The NetCDF Users Guide

Data Model, Programming Interfaces, and Format for Self-Describing, Portable Data
NetCDF Version 4.1.2-beta1
July 2010

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

Table of Contents

Foreword

Unidata (`http://www.unidata.ucar.edu`) 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, C++, Perl, or other language 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 access has been implemented in about half of Unidata's software, so far, and it is planned that such commonality will extend across all Unidata applications in order to:

- Facilitate the use of common datasets by distinct applications.
- Permit datasets to be transported between or shared by dissimilar computers transparently, i.e., 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 which simplifies the inclusion of new software into the Unidata system.

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 remote access protocols.

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 public domain and commercial data analysis systems can now accept netCDF datasets as input.

Several organizations have adopted netCDF as a data access standard, and there is an effort underway at the National Center for Supercomputer Applications (NCSA, which is associated with the University of Illinois at Urbana-Champaign) to support the netCDF programming interfaces as a means to store and retrieve data in "HDF files," i.e., in the format used by the popular NCSA tools. 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 making improvements 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, 1996

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, Fortran 90, and C++ interfaces for accessing netCDF data. These libraries are available for many common computing platforms.

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. It explains how the netCDF data model uses dimensions, variables, and attributes to store data. Language specific programming guides are available for C (see Section "Top" in *The NetCDF C Interface Guide*), C++ (see Section "Top" in *The NetCDF C++ Interface Guide*), Fortran 77 (see Section "Top" in *The NetCDF Fortran 77 Interface Guide*), and Fortran 90 (see Section "Top" in *The NetCDF Fortran 90 Interface Guide*).

Reference documentation for UNIX systems, in the form of UNIX 'man' pages for the C and FORTRAN interfaces is also available at the netCDF web site (`http://www.unidata.ucar.edu/netcdf`), and with the netCDF distribution.

The latest version of this document, and the language specific guides, can be found at the netCDF web site, `http://www.unidata.ucar.edu/netcdf/docs`, along with extensive additional information about netCDF, including pointers to other software that works with netCDF data.

Separate documentation of the Java netCDF library can be found at `http://www.unidata.ucar.edu/software/netcdf-java`.

For installation and porting information See Section "Top" in *The NetCDF Installation and Porting Guide*.

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 items which all have the same data 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 has led to improved accessibility of data and improved re-usability of software for array-oriented data management, analysis, and display.

The netCDF software implements an abstract data type, 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, (see Section “Top” in *The NetCDF C Interface Guide*), FORTRAN 77 (see Section “Top” in *The NetCDF Fortran 77 Interface Guide*), FORTRAN 90 (see Section “Top” in *The NetCDF Fortran 90 Interface Guide*), and C++ (see Section “Top” in *The NetCDF C++ Interface Guide*).

The netCDF library is supported for various UNIX operating systems. A MS Windows port is also available. 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 via FTP to encourage its widespread use. (`ftp://ftp.unidata.ucar.edu/pub/netcdf`).

For detailed installation instructions, see the Porting and Installation Guide. See Section “Top” in *The NetCDF Installation and Porting Guide*.

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 is not suitable 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 The netCDF File Format

Until version 3.6.0, all versions of netCDF employed only one binary data format, now referred to as netCDF classic format. NetCDF classic is the default format for all versions of netCDF.

In version 3.6.0 a new binary format was introduced, 64-bit offset format. Nearly identical to netCDF classic format, it uses 64-bit offsets (hence the name), and allows users to create far larger datasets.

In version 4.0.0 a third binary format was introduced: the HDF5 format. Starting with this version, the netCDF library can use HDF5 files as its base format. (Only HDF5 files created with netCDF-4 can be understood by netCDF-4).

By default, netCDF uses the classic format. To use the 64-bit offset or netCDF-4/HDF5 format, set the appropriate constant when creating the file.

To achieve network-transparency (machine-independence), netCDF classic and 64-bit offset formats are implemented in terms of an external representation much like XDR (eXternal Data Representation, see `http://www.ietf.org/rfc/rfc1832.txt`), 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.

Descriptions of the overall structure of netCDF classic and 64-bit offset files are provided later in this manual. See Chapter 4 [Structure], page 35.

The details of the classic and 64-bit offset formats are described in an appendix. See Appendix C [File Format], page 75. 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 How to Select the Format

With three different base formats, care must be taken in creating data files to choose the correct base format.

The format of a netCDF file is determined at create time.

When opening an existing netCDF file the netCDF library will transparently detect its format and adjust accordingly. However, netCDF library versions earlier than 3.6.0

cannot read 64-bit offset format files, and library versions before 4.0 can't read netCDF-4/HDF5 files. NetCDF classic format files (even if created by version 3.6.0 or later) remain compatible with older versions of the netCDF library.

Users are encouraged to use netCDF classic format to distribute data, for maximum portability.

To select 64-bit offset or netCDF-4 format files, C programmers should use flag NC_64BIT_OFFSET or NC_NETCDF4 in function nc_create. See Section "nc_create" in *The NetCDF C Interface Guide*.

In Fortran, use flag nf_64bit_offset or nf_format_netcdf4 in function NF_CREATE. See Section "NF_CREATE" in *The NetCDF Fortran 77 Interface Guide*.

It is also possible to change the default creation format, to convert a large body of code without changing every create call. C programmers see Section "nc_set_default_format" in *The NetCDF C Interface Guide*. Fortran programs see Section "NF_SET_DEFAULT_FORMAT" in *The NetCDF Fortran 77 Interface Guide*.

1.4.1 NetCDF Classic Format

The original netCDF format is identified using four bytes in the file header. All files in this format have "CDF\001" at the beginning of the file. In this documentation this format is referred to as "netCDF classic format."

NetCDF classic format is identical to the format used by every previous version of netCDF. It has maximum portability, and is still the default netCDF format.

For some users, the various 2 GiB format limitations of the classic format become a problem. (see Section 4.6 [Classic Limitations], page 38).

1.4.2 NetCDF 64-bit Offset Format

For these users, 64-bit offset format is a natural choice. It greatly eases the size restrictions of netCDF classic files (see Section 4.5 [64 bit Offset Limitations], page 38).

Files with the 64-bit offsets are identified with a "CDF\002" at the beginning of the file. In this documentation this format is called "64-bit offset format."

Since 64-bit offset format was introduced in version 3.6.0, earlier versions of the netCDF library can't read 64-bit offset files.

1.4.3 NetCDF-4 Format

In version 4.0, netCDF included another new underlying format: HDF5.

NetCDF-4 format files offer new features such as groups, compound types, variable length arrays, new unsigned integer types, parallel I/O access, etc. None of these new features can be used with classic or 64-bit offset files.

NetCDF-4 files can't be created at all, unless the netCDF configure script is run with –enable-netcdf-4. This also requires version 1.8.0 of HDF5.

For the netCDF-4.0 release, netCDF-4 features are only available from the C and Fortran interfaces. We plan to bring netCDF-4 features to the CXX API in a future release of netCDF.

NetCDF-4 files can't be read by any version of the netCDF library previous to 4.0. (But they can be read by HDF5, version 1.8.0 or better).

For more discussion of format issues see Section "Versions" in *The NetCDF Tutorial*.

1.5 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 4 [Structure], page 35.

The use of HDF5 as a data format adds significant overhead in metadata operations, less so in data access operations. We continue to study the challenge of implementing netCDF-4/HDF5 format without compromising performance.

1.6 Is NetCDF a Good Archive Format?

NetCDF classic or 64-bit offset formats 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), or the resulting data file may be compressed before storage (but must be uncompressed before it is read). Hence, using these netCDF formats may require more space than special-purpose archive formats that exploit knowledge of particular characteristics of specific datasets.

With netCDF-4/HDF5 format, the zlib library can provide compression on a per-variable basis. That is, some variables may be compressed, others not. In this case the compression and decompression of data happen transparently to the user, and the data may be stored, read, and written compressed.

1.7 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. For a description of reserved attributes (used by the netCDF library) and attribute conventions for generic application software, see Appendix B [Attribute Conventions], page 71.

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/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.

1.8 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, significant optimizations were implemented for supercomputer architectures, and the file format was formally specified in an appendix to the User's Guide.

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, max, min, product, and others) on netCDF arrays.

In 1996 and 1997, Joe Sirott implemented and made available the first implementation of a read-only netCDF interface for Java, Bill Noon made a Python module available for netCDF, and Konrad Hinsen contributed another netCDF interface for Python.

In May 1997, Version 3.3 of netCDF was released. This included a new type-safe interface for C and Fortran, as well as many other improvements. A month later, Charlie Zender released version 1.0 of the NCO (netCDF Operators) package, providing command-line utilities for general purpose operations on netCDF data.

Version 3.4 of Unidata's netCDF software, released in March 1998, included initial large file support, performance enhancements, and improved Cray platform support. Later in 1998, Dan Schmitt provided a Tcl/Tk interface, and Glenn Davis provided version 1.0 of netCDF for Java.

In May 1999, Glenn Davis, who was instrumental in creating and developing netCDF, died in a small plane crash during a thunderstorm. The memory of Glenn's passions and intellect continue to inspire those of us who worked with him.

In February 2000, an experimental Fortran 90 interface developed by Robert Pincus was released.

John Caron released netCDF for Java, version 2.0 in February 2001. This version incorporated a new high-performance package for multidimensional arrays, simplified the interface, and included OpenDAP (known previously as DODS) remote access, as well as remote netCDF access via HTTP contributed by Don Denbo.

In March 2001, NetCDF 3.5.0 was released. This release fully integrated the new Fortran 90 interface, enhanced portability, improved the C++ interface, and added a few new tuning functions.

Also in 2001, Takeshi Horinouchi and colleagues made a netCDF interface for Ruby available, as did David Pierce for the R language for statistical computing and graphics. Charles Denham released WetCDF, an independent implementation of the netCDF interface for Matlab, as well as updates to the popular netCDF Toolbox for Matlab.

In 2002, Unidata and collaborators developed NcML, an XML representation for netCDF data useful for cataloging data holdings, aggregation of data from multiple datasets, augmenting metadata in existing datasets, and support for alternative views of data. The Java interface currently provides access to netCDF data through NcML.

Additional developments in 2002 included translation of C and Fortran User Guides into Japanese by Masato Shiotani and colleagues, creation of a "Best Practices" guide for writing netCDF files, and provision of an Ada-95 interface by Alexandru Corlan.

In July 2003 a group of researchers at Northwestern University and Argonne National Laboratory (Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, and Rob Latham) contributed a new parallel interface for writing and reading netCDF data, tailored for use on high performance platforms with parallel I/O. The implementation built on the MPI-IO interface, providing portability to many platforms.

