter second-2
9.8696044 radian2
0.555556 kelvin @ 255.372
10.471976 radian second-1
9.80665 meter2 second-2
98636.5 kilogram meter-1 second-2
0.001 seconds since 1992-12-31 19:34:0.1000 UTC
```

(Note that the Fahrenheit unit is encoded as a deviation, in fractional kelvins, from an origin at 255.372 kelvin, and that the time in the last example has been referenced to UTC.)

The database for the units library is a formatted file containing unit definitions and is used to initialize this package. It is the first place to look to discover the set of valid names and symbols.

The format for the units-file is documented internally and the file may be modified by the user as necessary. In particular, additional units and constants may be easily added (including variant spellings of existing units or constants).

`utScan()` is case-sensitive. If this causes difficulties, you might try making appropriate additional entries to the units-file.

Some unit abbreviations in the default units-file might seem counterintuitive. In particular, note the following:

For	Use	Not	Which Instead Means
Celsius	Celsius	C	coulomb
gram	gram	g	<standard free fall>
gallon	gallon	gal	<acceleration>
radian	radian	rad	<absorbed dose>
Newton	newton or N	nt	nit (unit of photometry)

For additional information on this units library, please consult the manual pages that come with the distribution.

Appendix B File Format Specification

This appendix specifies the netCDF file format.

The format is first presented formally, using a BNF grammar notation. In the grammar, optional components are enclosed between braces (‘[‘ and ‘]’). Comments follow ‘//’ characters. Nonterminals are in lower case, and terminals are in upper case. A sequence of zero or more occurrences of an entity are denoted by ‘[entity …]’.

The Format in Detail

```
netcdf_file := header  data

header  := magic  numrecs  dim_array  gatt_array  var_array

magic   := 'C'  'D'  'F'  VERSION_BYTE

VERSION_BYTE := '\001'    // the file format version number

numrecs    := NON_NEG

dim_array  :=  ABSENT | NC_DIMENSION  nelems  [dim …]

gatt_array :=  att_array  // global attributes

att_array  :=  ABSENT | NC_ATTRIBUTE  nelems  [attr …]

var_array  :=  ABSENT | NC_VARIABLE   nelems  [var …]

ABSENT  := ZERO  ZERO     // Means array not present (equivalent to
                          // nelems == 0).

nelems  := NON_NEG        // number of elements in following sequence

dim     := name  dim_length

name    := string

dim_length := NON_NEG     // If zero, this is the record dimension.
                          // There can be at most one record dimension.

attr    := name  nc_type  nelems  [values]

nc_type := NC_BYTE | NC_CHAR | NC_SHORT | NC_INT | NC_FLOAT | NC_DOUBLE

var     := name  nelems  [dimid …]  vatt_array nc_type  vsize  begin
                          // nelems is the rank (dimensionality) of the
                          // variable; 0 for scalar, 1 for vector, 2 for
                          // matrix, …
```

```
vatt_array :=  att_array  // variable-specific attributes

dimid   := NON_NEG        // Dimension ID (index into dim_array) for
                          // variable shape.  We say this is a “record
                          // variable” if and only if the first
                          // dimension is the record dimension.

vsize    := NON_NEG       // Variable size.  If not a record variable,
                          // the amount of space, in bytes, allocated to
                          // that variable’s data. This number is the
                          // product of the dimension lengths times the
                          // size of the type, padded to a four byte
                          // boundary.  If a record variable, it is the
                          // amount of space per record. The netCDF
                          // “record size” is calculated as the sum of
                          // the vsize’s of the record variables.

begin   := NON_NEG        // Variable start location. The offset in
                          // bytes (seek index) in the file of the
                          // beginning of data for this variable.

data    := non_recs  recs

non_recs := [values …]    // Data for first non-record var, second
                          // non-record var, …

recs    := [rec …]        // First record, second record, …

rec     := [values …]     // Data for first record variable for record
                          // n, second record variable for record n, …
                          // See the note below for a special case.

values  := [bytes] | [chars] | [shorts] | [ints] | [floats] | [doubles]

string  := nelems  [chars]

bytes   := [BYTE …]  padding

chars   := [CHAR …]  padding

shorts  := [SHORT …]  padding

ints    := [INT …]

floats  := [FLOAT …]

doubles := [DOUBLE …]

padding := <0, 1, 2, or 3 bytes to next 4-byte boundary>
                          // In header, padding is with 0 bytes.  In
                          // data, padding is with variable’s fill-value.