In October 2003, Greg Sjaardema contributed support for an alternative format with 64-bit offsets, to provide more complete support for very large files. These changes, with slight modifications at Unidata, were incorporated into version 3.6.0, released in December, 2004.

In 2004, thanks to a NASA grant, Unidata and NCSA began a collaboration to increase the interoperability of netCDF and HDF5, and bring some advanced HDF5 features to netCDF users.

In February, 2006, release 3.6.1 fixed some minor bugs.

In March, 2007, release 3.6.2 introduced an improved build system that used automake and libtool, and an upgrade to the most recent autoconf release, to support shared libraries and the netcdf-4 builds. This release also introduced the NetCDF Tutorial and example programs.

The first beta release of netCDF-4.0 was celebrated with a giant party at Unidata in April, 2007. Over 2000 people danced 'til dawn at the NCAR Mesa Lab, listening to the Flaming Lips and the Denver Gilbert & Sullivan repertory company.

In June, 2008, netCDF-4.0 was released. Version 3.6.3, the same code but with netcdf-4 features turned off, was released at the same time. The 4.0 release uses HDF5 1.8.1 as the data storage layer for netcdf, and introduces many new features including groups and user-defined types. The 3.6.3/4.0 releases also introduced handling of UTF8-encoded Unicode names.

1.9 What's New Since the Previous Release?

This Guide documents the 4.1.2-beta1 release of netCDF, which introduces a new storage format, netCDF-4/HDF5, while maintaining full backward compatibility.

New features available with netCDF-4/HDF5 files include:

- The use of groups to organize datasets.
- New unsigned integer data types, 64-bit integer types, and a string type.
- A user defined compound type, which can be constructed by users to match a C struct or other arbitrary organization of types.
- A variable length array type.
- Multiple unlimited dimensions.
- Support for parallel I/O.

More information about netCDF-4 can be found at the netCDF web page `http://www.unidata.ucar.edu/netcdf/netcdf-4`.

1.10 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 have been removed or relaxed in netCDF-4 files, but still apply to netCDF classic and netCDF 64-bit offset files.

Currently, netCDF classic and 64-bit offset formats offer a limited number of external numeric data types: 8-, 16-, 32-bit integers, or 32- or 64-bit floating-point numbers. (The netCDF-4 format adds 64-bit integer types and unsigned integer types.) 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 netCDF-4/HDF5 format, new unsigned integers (of various sizes), 64-bit integers, and the string type allow greater expression of scientific data. The new VLEN and COMPOUND types allow users to organize data in new ways.

With the classic netCDF file format, there are constraints that limit how a dataset is structured to store more than 2 *GiBytes* (2^30 or 1,073,741,824 bytes, as compared to a *Gbyte*, which is 1,000,000,000 bytes.) of data in a single netCDF dataset. (see Section 4.6 [Classic Limitations], page 38). This limitation is a result of 32-bit offsets used for storing relative offsets within a classic netCDF format file. Since one of the goals of netCDF is portable data and some computing platforms still can't deal with files larger than 2 GiB, it is best to keep files that must be portable below this limit. Nevertheless, it is possible to create and access netCDF files larger than 2 GiB on platforms that provide support for such files (see Section 4.4 [Large File Support], page 37).

The new 64-bit offset format allows large files, and makes it easy to create to create fixed variables of about 4 GiB, and record variables of about 4 GiB per record. (see Section 4.5 [64 bit Offset Limitations], page 38). However, old netCDF applications will not be able to read the 64-bit offset files until they are upgraded to at least version 3.6.0 of netCDF (i.e. the version in which 64-bit offset format was introduced).

With the netCDF-4/HDF5 format size limitations are further relaxed, and files can be as large as the underlying file system supports. NetCDF-4/HDF5 files are unreadable to the netCDF library before version 4.0.

Another limitation of the classic (and 64-bit offset) 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 classic netCDF model does not permit variables with several unlimited dimensions or the use of multiple unlimited dimensions in different variables within the same dataset. Variables that have non-rectangular shapes (for example, ragged arrays) cannot be represented conveniently.

In netCDF-4/HDF5 files, multiple unlimited dimensions are fully supported. Any variable can be defined with any combination of limited and unlimited dimensions.

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 classic netCDF data model does not support nested data structures such as trees, nested arrays, or other recursive structures. (This limitation also applies to 64-bit offset files.) 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.

In netCDF-4/HDF5 format files, the introduction of the compound type allows the creation of complex data types, involving any combination of types. The VLEN type allows efficient storage of ragged arrays, and the introduction of hierarchical groups allows users to organize data.

Finally, for classic and 64-bit offset files, concurrent access to a netCDF dataset is limited. One writer and multiple readers may access data in a single dataset simultaneously, but there is no support for multiple concurrent writers.

NetCDF-4 supports parallel read/write access to netCDF-4/HDF5 files, using the underlying HDF5 library and parallel read/write access to classic and 64-bit offset files using the parallel-netcdf library.

For more information about HDF5, see the HDF5 web site: `http://hdfgroup.org/HDF5/`.

For more information about parallel-netcdf, see their web site: `http://www.mcs.anl.gov/parallel-netcdf`

1.11 Plans for NetCDF

Future versions of NetCDF will include the following features:

1. Extensions of netCDF-4 features to C++ API and to tools ncgen/ncdump.
2. Better documentation and more examples.

1.12 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.

2.1.1 Expanded Model in NetCDF-4 Files

Files created with the netCDF-4 format have access to an expanded data model, which includes named groups. Groups, like directories in a Unix file system, are hierarchically organized, to arbitrary depth. They can be used to organize large numbers of variables.

Each group acts as an entire netCDF dataset in the classic model. That is, each group may have attributes, dimensions, and variables, as well as other groups.

The default root is the root group, which allows the classic netCDF data model to fit neatly into the new model.

Dimensions are scoped such that they can be seen in all descendant groups. That is, dimensions can be shared between variables in different groups, if they are defined in a parent group.

In netCDF-4 files, the user may also define a type. For example a compound type may hold information from an array of C structures, or a variable length array allows the user to read and write arrays of variable length arrays.

Variables, groups, and types share a namespace. Within the same group, a variable, groups, and types must have unique names. (That is, a type and variable may not have the same name within the same group, and similarly for sub-groups of that group.)

Groups and user defined types are only available in files created in the NetCDF-4/HDF5 format. They are not available for classic or 64-bit offset format files.

2.1.2 Naming Conventions

The names of dimensions, variables and attributes (and, in netCDF-4 files, groups, user-defined types, compound member names, and enumeration symbols) consist of arbitrary sequences of alphanumeric characters, underscore ’`_`’, period ’.’, plus ’`+`’, hyphen ’-’, or at sign ’`@`’, but beginning with a letter or underscore. However names commencing with underscore are reserved for system use. Case is significant in netCDF names. A zero-length name is not allowed. Some widely used conventions restrict names to only alphanumeric characters or underscores. Beginning with versions 3.6.3 and 4.0, names may also include UTF-8 encoded Unicode characters as well as other special characters, except for the character ’/’, which may not appear in a name. Names that have trailing space characters are also not permitted.

2.1.3 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 the ncdump utility for producing human-oriented CDL text files from binary netCDF datasets and vice versa. (The ncdump utility has recently been enhanced to accommodate netCDF-4 features in the CDL output, but the example here is restricted to netCDF-3 CDL.)

```
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 5.5 [ncdump], page 65). 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 5.4 [ncgen], page 63). This version of ncgen can produce netcdf-3 or netcdf-4 files and can utilize CDL input that includes the netcdf-4 data model constructs. The older ncgen program is still available under the name ncgen3.

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 {
  types: [netcdf-4 only]
  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.

The ncgen utility provides a command line option which indicates the desired output format. Limitations are enforced for the selected format - that is, some CDL files may be expressible only in 64-bit offset or NetCDF-4 format.

For example, trying to create a file with very large variables in classic format may result in an error because size limits are violated.

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 classic or 64-bit offset netCDF dataset can have the length UNLIMITED. In a netCDF-4 dataset, any number of unlimited dimensions can be used.

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 classic or 64-bit offset 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 first dimension in corresponding C array declarations.

A netCDF-4 dataset may have multiple unlimited dimensions, and there are no restrictions on their order in the list of a variables dimensions.

To grow variables along an unlimited dimension, write the data using any of the netCDF data writing functions, and specify the index of the unlimited dimension to the desired record number. The netCDF library will write however many records are needed (using the fill value, unless that feature is turned off, to fill in any intervening records).

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. Use the “/” character to include group information (netCDF-4 output only).

There are four dimensions in the above example: lat, lon, level, and time (see Section 2.1 [Data Model], page 17). 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 (since version 3.1 of netCDF) to use the same dimension more than once in specifying a variable shape. 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. In classic and 64-bit offset files, only the original six types are available (byte, character, short, int, float, and double). Variables in netCDF-4 files may also use unsigned short, unsigned int, 64-bit int, unsigned 64-bit int, or string. Or the user may define a type, as an opaque blob of bytes, as an array of variable length arrays, or as a compound type, which acts like a C struct.

For more information on types for the C interface, see Section "Variable Types" in *The NetCDF C Interface Guide* in The NetCDF C Interface Guide.

For more information on types for the Fortran interface, see Section "Variable Types" in *The NetCDF Fortran 77 Interface Guide* in The NetCDF Fortran 77 Interface Guide.

In the CDL notation, classic and 64-bit offset type can be used. They are given the simpler names byte, char, short, int, float, and double. The name real may be used as a synonym for float in the CDL notation. The name long is a deprecated synonym for int. For the exact meaning of each of the types see Section 3.1 [External Types], page 25. The ncgen utility supports new primitive types with names ubyte, ushort, uint, int64, uint64, and string.

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 0 for C programs, 1 in Fortran programs. Thus the 700 millibar level would have an index value of 2 in the example above in a C program, and 3 in a Fortran program.

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).

In netCDF-4 file, attributes can also be added at the group level.

An attribute has an associated variable (the null "global variable" for a global or group-level 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.

For the netCDF classic model, 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 5.3 [CDL Constants], page 62).

The extended CDL syntax for the enhanced data model supported by netCDF-4 allows optional type specifications, including user-defined types, for attributes of user-defined types. See ncdump output or the reference documentation for ncgen4 for details of the extended CDL systax.

In the netCDF example (see Section 2.1 [Data Model], page 17), 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, called "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. For information about units, long_name, valid_min, valid_max, valid_range, scale_factor, add_offset, _FillValue, and other conventional attributes, see Appendix B [Attribute Conventions], page 71.

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 classic or 64-bit offset format dataset can incur the same expense as copying the dataset. For a more extensive discussion see Chapter 4 [Structure], page 35.

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 is specified when it is created, unless it is a zero-length attribute.

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 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 atomic external types supported by the netCDF interface are:

C name	Fortran name	storage
NC_BYTE	nf_byte	8-bit signed integer
NC_CHAR	nf_char	8-bit unsigned integer
NC_SHORT	nf_short	16-bit signed integer
NC_USHORT	nf_ushort	16-bit unsigned integer *
NC_INT (or NC_LONG)	nf_int	32-bit signed integer
NC_UINT	nf_uint	32-bit unsigned integer *
NC_INT64	nf_int64	64-bit signed integer *
NC_UINT64	nf_uint64	64-bit unsigned integer *
NC_FLOAT	nf_float	32-bit floating point
NC_DOUBLE	nf_double	64-bit floating point
NC_STRING	nf_string	variable length character string *

* These types are available only for netCDF-4 format files. All the unsigned ints (except NC_CHAR), the 64-bit ints, and string type are for netCDF-4 files only.

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.

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, ushort, int, uint, int64, uint64, float or real, double, string) 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.

For the correspondence between netCDF external data types and the data types of a language see Section 2.3 [Variables], page 20.

3.2 Data Structures in Classic and 64-bit Offset Files

The only kind of data structure directly supported by the netCDF classic (and 64-bit offset) 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 in netCDF classic or 64-bit offset formats, 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 netCDF classic 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. (NetCDF-4 includes native support for variable length arrays. See below.)

```
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 classic or 64-bit offset 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.

3.3 NetCDF-4 User Defined Data Types

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

In addition to the new atomic types the user may define types.

Types are defined in define mode, and must be fully defined before they are used. New types may be added to a file by re-entering define mode.

Once defined the type may be used to create a variable or attribute.

Types may be nested in complex ways. For example, a compound type containing an array of VLEN types, each containing variable length arrays of some other compound type, etc. Users are cautioned to keep types simple. Reading data of complex types can be challenging for Fortran users.

Types may be defined in any group in the data file, but they are always available globally in the file.

Types cannot have attributes (but variables of the type may have attributes).

Only files created with the netCDF-4/HDF5 mode flag (NC_NETCDF4, NF_NETCDF4, or NF90_NETCDF4), but without the classic model flag (NC_CLASSIC_MODEL, NF_CLASSIC_MODEL, or NF90_CLASSIC_MODEL.)

Once types are defined, use their ID like any other type ID when defining variables or attributes. Each API has functions to read and write variables and attributes of any type. Use these functions to read and write variables and attributes of user defined type. In C use nc_put_att/nc_get_att and the nc_put_var/nc_get_var, nc_put_var1/nc_get_var1, nc_put_vara/nc_get_vara, or nc_put_vars/nc_get_vars functons to access attribute and variable data of user defined type.

3.3.1 Compound Types

Compound types allow the user to combine atomic and user-defined types into C-like structs. Since users defined types may be used within a compound type, they can contain nested compound types.

Users define a compound type, and (in their C code) a corresponding C struct. They can then use the nc_put_var[1asm] calls to write multi-dimensional arrays of these structs, and nc_get_var[1asm] calls to read them. (For example, the nc_put_varm function will write mapped arrays of these structs.)

While structs, in general, are not portable from platform to platform, the HDF5 layer (when installed) performs the magic required to figure out your platform's idiosyncrasies, and adjust to them. The end result is that HDF5 compound types (and therefore, netCDF-4 compound types), are portable.

For more information on creating and using compound types, see Section "Compound Types" in *The NetCDF C Interface Guide* in The NetCDF C Interface Guide.

3.3.2 VLEN Types

Variable length arrays can be used to create a ragged array of data, in which one of the dimensions varies in size from point to point.

An example of VLEN use would the to store a 1-D array of dropsonde data, in which the data at each drop point is of variable length.

There is no special restriction on the dimensionality of VLEN variables. It's possible to have 2D, 3D, 4D, etc. data, in which each point contains a VLEN.

A VLEN has a base type (that is, the type that it is a VLEN of). This may be one of the atomic types (forming, for example, a variable length array of NC_INT), or it can be another user defined type, like a compound type.

With VLEN data, special memory allocation and deallocation procedures must be followed, or memory leaks may occur.

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

For more information on creating and using variable length arrays, see Section "Variable Length Arrays" in *The NetCDF C Interface Guide* in The NetCDF C Interface Guide.

3.3.3 Opaque Types

Opaque types allow the user to store arrays of data blobs of a fixed size.

For more information on creating and using opaque types, see Section "Opaque Type" in *The NetCDF C Interface Guide* in The NetCDF C Interface Guide.

3.3.4 Enum Types

Enum types allow the user to specify an enumeration.

For more information on creating and using enum types, see Section "Enum Type" in *The NetCDF C Interface Guide* in The NetCDF C Interface Guide.

3.3.5 Groups

Although not a type of data, groups can help organize data within a dataset. Like a directory structure on a Unix file-system, the grouping feature allows users to organize variables and dimensions into distinct, named, hierarchical areas, called groups. For more information on groups types, see Section "Groups" in *The NetCDF C Interface Guide* in The NetCDF C Interface Guide.

3.4 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 in classic and 64-bit offset format is direct. Access to netCDF-4 data is buffered by the HDF5 layer. In either case 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 and FORTRAN 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.

3.4.1 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 sub-sampled 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.

3.4.2 A C Example of Array-Section Access

Assume that in our earlier example of a netCDF dataset (see Section 2.1 [Network Common Data Form Language (CDL)], page 17), 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.

A corresponding C variable that holds data for only one level might be declared as

```
#define LATS  5
#define LONS 10
#define LEVELS 1
#define TIMES 3                 /* currently */
    ...
float   temp[TIMES*LEVELS*LATS*LONS];

to keep the data in a one-dimensional array, or

    ...
float   temp[TIMES][LEVELS][LATS][LONS];
```

using a multidimensional array declaration.

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 (0, 1, 0, 0) in C, 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 (3, 1, 5, 10) in C, (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 last dimension, lon, varying fastest:

```
temp[0][1][0][0]
temp[0][1][0][1]
temp[0][1][0][2]
temp[0][1][0][3]

      ...
```

```
temp[2][1][4][7]
temp[2][1][4][8]
temp[2][1][4][9]
```

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.

3.4.3 More on General Array Section Access for C

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 C, a point's coordinate offset vector is the same as its coordinate vector.

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.

For example, the following C definitions

```
struct vel {
    int flags;
    float u;
    float v;
} vel[NX][NY];
ptrdiff_t imap[2] = {
    sizeof(struct vel),
    sizeof(struct vel)*NY
};
```

where imap is the index mapping vector, can be used to access the memory-resident values of the netCDF variable, vel(NY,NX), even though the dimensions are transposed and the data is contained in a 2-D array of structures rather than a 2-D array of floating-point values.

A detailed example of mapped array access is presented in the description of the interfaces for mapped array access. See Section "nc_put_varm_ type" in *The NetCDF C Interface Guide*.

Note that, although the netCDF abstraction allows the use of subsampled or mapped array-section access there 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.

3.4.4 A Fortran Example of Array-Section Access

Assume that in our earlier example of a netCDF dataset (see Section 2.1 [Data Model], page 17), 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, 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 a FORTRAN declarations for a variable that holds data for only one level is

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

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, 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, 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.

3.4.5 More on General Array Section Access for Fortran

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, 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 "nf_put_varm_ type" in *The NetCDF Fortran 77 Interface Guide*.

Note that, although the netCDF abstraction allows the use of subsampled or mapped array-section access there 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.

3.5 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.

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.

4 File Structure and Performance

This chapter describes the file structure of a netCDF classic or 64-bit offset 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.

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.

4.1 Parts of a NetCDF Classic File

A netCDF classic or 64-bit offset 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.

By default, this header has little 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, with a small amount of extra space from rounding up to the nearest disk block size. 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 the enddef function is called: nc_enddef in C (see Section "nc_enddef" in *The NetCDF C Interface Guide*), NF_ENDDEF in Fortran (see Section "NF_ENDDEF" in *The NetCDF Fortran 77 Interface Guide*), after a previous call to the redef function: nc_redef in C (see Section "nc_redef" in *The NetCDF C Interface Guide*) or NF_REDEF in Fortran (see Section "NF_REDEF" in *The NetCDF Fortran 77 Interface Guide*). 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.

Alternatively, you can use an alternative version of the enddef function with two underbar characters instead of one to explicitly reserve extra space in the file header when the file is created: in C nc__enddef (see Section "nc__enddef" in *The NetCDF C Interface Guide*), in Fortran NF__ENDDEF (see Section "NF__ENDDEF" in *The NetCDF Fortran 77 Interface Guide*), after a previous call to the redef function. This avoids the expense of moving all the data later by reserving enough extra space in the header to accommodate anticipated changes, such as the addition of new attributes or the extension of existing string attributes to hold longer strings.

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 nc_sync/NF_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.

For more detail see Appendix C [File Format], page 75.

4.2 Parts of a NetCDF-4 HDF5 File

NetCDF-4 files are created with the HDF5 library, and are HDF5 files in every way, and can be read without the netCDF-4 interface. (Note that modifying these files with HDF5 will almost certainly make them unreadable to netCDF-4.)

Groups in a netCDF-4 file correspond with HDF5 groups (although the netCDF-4 tree is rooted not at the HDF5 root, but in group "_netCDF").

Variables in netCDF coo-respond with identically named datasets in HDF5. Attributes similarly.

Since there is more metadata in a netCDF file than an HDF5 file, special datasets are used to hold netCDF metadata.

The _netcdf_dim_info dataset (in group _netCDF) contains the ids of the shared dimensions, and their length (0 for unlimited dimensions).

The _netcdf_var_info dataset (in group _netCDF) holds an array of compound types which contain the variable ID, and the associated dimension ids.

4.3 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. Classic or 64-bit offset 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.

4.4 Large File Support

It is possible to write netCDF files that exceed 2 GiByte on platforms that have "Large File Support" (LFS). Such files are platform-independent to other LFS platforms, but trying to open them on an older platform without LFS yields a "file too large" error.

Without LFS, no files larger than 2 GiBytes can be used. The rest of this section applies only to systems with LFS.

The original binary format of netCDF (classic format) limits the size of data files by using a signed 32-bit offset within its internal structure. Files larger than 2 GiB can be created, with certain limitations. See Section 4.6 [Classic Limitations], page 38.

In version 3.6.0, netCDF included its first-ever variant of the underlying data format. The new format introduced in 3.6.0 uses 64-bit file offsets in place of the 32-bit offsets. There are still some limits on the sizes of variables, but the new format can create very large datasets. See Section 4.5 [64 bit Offset Limitations], page 38.