NON_NEG := <INT with non-negative value>
```

```
ZERO    := <INT with zero value>

BYTE    := <8-bit byte>

CHAR    := <8-bit ACSII/ISO encoded character>

SHORT   := <16-bit signed integer, Bigendian, two’s complement>

INT     := <32-bit signed integer, Bigendian, two’s complement>

FLOAT   := <32-bit IEEE single-precision float, Bigendian>

DOUBLE  := <64-bit IEEE double-precision float, Bigendian>

// tags are 32-bit INTs
NC_BYTE      := 1         // data is array of 8 bit signed integer
NC_CHAR      := 2         // data is array of characters, i.e., text
NC_SHORT     := 3         // data is array of 16 bit signed integer
NC_INT       := 4         // data is array of 32 bit signed integer
NC_FLOAT     := 5         // data is array of IEEE single precision float
NC_DOUBLE    := 6         // data is array of IEEE double precision float
NC_DIMENSION := 10
NC_VARIABLE  := 11
NC_ATTRIBUTE := 12
```

Computing File Offsets

To calculate the offset (position within the file) of a specified data value, let *external_sizeof* be the external size in bytes of one data value of the appropriate type for the specified variable, *nc_type*:

```
NC_BYTE         1
NC_CHAR         1
NC_SHORT        2
NC_INT          4
NC_FLOAT        4
NC_DOUBLE       8
```

On a call to `NF90_OPEN` (or `NF90_ENDDEF`), scan through the array of variables, denoted *var_array* above, and sum the *vsize* fields of “record” variables to compute *recsize*.

Form the products of the dimension lengths for the variable from right to left, skipping the leftmost (record) dimension for record variables, and storing the results in a *product* array for each variable. For example:

```
Non-record variable:

        dimension lengths:      [  5  3  2 7]
        product:                [210 42 14 7]

Record variable:
        dimension lengths:      [0  2  9 4]
        product:                [0 72 36 4]
```

At this point, the leftmost product, when rounded up to the next multiple of 4, is the variable size, *vsize*, in the grammar above. For example, in the non-record variable above, the value of the *vsize* field is 212 (210 rounded up to a multiple of 4). For the record variable, the value of *vsize* is just 72, since this is already a multiple of 4.

Let *coord* be an array of the coordinates of the desired data value, and *offset* be the desired result. Then *offset* is just the file offset of the first data value of the desired variable (its *begin* field) added to the inner product of the *coord* and *product* vectors times the size, in bytes, of each datum for the variable. Finally, if the variable is a record variable, the product of the record number, ‘`coord[0]`’, and the record size, *recsize* is added to yield the final *offset* value.

In pseudo-C code, here’s the calculation of *offset*:

```
for (innerProduct = i = 0; i < var.rank; i++)
        innerProduct += product[i] * coord[i]
offset = var.begin;
offset += external_sizeof * innerProduct
if(IS_RECVAR(var))
        offset += coord[0] * recsize;
```

So, to get the data value (in external representation):

```
lseek(fd, offset, SEEK_SET);
read(fd, buf, external_sizeof);
```

A special case: Where there is exactly one record variable, we drop the restriction that each record be four-byte aligned, so in this case there is no record padding.

Examples

By using the grammar above, we can derive the smallest valid netCDF file, having no dimensions, no variables, no attributes, and hence, no data. A CDL representation of the empty netCDF file is

```
netcdf empty { }
```

This empty netCDF file has 32 bytes, as you may verify by using ‘`ncgen -b empty.cdl`’ to generate it from the CDL representation. It begins with the four-byte “magic number” that identifies it as a netCDF version 1 file: ‘C’, ‘D’, ‘F’, ‘\001’. Following are seven 32-bit integer zeros representing the number of records, an empty array of dimensions, an empty array of global attributes, and an empty array of variables.

Below is an (edited) dump of the file produced on a big-endian machine using the Unix command

```
od -xcs empty.nc
```

Each 16-byte portion of the file is displayed with 4 lines. The first line displays the bytes in hexadecimal. The second line displays the bytes as characters. The third line displays each group of two bytes interpreted as a signed 16-bit integer. The fourth line (added by human) presents the interpretation of the bytes in terms of netCDF components and values.