NetCDF-4 variables and files can be any size supported by the underlying file system.

The original data format (netCDF classic), is still the default data format for the netCDF library.

The following table summarizes the size limitations of various permutations of LFS support, netCDF version, and data format. Note that 1 GiB = 2^30 bytes or about 1.07e+9 bytes, 1 EiB = 2^60 bytes or about 1.15e+18 bytes. Note also that all sizes are really 4 bytes less than the ones given below. For example the maximum size of a fixed variable in netCDF 3.6 classic format is really 2 GiB - 4 bytes.

Limit	No LFS	v3.5	v3.6/classic	v3.6/64-bit offset	v4.0/netCDF-4
Max File Size	2 GiB	8 EiB	8 EiB	8 EiB	??
Max Number of Fixed Vars > 2 GiB	0	1 (last)	1 (last)	2^32	??

Max Record Vars w/ Rec Size > 2 GiB	0	1 (last)	1 (last)	2^32	??
Max Size of Fixed/Record Size of Record Var	2 GiB	2 GiB	2 GiB	4 GiB	??
Max Record Size	2 GiB/nrecs	4 GiB	8 EiB/nrecs	8 EiB/nrecs	??

For more information about the different file formats of netCDF See Section 1.4 [Which Format], page 6.

4.5 NetCDF 64-bit Offset Format Limitations

Although the 64-bit offset format allows the creation of much larger netCDF files than was possible with the classic format, there are still some restrictions on the size of variables.

It's important to note that without Large File Support (LFS) in the operating system, it's impossible to create any file larger than 2 GiBytes. Assuming an operating system with LFS, the following restrictions apply to the netCDF 64-bit offset format.

No fixed-size variable can require more than 2^32 - 4 bytes (i.e. 4GiB - 4 bytes, or 4,294,967,292 bytes) of storage for its data, unless it is the last fixed-size variable and there are no record variables. When there are no record variables, the last fixed-size variable can be any size supported by the file system, e.g. terabytes.

A 64-bit offset format netCDF file can have up to 2^32 - 1 fixed sized variables, each under 4GiB in size. If there are no record variables in the file the last fixed variable can be any size.

No record variable can require more than 2^32 - 4 bytes of storage for each record's worth of data, unless it is the last record variable. A 64-bit offset format netCDF file can have up to 2^32 - 1 records, of up to 2^32 - 1 variables, as long as the size of one record's data for each record variable except the last is less than 4 GiB - 4.

Note also that all netCDF variables and records are padded to 4 byte boundaries.

4.6 NetCDF Classic Format Limitations

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

The maximum size of a record in the classic format in versions 3.5.1 and earlier is 2^32 - 4 bytes, or about 4 GiB. In versions 3.6.0 and later, there is no such restriction on total record size for the classic format or 64-bit offset format.

If you don't use the unlimited dimension, only one variable can exceed 2 GiB 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 GiB.

The limit is really 2^31 - 4. If you were to specify a variable size of 2^31 -3, for example, it would be rounded up to the nearest multiple of 4 bytes, which would be 2^31, which is larger than the largest signed integer, 2^31 - 1.

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, record variables may exceed 2 GiB in size, as long as the offset of the start of each record variable within a record is less than 2 GiB - 4. 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, 2400000000 bytes per record
       double var1(t, x, y, z);
       double var2(t, x, y, z);
       double var3(t, x, y, z);
    }
```

4.7 The NetCDF-3 I/O Layer

The following discussion applies only to netCDF classic and 64-bit offset files. For netCDF-4 files, the I/O layer is the HDF5 library.

For netCDF classic and 64-bit offset files, 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 nc_sync/NF_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; The C function nc_sync (see Section "nc_sync" in *The NetCDF C Interface Guide*), or the Fortran function NF_SYNC (see Section "NF_SYNC" in *The NetCDF Fortran 77 Interface Guide*), also brings header changes up-to-date (for example, changes to attribute values). Opening the file with the NC_SHARE (in C) or the NF_SHARE (in Fortran) 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 classic or 64-bit offset format 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.

4.8 UNICOS Optimization

It should be noted that no UNICOS platform has been available at Unidata for netCDF testing for some years. The following information is left here for historical reasons.

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 its 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.

4.9 Improving Performance With Chunking

NetCDF may use HDF5 as a storage format (when files are created with NC_NETCDF4/NF_NETCDF4/NF90_NETCDF4). For those files, the writer may control the size of the chunks of data that are written to the HDF5, along with other aspects of the data, such as endianness, a shuffle and checksum filter, on-the-fly compression/decompression of the data.

The chunk sizes of a variable are specified after the variable is defined, but before any data are written. If chunk sizes are not specified for a variable, default chunk sizes are chosen by the library.

The selection of good chunk sizes is a complex topic, and one that data writers must grapple with. Once the data are written, there is no way to change the chunk sizes except to copy the data to a new variable.

Chunks should match read access patterns; the best chunk performance can be achieved by writing chunks which exactly match the size of the subsets of data that will be read. When multiple read access patterns are to be used, there is no one way to best set the chunk sizes.

Some good discussion of chunking can be found in the HDF5-EOS XIII workshop presentation (`http://hdfeos.org/workshops/ws13/presentations/day1/HDF5-EOSXIII-Advanced-Chunking.pp`

4.9.1 The Chunk Cache

When data are first read or written to a netCDF-4/HDF5 variable, the HDF5 library opens a cache for that variable. The default size of that cache (settable with the –with-chunk-cache-size at netCDF build time).

For good performance your chunk cache must be larger than one chunk of your data - preferably that it be large enough to hold multiple chunks of data.

In addition, when a file is opened (or a variable created in an open file), the netCDF-4 library checks to make sure the default chunk cache size will work for that variable. The cache will be large enough to hold N chunks, up to a maximum size of M bytes. (Both N and M are settable at configure time with the –with-default-chunks-in-cache and the –with-max-default-cache-size options to the configure script. Currently they are set to 10 and 64 MB.)

To change the default chunk cache size, use the set_chunk_cache function before opening the file. C programmers see Section "nc_set_chunk_cache" in *The NetCDF C Interface Guide*, Fortran 77 programmers see Section "NF_SET_CHUNK_CACHE" in *The NetCDF Fortran 77 Interface Guide*). Fortran 90 programmers use the optional cache_size, cache_nelems, and cache_preemption parameters to nf90_open/nf90_create to change the chunk size before opening the file.

To change the per-variable cache size, use the set_var_chunk_cache function at any time on an open file. C programmers see Section "nc_set_var_chunk_cache" in *The NetCDF C Interface Guide*, Fortran 77 programmers see Section "NF_SET_VAR_CHUNK_CACHE" in *The NetCDF Fortran 77 Interface Guide*,).

4.9.2 The Default Chunking Scheme in version 4.1 (and 4.1.1)

When the data writer does not specify chunk sizes for variable, the netCDF library has to come up with some default values.

The C code below determines the default chunks sizes.

For unlimited dimensions, a chunk size of one is always used. Users are advised to set chunk sizes for large data sets with one or more unlimited dimensions, since a chunk size of one is quite inefficient.

For fixed dimensions, the algorithm below finds a size for the chunk sizes in each dimension which results in chunks of DEFAULT_CHUNK_SIZE (which can be modified in the netCDF configure script).

```
     /* Unlimited dim always gets chunksize of 1. */
     if (dim->unlimited)
chunksize[d] = 1;
     else
chunksize[d] = pow((double)DEFAULT_CHUNK_SIZE/type_size,
   1/(double)(var->ndims - unlimdim));
```

4.9.3 The Default Chunking Scheme in version 4.0.1

In the 4.0.1 release, the default chunk sizes were chosen with a different scheme, as demonstrated in the following C code:

```
/* These are limits for default chunk sizes. (2^16 and 2^20). */
#define NC_LEN_TOO_BIG 65536
#define NC_LEN_WAY_TOO_BIG 1048576

      /* Now we must determine the default chunksize. */
      if (dim->unlimited)
         chunksize[d] = 1;
      else if (dim->len < NC_LEN_TOO_BIG)
         chunksize[d] = dim->len;
      else if (dim->len > NC_LEN_TOO_BIG && dim->len <= NC_LEN_WAY_TOO_BIG)
         chunksize[d] = dim->len / 2 + 1;
      else
         chunksize[d] = NC_LEN_WAY_TOO_BIG;
```

As can be seen from this code, the default chunksize is 1 for unlimited dimensions, otherwise it is the full length of the dimension (if it is under NC_LEN_TOO_BIG), or half the size of the dimension (if it is between NC_LEN_TOO_BIG and NC_LEN_WAY_TOO_BIG), and, if it's longer than NC_LEN_WAY_TOO_BIG, it is set to NC_LEN_WAY_TOO_BIG.

Our experience is that these defaults work well for small data sets, but once variable size reaches the GB range, the user is better off determining chunk sizes for their read access patterns.

In particular, the idea of using 1 for the chunksize of an unlimited dimension works well if the data are being read a record at a time. Any other read access patterns will result in slower performance.

4.9.4 Chunking and Parallel I/O

When files are opened for read/write parallel I/O access, the chunk cache is not used. Therefore it is important to open parallel files with read only access when possible, to achieve the best performance.

4.9.5 A Utility to Help Benchmark Results: bm_file

The bm_file utility may be used to copy files, from one netCDF format to another, changing chunking, filter, parallel I/O, and other parameters. This program may be used for benchmarking netCDF performance for user data files with a range of choices, allowing data producers to pick settings that best serve their user base.

NetCDF must have been configured with –enable-benchmarks at build time for the bm_file program to be built. When built with –enable-benchmarks, netCDF will include tests (run with “make check”) that will run the bm_file program on sample data files.

Since data files and their access patterns vary from case to case, these benchmark tests are intended to suggest further use of the bm_file program for users.

Here’s an example of a call to bm_file:

```
./bm_file -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:1024:256:256 tst_elena_int_3D.nc
```

Generally a range of settings must be tested. This is best done with a shell script, which calls bf_file repeatedly, to create output like this:

```
*** Running benchmarking program bm_file for simple shorts test files, 1D to 6D...
input format, output_format, input size, output size, meta read time, meta write time,
1, 4, 200092, 207283, 1613, 1054, 409, 312, 0, 1208, 1551, 488.998, 641.026, 128.949,
1, 4, 199824, 208093, 1545, 1293, 397, 284, 0, 1382, 1563, 503.053, 703.211, 127.775,
1, 4, 194804, 204260, 1562, 1611, 390, 10704, 0, 1627, 2578, 499.159, 18.1868, 75.5128
1, 4, 167196, 177744, 1531, 1888, 330, 265, 0, 12888, 1301, 506.188, 630.347, 128.395,
1, 4, 200172, 211821, 1509, 2065, 422, 308, 0, 1979, 1550, 473.934, 649.351, 129.032,
1, 4, 93504, 106272, 1496, 2467, 191, 214, 0, 32208, 809, 488.544, 436.037, 115.342, 0
*** SUCCESS!!!
```

Such tables are suitable for import into spreadsheets, for easy graphing of results.

Several test scripts are run during the “make check” of the netCDF build, in the nc_test4 directory. The following example may be found in nc_test4/run_bm_elena.sh.

```
#!/bin/sh

# This shell runs some benchmarks that Elena ran as described here:
# http://hdfeos.org/workshops/ws06/presentations/Pourmal/HDF5_IO_Perf.pdf

# $Id: netcdf.texi,v 1.82 2010/05/15 20:43:13 dmh Exp $
```

```
set -e
echo ""

echo "*** Testing the benchmarking program bm_file for simple float file, no compressi
./bm_file -h -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:1024:16:256 tst_elena_int_3D.nc
./bm_file -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:1024:256:256 tst_elena_int_3D.nc
./bm_file -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:512:64:256 tst_elena_int_3D.nc
./bm_file -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:512:256:256 tst_elena_int_3D.nc
./bm_file -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:256:64:256 tst_elena_int_3D.nc
./bm_file -d -f 3 -o  tst_elena_out.nc -c 0:-1:0:256:256:256 tst_elena_int_3D.nc
echo '*** SUCCESS!!!'

exit 0
```

The reading that bm_file does can be tailored to match the expected access pattern.

The bm_file program is controlled with command line options.

```
./bm_file
bm_file -v [-s N]|[-t V:S:S:S -u V:C:C:C -r V:I:I:I] -o file_out -f N -h -c V:C:C,V:C:
  [-v]        Verbose
  [-o file]   Output file name
  [-f N]      Output format (1 - classic, 2 - 64-bit offset, 3 - netCDF-4, 4 - netCDF4
  [-h]        Print output header
  [-c V:Z:S:C:C:C[,V:Z:S:C:C:C, etc.]] Deflate, shuffle, and chunking parameters for v
  [-t V:S:S:S[,V:S:S:S, etc.]] Starts for reads/writes
  [-u V:C:C:C[,V:C:C:C, etc.]] Counts for reads/writes
  [-r V:I:I:I[,V:I:I:I, etc.]] Incs for reads/writes
  [-d]        Doublecheck output by rereading each value
  [-m]        Do compare of each data value during doublecheck (slow for large files!)
  [-p]        Use parallel I/O
  [-s N]      Denom of fraction of slowest varying dimension read.
  [-i]        Use MPIIO (only relevant for parallel builds).
  [-e 1|2]    Set the endianness of output (1=little 2=big).
  file        Name of netCDF file
```

4.10 Parallel Access with NetCDF-4

Use the special parallel open (or create) calls to open (or create) a file, and then to use parallel I/O to read or write that file. C programmers see Section "nc_open_par" in *The NetCDF C Interface Guide*, Fortran 77 programmers see Section "NF_OPEN_PAR" in *The NetCDF Fortran 77 Interface Guide*). Fortran 90 programmers use the optional comm and info parameters to nf90_open/nf90_create to initiate parallel access.

Note that the chunk cache is turned off if a file is opened for parallel I/O in read/write mode. Open the file in read-only mode to engage the chunk cache.

NetCDF uses the HDF5 parallel programming model for parallel I/O with netCDF-4/HDF5 files. The HDF5 tutorial (`http://hdfgroup.org/HDF5//HDF5/Tutor`) is a good reference.

For classic and 64-bit offset files, netCDF uses the parallel-netcdf (formerly pnetcdf) library from Argonne National Labs/Nortwestern University. For parallel access of classic and 64-bit offset files, netCDF must be configured with the –with-pnetcdf option at build time. See the parallel-netcdf site for more information (`http://www.mcs.anl.gov/parallel-netcdf`).

4.11 Interoperability with HDF5

To create HDF5 files that can be read by netCDF-4, use HDF5 1.8, which is not yet released. However most (but not all) of the necessary features can be found in their latest development snapshot.

HDF5 has some features that will not be supported by netCDF-4, and will cause problems for interoperability:

- HDF5 allows a Group to be both an ancestor and a descendant of another Group, creating cycles in the subgroup graph. HDF5 also permits multiple parents for a Group. In the netCDF-4 data model, Groups form a tree with no cycles, so each Group (except the top-level unnamed Group) has a unique parent.
- HDF5 supports "references" which are like pointers to objects and data regions within a file. The netCDF-4 data model omits references.
- HDF5 supports some primitive types that are not included in the netCDF-4 data model, including H5T_TIME and H5T_BITFIELD.
- HDF5 supports multiple names for data objects like Datasets (netCDF-4 variables) with no distinguished name. The netCDF-4 data model requires that each variable, attribute, dimension, and group have a single distinguished name.

These are fairly easy requirements to meet, but there is one relating to shared dimensions which is a little more challenging. Every HDF5 dataset must have a dimension scale attached to each dimension.

Dimension scales are a new feature for HF 1.8, which allow specification of shared dimensions.

(In the future netCDF-4 will be able to deal with HDF5 files which do not have dimension scales. However, this is not expected before netCDF 4.1.)

Finally, there is one feature which is missing from all current HDF5 releases, but which will be in 1.8 - the ability to track object creation order. As you may know, netCDF keeps track of the creation order of variables, dimensions, etc. HDF5 (currently) does not.

There is a bit of a hack in place in netCDF-4 files for this, but that hack will go away when HDF5 1.8 comes out.

Without creation order, the files will still be readable to netCDF-4, it's just that netCDF-4 will number the variables in alphabetical, rather than creation, order.

Interoperability is a complex task, and all of this is in the alpha release stage. It is tested in libsrc4/tst_interops.c, which contains some examples of how to create HDF5 files, modify them in netCDF-4, and then verify them in HDF5. (And vice versa).

4.12 DAP Support

Beginning with NetCDF version 4.1, optional support is provided for accessing data through OPeNDAP servers using the DAP protocol.

DAP support is automatically enabled if a usable curl library can be located using the curl-config program or by the –with-curl-config flag. It can forcibly be enabled or disabled using the –enable-dap flag or the –disable-dap flag, respectively. If enabled, then DAP support requires access to the curl library. Refer to the installation manual for details Section “Top” in *The NetCDF Installation and Porting Guide*.

DAP uses a data model that is different from that supported by netCDF, either classic or enhanced. Generically, the DAP data model is encoded textually in a DDS (Dataset Descriptor Structure). There is a second data model for DAP attributes, which is encoded textually in a DAS (Dataset Attribute Structure). For detailed information about the DAP DDS and DAS, refer to the OPeNDAP web site `http://opendap.org`.

4.12.1 Accessing OPeNDAP Data

In order to access an OPeNDAP data source through the netCDF API, the file name normally used is replaced with a URL with a specific format. The URL is composed of four parts.

1. Client parameters - these are prefixed to the front of the URL and are of the general form [<name>] or [<name>=value]. Examples include [cache=1] and [netcdf3].
2. URL - this is a standard form URL such as http://test.opendap.org:8080/dods/dts/test.01
3. Constraints - these are suffixed to the URL and take the form “?<projections>&selections”. The meaning of the terms projection and selection is somewhat complicated; and the OPeNDAP web site, `http://www.opendap.or`, should be consulted. The interaction of DAP constraints with netCDF is complex and at the moment requires an understanding of how DAP is translated to netCDF.

It is possible to see what the translation does to a particular DAP data source in either of two ways. First, one can examine the DDS source through a web browser and then examine the translation using the ncdump -h command to see the netCDF Classic translation. The ncdump output will actually be the union of the DDS with the DAS, so to see the complete translation, it is necessary to view both.

For example, if a web browser is given the following, the first URL will return the DDS for the specified dataset, and the second URL will return the DAS for the specified dataset.

```
http://test.opendap.org:8080/dods/dts/test.01.dds
http://test.opendap.org:8080/dods/dts/test.01.das
```

Then by using the following ncdump command, it is possible to see the equivalent netCDF Classic translation.

```
ncdump -h http://test.opendap.org:8080/dods/dts/test.01
```

The DDS output from the web server should look like this.

```
Dataset {
    Byte b;
    Int32 i32;
    UInt32 ui32;
    Int16 i16;
    UInt16 ui16;
    Float32 f32;
    Float64 f64;
    String s;
    Url u;
} SimpleTypes;
```

The DAS output from the web server should look like this.

```
Attributes {
    Facility {
        String PrincipleInvestigator ‘‘Mark Abbott’’, ‘‘Ph.D’’;
        String DataCenter ‘‘COAS Environmental Computer Facility’’;
        String DrifterType ‘‘MetOcean WOCE/OCM’’;
    }
    b {
        String Description ‘‘A test byte’’;
        String units ‘‘unknown’’;
    }
    i32 {
        String Description ‘‘A 32 bit test server int’’;
        String units ‘‘unknown’’;
    }
}
```

The output from ncdump should look like this.

```
netcdf test {
dimensions:
        stringdim64 = 64 ;
variables:
        byte b ;
                b:Description = "A test byte" ;
                b:units = "unknown" ;
        int i32 ;
                i32:Description = "A 32 bit test server int" ;
                i32:units = "unknown" ;
        int ui32 ;
        short i16 ;
        short ui16 ;
        float f32 ;
        double f64 ;
        char s(stringdim64) ;
        char u(stringdim64) ;
}
```

Note that the fields of type String and type URL have suddenly acquired a dimension. This is because strings are translated to arrays of char, which requires adding an extra dimension. The size of the dimension is determined in a variety of ways and can be specified.

It defaults to 64 and when read, the underlying string is either padded or truncated to that length.

Also note that the `Facility` attributes do not appear in the translation because they are neither global nor associated with a variable in the DDS.

Alternately, one can get the text of the DDS as a global attribute by using the client parameters mechanism . In this case, the parameter "[show=dds]" can be prefixed to the URL and the data retrieved using the following command

```
ncdump -h [show=dds]http://test.opendap.org:8080/dods/dts/test.01.dds
```

The ncdump -h command will then show both the translation and the original DDS. In the above example, the DDS would appear as the global attribute "_DDS" as follows.