```
     4344    4601    0000    0000    0000    0000    0000    0000
    C   D   F 001  \0  \0  \0  \0  \0  \0  \0  \0 \0  \0  \0  \0
   17220   17921   00000   00000   00000   00000   00000   00000
  [magic number ] [  0 records  ] [  0 dimensions   (ABSENT)    ]

     0000    0000    0000    0000    0000    0000    0000    0000
   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 \0  \0  \0  \0  \0
   00000   00000   00000   00000   00000   00000   00000   00000
  [  0 global atts  (ABSENT)    ] [  0 variables    (ABSENT)    ]
```

As a slightly less trivial example, consider the CDL

```
netcdf tiny {
dimensions:
        dim = 5;
variables:
        short vx(dim);
data:
        vx = 3, 1, 4, 1, 5 ;
}
```

which corresponds to a 92-byte netCDF file. The following is an edited dump of this file:

```
     4344    4601    0000    0000    0000    000a    0000    0001
    C   D   F 001  \0  \0  \0  \0  \0  \0  \0  \n \0  \0  \0 001
   17220   17921   00000   00000   00000   00010   00000   00001
  [magic number ] [  0 records  ] [NC_DIMENSION ] [ 1 dimension ]

     0000    0003    6469    6d00    0000    0005    0000    0000
   \0  \0  \0 003   d   i   m  \0  \0  \0  \0 005 \0  \0  \0  \0
   00000   00003   25705   27904   00000   00005   00000   00000
  [  3 char name = "dim"        ] [ size = 5    ] [ 0 global atts

     0000    0000    0000    000b    0000    0001    0000    0002
   \0  \0  \0  \0  \0  \0  \0 013  \0  \0  \0 001 \0  \0  \0 002
   00000   00000   00000   00011   00000   00001   00000   00002
   (ABSENT)     ] [NC_VARIABLE  ] [ 1 variable  ] [ 2 char name =

     7678    0000    0000    0001    0000    0000    0000    0000
    v   x  \0  \0  \0  \0  \0 001  \0  \0  \0  \0 \0  \0  \0  \0
   30328   00000   00000   00001   00000   00000   00000   00000
   "vx"         ] [1 dimension  ] [ with ID 0   ] [ 0 attributes

     0000    0000    0000    0003    0000    000c    0000    0050
   \0  \0  \0  \0  \0  \0  \0 003  \0  \0  \0 \f  \0  \0  \0   P
   00000   00000   00000   00003   00000   00012   00000   00080
   (ABSENT)     ] [type NC_SHORT] [size 12 bytes] [offset:    80]

     0003    0001    0004    0001    0005    8001
   \0 003  \0 001  \0 004  \0 001  \0 005 200 001
    00003   00001   00004   00001   00005  -32767
  [    3] [    1] [    4] [    1] [    5] [fill ]
```

Appendix C Summary of Fortran 90 Interface

```
Dataset Functions

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

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

   function nf90_create(path, cmode, ncid)
     character (len = *), intent(in   ) :: path
     integer,             intent(in   ) :: cmode
     integer, optional,   intent(in   ) :: initialsize
     integer, optional,   intent(inout) :: chunksize
     integer,             intent(  out) :: ncid
     integer                            :: nf90_create

   function nf90_open(path, mode, ncid, chunksize)
     character (len = *), intent(in   ) :: path
     integer,             intent(in   ) :: mode
     integer,             intent(  out) :: ncid
     integer, optional,   intent(inout) :: chunksize
     integer                            :: nf90_open

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

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

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

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

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

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

```
    integer              :: nf90_close

  function nf90_Inquire(ncid, nDimensions, nVariables, nAttributes, &
                        unlimitedDimId)
    integer,           intent( in) :: ncid
    integer, optional, intent(out) :: nDimensions, nVariables, nAttributes,&
                                      unlimitedDimId
    integer                        :: nf90_Inquire
```

Dimension functions

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

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

  function nf90_Inquire_Dimension(ncid, dimid, name, len)
    integer,                       intent( in) :: ncid, dimid
    character (len = *), optional, intent(out) :: name
    integer,             optional, intent(out) :: len
    integer                                    :: nf90_Inquire_Dimension