```
netcdf test {
...
variables:
        :_DDS = "Dataset { Byte b; Int32 i32; UInt32 ui32; Int16 i16;
                 UInt16 ui16; Float32 f32; Float64 f64;
                 Strings; Url u; } SimpleTypes;"

        byte b ;
...
}
```

4.12.2 DAP to NetCDF Translation Rules

Two translations are currently available.

- DAP 2 Protocol to netCDF-3
- DAP 2 Protocol to netCDF-4

4.12.2.1 netCDF-3 Translation Rules

The current default translation code translates the OPeNDAP protocol to netCDF-3 (classic). This netCDF-3 translation converts an OPeNDAP DAP protocol version 2 DDS to netCDF-3 and is designed to mimic as closely as possible the translation provided by the libnc-dap system. In addition, a translation to netCDF-4 (enhanced) is provided that is entirely new.

For illustrative purposes, the following example will be used.

```
Dataset {
  Int32 f1;
  Structure {
    Int32 f11;
    Structure {
      Int32 f1[3];
      Int32 f2;
    } FS2[2];
  } S1;
  Structure {
    Grid {
      Array:
        Float32 temp[lat=2][lon=2];
      Maps:
        Int32 lat[lat=2];
        Int32 lon[lon=2];
    } G1;
  } S2;
  Grid {
      Array:
        Float32 G2[lat=2][lon=2];
      Maps:
        Int32 lat[2];
        Int32 lon[2];
  } G2;
  Int32 lat[lat=2];
  Int32 lon[lon=2];
} D1;
```

4.12.2.2 Variable Definition

The set of netCDF variables is derived from the fields with primitive base types as they occur in Sequences, Grids, and Structures. The field names are modified to be fully qualified initially. For the above, the set of variables are as follows. The coordinate variables within grids are left out in order to mimic the behavior of libnc-dap.

1. f1
2. S1.f11
3. S1.FS2.f1
4. S1.FS2.f2
5. S2.G1.temp
6. S2.G2.G2
7. lat
8. lon

4.12.2.3 Variable Dimension Translation

A variable's rank is determined from three sources.

1. The variable has the dimensions associated with the field it represents (e.g. S1.FS2.f1[3] in the above example).
2. The variable inherits the dimensions associated with any containing structure that has a rank greater than zero. These dimensions precede those of case 1. Thus, we have in our example, f1[2][3], where the first dimension comes from the containing Structure FS2[2].
3. The variable's set of dimensions are altered if any of its containers is a DAP DDS Sequence. This is discussed more fully below.
4. If the type of the netCDF variable is char, then an extra string dimension is added as the last dimension.

4.12.2.4 Dimension translation

For dimensions, the rules are as follows.

1. Fields in dimensioned structures inherit the dimension of the structure; thus the above list would have the following dimensioned variables.
    - S1.FS2.f1 -> S1.FS2.f1[2][3]
    - S1.FS2.f2 -> S1.FS2.f2[2]
    - S2.G1.temp -> S2.G1.temp[lat=2][lon=2]
    - S2.G1.lat -> S2.G1.lat[lat=2]
    - S2.G1.lon -> S2.G1.lon[lon=2]
    - S2.G2.G2 -> S2.G2.lon[lat=2][lon=2]
    - S2.G2.lat -> S2.G2.lat[lat=2]
    - S2.G2.lon -> S2.G2.lon[lon=2]
    - lat -> lat[lat=2]
    - lon -> lon[lon=2]
2. Collect all of the dimension specifications from the DDS, both named and anonymous (unnamed) For each unique anonymous dimension with value NN create a netCDF dimension of the form "XX_<i>=NN", where XX is the fully qualified name of the variable and i is the i'th (inherited) dimension of the array where the anonymous dimension occurs. For our example, this would create the following dimensions.
    - S1.FS2.f1_0 = 2 ;
    - S1.FS2.f1_1 = 3 ;
    - S1.FS2.f2_0 = 2 ;
    - S2.G2.lat_0 = 2 ;
    - S2.G2.lon_0 = 2 ;
3. If however, the anonymous dimension is the single dimension of a MAP vector in a Grid then the dimension is given the same name as the map vector This leads to the following.
    - S2.G2.lat_0 -> S2.G2.lat
    - S2.G2.lon_0 -> S2.G2.lon
4. For each unique named dimension "<name>=NN", create a netCDF dimension of the form "<name>=NN", where name has the qualifications removed. If this leads to duplicates (i.e. same name and same value), then the duplicates are ignored. This produces the following.
    - S2.G2.lat -> lat
    - S2.G2.lon -> lon

    Note that this produces duplicates that will be ignored later.
5. At this point the only dimensions left to process should be named dimensions with the same name as some dimension from step number 3, but with a different value. For those dimensions create a dimension of the form "<name>M=NN" where M is a counter starting at 1. The example has no instances of this.

6. Finally and if needed, define a single UNLIMITED dimension named "unlimited" with value zero. Unlimited will be used to handle certain kinds of DAP sequences (see below).

This leads to the following set of dimensions.

```
dimensions:
  unlimited = UNLIMITED;
  lat = 2 ;
  lon = 2 ;
  S1.FS2.f1_0 = 2 ;
  S1.FS2.f1_1 = 3 ;
  S1.FS2.f2_0 = 2 ;
```

4.12.2.5 Variable Name Translation

The steps for variable name translation are as follows.

1. Take the set of variables captured above. Thus for the above DDS, the following fields would be collected.
   - f1
   - S1.f11
   - S1.FS2.f1
   - S1.FS2.f2
   - S2.G1.temp
   - S2.G2.G2
   - lat
   - lon
2. All grid array variables are renamed to be the same as the containing grid and the grid prefix is removed. In the above DDS, this results in the following changes.
   1. G1.temp -> G1
   2. G2.G2 -> G2

It is important to note that this process could produce duplicate variables (i.e. with the same name); in that case they are all assumed to have the same content and the duplicates are ignored. If it turns out that the duplicates have different content, then the translation will not detect this. YOU HAVE BEEN WARNED.

The final netCDF-3 schema (minus attributes) is then as follows.

```
netcdf t {
dimensions:
        unlimited = UNLIMITED ;
        lat = 2 ;
        lon = 2 ;
        S1.FS2.f1_0 = 2 ;
        S1.FS2.f1_1 = 3 ;
        S1.FS2.f2_0 = 2 ;
variables:
        int f1 ;
        int lat(lat) ;
        int lon(lon) ;
        int S1.f11 ;
        int S1.FS2.f1(S1.FS2.f1_0, S1.FS2.f1_1) ;
        int S1.FS2.f2(S1_FS2_f2_0) ;
        float S2.G1(lat, lon) ;
        float G2(lat, lon) ;
}
```

In actuality, the unlimited dimension is dropped because it is unused.

There are differences with the original libnc-dap here because libnc-dap technically was incorrect. The original would have said this, for example.

```
int S1.FS2.f1(lat, lat) ;
```

Note that this is incorrect because it dimensions S1.FS2.f1(2,2) rather than S1.FS2.f1(2,3).

4.12.2.6 Translating DAP DDS Sequences

Any variable (as determined above) that is contained directly or indirectly by a Sequence is subject to revision of its rank using the following rules.

1. Let the variable be contained in Sequence Q1, where Q1 is the innermost containing sequence. If Q1 is itself contained (directly or indirectly) in a sequence, or Q1 is contained (again directly or indirectly) in a structure that has rank greater than 0, then the variable will have an initial UNLIMITED dimension. Further, all dimensions coming from "above" and including (in the containment sense) the innermost Sequence, Q1, will be removed and replaced by that single UNLIMITED dimension. The size associated with that UNLIMITED is zero, which means that its contents are inaccessible through the netCDF-3 API. Again, this differs from libnc-dap, which leaves out such variables. Again, however, this difference is backward compatible.
2. If the variable is contained in a single Sequence (i.e. not nested) and all containing structures have rank 0, then the variable will have an initial dimension whose size is the record count for that Sequence. The name of the new dimension will be the name of the Sequence.

Consider this example.

```
Dataset {
  Structure {
    Sequence {
      Int32 f1[3];
      Int32 f2;
    } SQ1;
  } S1[2];
  Sequence {
    Structure {
      Int32 x1[7];
    } S2[5];
  } Q2;
} D;
```

The corresponding netCDF-3 translation is pretty much as follows (the value for dimension Q2 may differ).

```
dimensions:
    unlimited = UNLIMITED ; // (0 currently)
    S1.SQ1.f1_0 = 2 ;
    S1.SQ1.f1_1 = 3 ;
    S1.SQ1.f2_0 = 2 ;
    Q2.S2.x1_0 = 5 ;
    Q2.S2.x1_1 = 7 ;
    Q2 = 5 ;
variables:
    int S1.SQ1.f1(unlimited, S1.SQ1.f1_1) ;
    int S1.SQ1.f2(unlimited) ;
    int Q2.S2.x1(Q2, Q2.S2.x1_0, Q2.S2.x1_1) ;
```

Note that for example S1.SQ1.f1_0 is not actually used because it has been folded into the unlimited dimension.

Note that there is a performance cost because the translation code has to walk the data to determine how many records are associated with the sequence. Since libnc-dap did essentially the same thing, it can be assumed that the cost is not prohibitive.

4.12.2.7 netCDF-4 Translation Rules

A DAP to netCDF-4 translation also exists, but is not the default and in any case is only available if the "–enable-netcdf-4" option is specified at configure time. This translation includes some elements of the libnc-dap translation, but attempts to provide a simpler (but not, unfortunately, simple) set of translation rules than is used for the netCDF-3 translation. Please note that the translation is still experimental and will change to respond to unforeseen problems or to suggested improvements.

This text will use this running example.

```
Dataset {
  Int32 f1[fdim=10];
  Structure {
    Int32 f11;
    Structure {
      Int32 f1[3];
      Int32 f2;
    } FS2[2];
  } S1;
  Grid {
    Array:
      Float32 temp[lat=2][lon=2];
    Maps:
      Int32 lat[2];
      Int32 lon[2];
  } G1;
  Sequence {
    Float64 depth;
  } Q1;
} D
```

4.12.2.8 Variable Definition

The rule for choosing variables is relatively simple. Start with the names of the top-level fields of the DDS. The term top-level means that the object is a direct subnode of the Dataset object. In our example, this produces the set [f1, S1, G1, Q1].

4.12.2.9 Dimension Definition

The rules for choosing and defining dimensions is as follows.

1. Collect the set of dimensions (named and anonymous) directly associated with the variables as defined above. This means that dimensions within user-defined types are ignored. From our example, the dimension set is [fdim=10,lat=2,lon=2,2,2]. Note that the unqualified names are used.
2. All remaining anonymous dimensions are given the name "<var>_NN", where "<var>" is the unqualified name of the variable in which the anonymous dimension appears and NN is the relative position of that dimension in the dimensions associated with that array. No instances of this rule occur in the running example.
3. Remove duplicate dimensions (those with same name and value). Our dimension set now becomes [fdim=10,lat=2,lon=2].
4. The final case occurs when there are dimensions with the same name but with different values. For this case, the size of the dimension is appended to the dimension name.

4.12.2.10 Type Definition

The rules for choosing user-defined types are as follows.

1. For every Structure, Grid, and Sequence, a netCDF-4 compound type is created whose fields are the fields of the Structure, Sequence, or Grid. With one exception, the name of the type is the same as the Structure or Grid name suffixed with "_t". The exception is that the compound types derived from Sequences are instead suffixed with "_record_t".

   The types of the fields are the types of the corresponding field of the Structure, Sequence, or Grid. Note that this type might be itself a user-defined type.

   From the example, we get the following compound types.

   ```
   compound FS2_t {
       int f1(3);
       int f2;
   };
   compound S1_t {
       int f11;
       FS2_t FS2(2);
   };
   compound G1_t {
       float temp(2,2);
   ```

```
        int lat(2);
        int lon(2);
    }
    compound Q1_record_t {
        double depth;
    };
```

2. For all sequences of name X, also create this type.

   ```
   X_record_t (*) X_t
   ```

   In our example, this produces the following type.

   ```
   Q1_record_t (*) Q1_t
   ```

3. If a Sequence, Q has a single field F, whose type is a primitive type, T, (e.g., int, float, string), then do not apply the previous rule, but instead replace the whole sequence with the the following field.

   ```
   T (*) Q.f
   ```

4.12.2.11 Choosing a Translation

The decision about whether to translate to netCDF-3 or netCDF-4 is determined by applying the following rules in order.

1. If the NC_CLASSIC_MODEL flag is set on nc_open(), then netCDF-3 translation is used.
2. If the NC_NETCDF4 flag is set on nc_open(), then netCDF-4 translation is used.
3. If the URL is prefixed with the client parameter "[netcdf3]" or "[netcdf-3]" then netCF-3 translation is used.
4. If the URL is prefixed with the client parameter "[netcdf4]" or "[netcdf-4]" then netCF-4 translation is used.
5. If none of the above holds, then default to netCDF-3 classic translation.

4.12.2.12 Caching

In an effort to provide better performance for some access patterns, client-side caching of data is available. The default is no caching, but it may be enabled by prefixing the URL with "[cache]".

Caching operates basically as follows.

1. When a URL is first accessed using nc_open(), netCDF automatically does a pre-fetch of selected variables. These include all variables smaller than a specified (and user definable) size. This allows, for example, quick access to coordinate variables.
2. Whenever a request is made using some variant of the nc_get_var() API procedures, the complete variable is fetched and stored in the cache as a new cache entry. Subsequence requests for any part of that variable will access the cache entry to obtain the data.
3. The cache may become too full, either because there are too many entries or because it is taking up too much disk space. In this case cache entries are purged until the cache size limits are reached. The cache purge algorithm is LRU (least recently used) so that variables that are repeatedly referenced will tend to stay in the cache.
4. The cache is completely purged when nc_close() is invoked.

In order to decide if you should enable caching, you will need to have some understanding of the access patterns of your program.

- The ncdump program always dumps one or more whole variables so it turns on caching.
- If your program accesses only parts of a number of variables, then caching should probably not be used since fetching whole variables will probably slow down your program for no purpose.

Unfortunately, caching is currently an all or nothing proposition, so for more complex access patterns, the decision to cache or not may not have an obvious answer. Probably a good rule of thumb is to avoid caching initially and later turn it on to see its effect on performance.

4.12.2.13 Defined Client Parameters

Currently, a limited set of client parameters is recognized. Parameters not listed here are ignored, but no error is signalled.

Parameter Name Legal Values Semantics

[netcdf-3]|[netcdf-3]

Specify translation to netCDF-3.

[netcdf-4]|[netcdf-4]

Specify translation to netCDF-4.

"*[log]|[log=<file>]*" ""

Turn on logging and send the log output to the specified file. If no file is specified, then output to standard error.

"*[show=...]*" *das|dds|url*

This causes information to appear as specific global attributes. The tags may be combined using comma with no spaces (e.g. "show=dds,url"). The currently recognized tags are "dds" to display the underlying DDS, "das" similarly, and "url" to display the url used to retrieve the data.

"*[show=fetch]*"

This parameter causes the netCDF code to log a copy of the complete url for every HTTP get request. If logging is enabled, then this can be helpful in checking to see the access behavior of the netCDF code.

"*[stringlength=NN]*"

Specify the default string length to use for string dimensions. The default is 64.

"*[stringlength_<var>=NN]*"

Specify the default string length to use for a string dimension for the specified variable. The default is 64.

"*[cache]*" This enables caching.

"*[cachelimit=NN]*"

Specify the maximum amount of space allowed for the cache.

"*[cachecount=NN]*"

Specify the maximum number of entries in the cache.

4.12.3 Notes on Debugging OPeNDAP Access

The OPeNDAP support makes use of the logging facility of the underlying oc system. Note that this is currently separate from the existing netCDF logging facility. Turning on this logging can sometimes give important information. Logging can be enabled by prefixing the url with the client parameter [log] or [log=filename], where the first case will send log output to standard error and the second will send log output to the specified file.

Users should also be aware that the DAP subsystem creates temporary files of the name dataddsXXXXXX, where XXXXX is some random string. If the program using the DAP subsystem crashes, these files may be left around. It is perfectly safe to delete them. Also, if you are accessing data over an NFS mount, you may see some .nfsxxxxx files; those can be ignored as well.

5 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 three 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/ncgen4`	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
`nccopy`	reads a netCDF dataset using the netCDF programming interface and copies it, optionally to a different kind of 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/netcdf/software.html`.

This chapter describes the ncgen, ncgen4, and ncdump utilities. These three 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).

5.1 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 for a classic model file consists of three optional parts: dimensions, variables, and data. The variable part may contain variable declarations and attribute assignments. For the enhanced model supported by netCDF-4, a CDL decription may also includes groups, subgroups, and user-defined types.

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 classic 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). Any number of dimensions can be declared of unlimited length in CDL for an enhanced model 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 or containing group. 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 one-dimensional arrays of values. For example, "units" might be an attribute represented by a string 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 or metadata (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, if not explicitly specified, 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. In the netCDF-4 enhanced model, attributes may be declared to be of user-defined type, like variables.

In CDL, just as for netCDF, the names of dimensions, variables and attributes (and, in netCDF-4 files, groups, user-defined types, compound member names, and enumeration symbols) consist of arbitrary sequences of alphanumeric characters, underscore '_', period '.', plus '+', hyphen '-', or at sign '@', but beginning with a letter or underscore. However names commencing with underscore are reserved for system use. Case is significant in netCDF names. A zero-length name is not allowed. Some widely used conventions restrict names to only alphanumeric characters or underscores. Names that have trailing space characters are also not permitted.

Beginning with versions 3.6.3 and 4.0, names may also include UTF-8 encoded Unicode characters as well as other special characters, except for the character '/', which may not appear in a name (because it is reserved for path names of nested groups). In CDL, most special characters are escaped with a backslash '\' character, but that character is not actually part of the netCDF name. The special characters that do not need to be escaped in CDL names are underscore '_', period '.', plus '+', hyphen '-', or at sign '@'. For the formal specification of CDL name syntax See Section 1.3 [Format], page 6. Note that by using special characters in names, you may make your data not compliant with conventions that have more stringent requirements on valid names for netCDF components, for example the CF Conventions.

The names for the primitive data types are reserved words in CDL, so names of variables, dimensions, and attributes must not be primitive 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 multidimensional 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 among primitive types are supported.

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

5.2 CDL Data Types

The CDL primitive data types for the classic model are:

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

NetCDF-4 supports the additional primitive types:

`ubyte`	Unsigned eight-bit integers.

`ushort`	Unsigned 16-bit integers.
`uint`	Unsigned 32-bit integers.
`int64`	64-bit singed integers.
`uint64`	Unsigned 64-bit singed integers.
`string`	Variable-length string of characters

Except for the added data-type byte, CDL supports the same primitive data types as C. For backward compatibility, in declarations primitive 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 numeric data, and the zero byte has no special significance, as it may for character data. The short type holds values between -32768 and 32767. The ushort type holds values between 0 and 65536. The int type can hold values between -2147483648 and 2147483647. The uint type holds values between 0 and 4294967296. The int64 type can hold values between -9223372036854775808 and 9223372036854775807. The uint64 type can hold values between 0 and 18446744073709551616.

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 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 string type holds variable length strings.

5.3 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 length and type (if primitive and not explicitly declared). CDL defines a syntax for constant values that permits distinguishing among different netCDF primitive types. The syntax for CDL constants is similar to C syntax, with type suffixes appended to bytes, 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
```

5.4 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.

The ncgen tool is now is capable of producing netcdf-4 files. It operates essentially identically to the original ncgen.

The CDL input to ncgen may include data model constructs from the netcdf- data model. In particular, it includes new primitive types such as unsigned integers and strings, opaque data, enumerations, and user-defined constructs using vlen and compound types. The ncgen man page should be consulted for more detailed information.

UNIX syntax for invoking ncgen:

```
ncgen [-b] [-o netcdf-file] [-c] [-f] [-k<kind>] [-l<language>] [-x] [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. The -c flag is deprecated and the -lc flag should be used intstead.

`-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. The -f flag is deprecated and the -lf77 flag should be used intstead.

`-k` The -k file specifies the kind of netCDF file to generate. The arguments to the -k flag can be as follows.

- 1, classic – Produce a netcdf classic file format file."
- 2, 64-bit-offset, '64-bit offset' – Produce a netcdf 64 bit classic file format file.
- 3, hdf5, netCDF-4, enhanced – Produce a netcdf-4 format file.
- 4, hdf5-nc3, 'netCDF-4 classic model', enhanced-nc3 – Produce a netcdf-4 file format, but restricted to netcdf-3 classic CDL intput.

Note that the -v flag is a deprecated alias for -k.

`-l` The -l file specifies that ncgen should output (to standard output) the text of a program that, when compiled and executed, will produce the corresponding binary .nc file. The arguments to the -l flag can be as follows.

- c|C => C language output.
- f77|fortran77 => FORTRAN 77 language output; note that currently only the classic model is supported for fortran output.
- cml|CML => (experimental) NcML language output

- j|java => (experimental) Java language output; the generated java code targets the existing Unidata Java interface, which means that only the classic model is supported.

`-x` Use "no fill" mode, omitting the initialization of variable values with fill values. This can make the creation of large files much faster, but it will also eliminate the possibility of detecting the inadvertent reading of values that haven't been written.

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
```

5.5 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.