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

Variable functions

```
  function nf90_def_var(ncid, name, xtype, dimids, varid)
    integer,               intent( in) :: ncid
    character (len = *),   intent( in) :: name
    integer,               intent( in) :: xtype
    integer, dimension(:), intent( in) :: dimids ! May be omitted, scalar,
                                                 ! vector
    integer                            :: nf90_def_var

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

  function nf90_Inquire_Variable(ncid, varid, name, xtype, ndims, &
                                 dimids, nAtts)
    integer,                          intent( in) :: ncid, varid
    character (len = *),   optional, intent(out) :: name
    integer,               optional, intent(out) :: xtype, ndims
    integer, dimension(*), optional, intent(out) :: dimids
    integer,               optional, intent(out) :: nAtts
    integer                                      :: nf90_Inquire_Variable

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

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

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

Attribute functions

  function nf90_Inquire_Attribute(ncid, varid, name, xtype, len, attnum)
    integer,             intent( in)           :: ncid, varid
    character (len = *), intent( in)           :: name
    integer,             intent(out), optional :: xtype, len, attnum
    integer                                    :: nf90_Inquire_Attribute

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

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

  function nf90_get_att(ncid, varid, name, values)

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

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

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

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

Appendix D FORTRAN 77 to Fortran 90 Transition Guide

The new Fortran 90 interface

The Fortran 90 interface to the netCDF library closely follows the `FORTRAN 77` interface. In most cases, function and constant names and argument lists are the same, except that `nf90_` replaces `nf_` in names. The Fortran 90 interface is much smaller than the `FORTRAN 77` interface, however. This has been accomplished by using optional arguments and overloaded functions wherever possible.

Because `FORTRAN 77` is a subset of Fortran 90, there is no reason to modify working `FORTRAN` code to use the Fortran 90 interface. New code, however, can easily be patterned after existing `FORTRAN` while taking advantage of the simpler interface. Some compilers may provide additional support when using Fortran 90. For example, compilers may issue warnings if arguments with `intent( in)` are not set before they are passed to a procedure.

The Fortran 90 interface is currently implemented as a set of wrappers around the base `FORTRAN` subroutines in the netCDF distribution. Future versions may be implemented entirely in Fortran 90, adding additional error checking possibilities.

Changes to Inquiry functions

In the Fortran 90 interface there are two inquiry functions each for dimensions, variables, and attributes, and a single inquiry function for datasets. These functions take optional arguments, allowing users to request only the information they need. These functions replace the many-argument and single-argument inquiry functions in the `FORTRAN` interface.

As an example, compare the attribute inquiry functions in the Fortran 90 interface

```
function nf90_Inquire_Attribute(ncid, varid, name, xtype, len, attnum)
  integer,             intent( in)           :: ncid, varid
  character (len = *), intent( in)           :: name
  integer,             intent(out), optional :: xtype, len, attnum
  integer                                    :: nf90_Inquire_Attribute

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

with those in the FORTRAN interface

```
INTEGER FUNCTION  NF_INQ_ATT        (NCID, VARID, NAME, xtype, len)
INTEGER FUNCTION  NF_INQ_ATTID      (NCID, VARID, NAME, attnum)
INTEGER FUNCTION  NF_INQ_ATTTYPE    (NCID, VARID, NAME, xtype)
```

```
   INTEGER FUNCTION  NF_INQ_ATTLEN     (NCID, VARID, NAME, len)
   INTEGER FUNCTION  NF_INQ_ATTNAME    (NCID, VARID, ATTNUM, name)
```

Changes to put and get function

The biggest simplification in the Fortran 90 is in the `nf90_put_var` and `nf90_get_var` functions. Both functions are overloaded: the `values` argument can be a scalar or an array any rank (7 is the maximum rank allowed by Fortran 90), and may be of any numeric type or the default character type. The netCDF library provides transparent conversion between the external representation of the data and the desired internal representation.

The `start`, `count`, `stride`, and `map` arguments to `nf90_put_var` and `nf90_get_var` are optional. By default, data is read from or written to consecutive values of starting at the origin of the netCDF variable; the shape of the argument determines how many values are read from or written to each dimension. Any or all of these arguments may be supplied to override the default behavior.

Note also that Fortran 90 allows arbitrary array sections to be passed to any procedure, which may greatly simplify programming. See Section 7.5 “Writing Data Values: NF90_PUT_VAR,” page 56, and Section 7.6 “Reading Data Values: NF90_GET_VAR,” page 61, for examples.

Index for Fortran 90