As of NetCDF version 4.1, ncdump can also access DAP data sources if DAP support is enabled in the underlying NetCDF library. Instead of specifying a file name as argument to ncdump, the user specifies a URL to a DAP source.

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]] [ -s ] [ -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.

`-s` Specifies that special virtual attributes should be output for the file format variant and for variable properties such as compression, chunking, and other properties specific to the format implementation that are primarily related to performance rather than the logical schema of the data. All the special virtual attributes begin with '_' followed by an upper-case letter. Currently they include the global attribute "_Format" and the variable attributes "_Fletcher32", "_ChunkSizes", "_Endianness", "_DeflateLevel", "_Shuffle", "_Storage", and "_NoFill". The ncgen utility recognizes these attributes and supports them appropriately.

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
```

Examine the translated DDS for the DAP source from the specified URL.

```
ncdump -h http://test.opendap.org:8080/dods/dts/test.01
```

5.6 ncgen3

The ncgen3 tool is the new name for the older, original ncgen utility.

The ncgen3 tool generates a netCDF file or a C or FORTRAN program that creates a netCDF dataset. If no options are specified in invoking ncgen3, the program merely checks the syntax of the CDL input, producing error messages for any violations of CDL syntax.

The ncgen3 utility can only generate classic-model netCDF-4 files or programs.

UNIX syntax for invoking ncgen3:

```
ncgen3 [-b] [-o netcdf-file] [-c] [-f] [-v2|-v3] [-x] [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.

`-v2` The generated netCDF file or program will use the version of the format with 64-bit offsets, to allow for the creation of very large files. These files are not as portable as classic format netCDF files, because they require version 3.6.0 or later of the netCDF library.

`-v3` The generated netCDF file will be in netCDF-4/HDF5 format. These files are not as portable as classic format netCDF files, because they require version 4.0 or later of the netCDF library.

`-x` Use “no fill” mode, omitting the initialization of variable values with fill values. This can make the creation of large files much faster, but it will also eliminate the possibility of detecting the inadvertent reading of values that haven't been written.

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 counter-intuitive. 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 its distribution.

Appendix B 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. For more information see Appendix A [Units], page 69.

`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.

`_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 nc_set_fill in C (see Section "nc_set_fill" in *The NetCDF C Interface Guide*) or NF_SET_FILL in Fortran (see Section "NF_SET_FILL" in *The NetCDF Fortran 77 Interface Guide*). 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. If the variable is packed using scale_factor and add_offset attributes (see below), the _FillValue attribute should have the data type of the packed data.

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.

For more information for C programmers see Section "Fill Values" in *The NetCDF C Interface Guide*. For more information for Fortran programmers see Section "Fill Values" in *The NetCDF Fortran 77 Interface Guide*.

`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.

When scale_factor and add_offset are used for packing, the value(s) of the missing_value attribute should be specified in the domain of the data in the file (the packed data), so that missing values can be detected before the scale_factor and add_offset are applied.

`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.

If the variable is packed using scale_factor and add_offset attributes (see below), the _FillValue, missing_value, valid_range, valid_min, or valid_max attributes should have the data type of the packed data.

`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.

If valid values are specified using the valid_min, valid_max, valid_range, or _FillValue attributes, those values should be specified in the domain of the data in the file (the packed data), so that they can be interpreted before the scale_factor and add_offset are applied.

`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, rounding the result to the nearest integer to avoid a bias caused by truncation towards zero.

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.

`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 non-negative values, you can use valid_min = 0 and valid_max = 255. This attribute is ignored by the netCDF library.

`C_format` A character array providing the format that should be used by C 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 C_format attribute as "%.3g". The ncdump utility program uses this attribute for variables for which it is defined. The format applies to the scaled (internal) type and value, regardless of the presence of the scaling attributes scale_factor and add_offset.

`FORTRAN_format`
A character array providing the format that should be used by FORTRAN 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. Originally, these conventions were named by a string that was interpreted as a directory name relative to the directory /pub/netcdf/Conventions/ on the host ftp.unidata.ucar.edu. The web page http://www.unidata.ucar.edu/netcdf/conventions.html is now the preferred and authoritative location for registering a URI reference to a set of conventions maintained elsewhere. The FTP site will be preserved for compatibility with existing references, but authors of new conventions should submit a request to support-netcdf@unidata.ucar.edu for listing on the Unidata conventions web page.

It may be convenient for defining institutions and groups to use a hierarchical structure for general conventions and more specialized conventions. 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.

It is possible for a netCDF file to adhere to more than one set of conventions, even when there is no inheritance relationship among the conventions. In this case, the value of the 'Conventions' attribute may be a single text string containing a list of the convention names separated by blank space (recommended) or commas (if a convention name contains blanks).

Typical conventions web sites will include references to documents in some form agreed upon by the community that supports the conventions and examples of netCDF file structures that follow the conventions.

Appendix C File Format Specification

In different contexts, "netCDF" may refer to an abstract data model, a software implementation with associated application program interfaces (APIs), or a data format. Confusion may easily arise in discussions of different versions of the data models, software, and formats, because the relationships among versions of these entities is more complex than a simple one-to-one correspondence by version. For example, compatibility commitments require that new versions of the software support all previous variants of the format and data model.

To avoid this potential confusion, we assign distinct names to versions of the formats, data models, and software releases that will be used consistently in the remainder of this appendix.

In this appendix, two format variants are specified formally, the *classic format* and the *64-bit offset format* for netCDF data. Two additional format variants are discussed less formally, the *netCDF-4 format* and the *netCDF-4 classic model format*.

The classic format was the only format for netCDF data created between 1989 and 2004 by various versions of the reference software from Unidata. In 2004, the 64-bit offset format variant was introduced for creation of and access to much larger files. The reference software, available for C-based and Java-based programs, supported use of the same APIs for accessing either classic or 64-bit offset files, so programs reading the files would not have to depend on which format was used.

There are only two netCDF data models, the *classic model* and the *enhanced model*. The classic model is the simpler of the two, and is used for all data stored in classic format, 64-bit offset format, or netCDF-4 classic model format. The enhanced model (also referred to as the netCDF- 4 data model) was introduced in 2008 as an extension of the classic model that adds more powerful forms of data representation and data types at the expense of some additional complexity. Although data represented with the classic model can also be represented using the enhanced model, datasets that use features of the enhanced model, such as user-defined nested data types, cannot be represented with the classic model. Use of added features of the enhanced model requires that data be stored in the netCDF-4 format.

Versions 1.0 through 3.5 of the Unidata C-based reference software, released between 1989 and 2000, supported only the classic data model and classic format. Version 3.6, released in late 2004, first provided support for the 64-bit offset format, but still used the classic data model. With version 4.0, released in 2008, the enhanced data model was introduced along with the two new HDF5-based format variants, the netCDF-4 format and the netCDF-4 classic model format. Evolution of the data models, formats, and APIs will continue the commitment to support all previous netCDF data models, data format variants, and APIs in future software releases.

Use of the HDF5 storage layer in netCDF-4 software provides features for improved performance, independent of the data model used, for example compression and dynamic schema changes. Such performance improvements are available for data stored in the netCDF-4 classic model format, even when accessed by programs that only support the classic model.

Related formats not discussed in this appendix include CDL ("Common Data Language", the original ASCII form of binary netCDF data), and NcML (NetCDF Markup Language, an XML-based representation for netCDF metadata and data).

Knowledge of format details is not required to read or write netCDF datasets. Software that reads netCDF data using the reference implementation automatically detects and uses the correct version of the format for accessing data. Understanding details may be helpful for understanding performance issues related to disk or server access.

The netCDF reference library, developed and supported by Unidata, is written in C, with Fortran77, Fortran90, and C++ interfaces. A number of community and commercially supported interfaces to other languages are also available, including IDL, Matlab, Perl, Python, and Ruby. An independent implementation, also developed and supported by Unidata, is written entirely in Java.

C.1 The NetCDF Classic Format Specification

To present the format more formally, we use a BNF grammar notation. In this notation:

- Non-terminals (entities defined by grammar rules) are in lower case.
- Terminals (atomic entities in terms of which the format specification is written) are in upper case, and are specified literally as US-ASCII characters within single-quote characters or are described with text between angle brackets (‘`<`’ and ‘`>`’).
- Optional entities are enclosed between braces (‘`[`’ and ‘`]`’).
- A sequence of zero or more occurrences of an entity is denoted by ‘`[entity ...]`’.
- A vertical line character (‘`|`’) separates alternatives. Alternation has lower precedence than concatenation.
- Comments follow ‘`//`’ characters.
- A single byte that is not a printable character is denoted using a hexadecimal number with the notation ‘`\xDD`’, where each D is a hexadecimal digit.
- A literal single-quote character is denoted by ‘`\'`’, and a literal back-slash character is denoted by ‘`\\`’.

Following the grammar, a few additional notes are included to specify format characteristics that are impractical to capture in a BNF grammar, and to note some special cases for implementers. Comments in the grammar point to the notes and special cases, and help to clarify the intent of elements of the format.

The Format in Detail

```
netcdf_file  = header  data
header       = magic  numrecs  dim_list  gatt_list  var_list
magic        = 'C'  'D'  'F'  VERSION
VERSION      = \x01 |                      // classic format
               \x02                        // 64-bit offset format
numrecs      = NON_NEG | STREAMING         // length of record dimension
dim_list     = ABSENT | NC_DIMENSION  nelems  [dim ...]
gatt_list    = att_list                    // global attributes
att_list     = ABSENT | NC_ATTRIBUTE  nelems  [attr ...]
var_list     = ABSENT | NC_VARIABLE   nelems  [var ...]
ABSENT       = ZERO  ZERO                  // Means list is not present
ZERO         = \x00 \x00 \x00 \x00         // 32-bit zero
NC_DIMENSION = \x00 \x00 \x00 \x0A         // tag for list of dimensions
```

```
NC_VARIABLE  = \x00 \x00 \x00 \x0B         // tag for list of variables
NC_ATTRIBUTE = \x00 \x00 \x00 \x0C         // tag for list of attributes
nelems       = NON_NEG       // number of elements in following sequence
dim          = name  dim_length
name         = nelems  namestring
                    // Names a dimension, variable, or attribute.
                    // Names should match the regular expression
                    // ([a-zA-Z0-9_]|{MUTF8})([^\x00-\x1F/\x7F-\xFF]|{MUTF8})*
                    // For other constraints, see "Note on names", below.
namestring   = ID1 [IDN ...] padding
ID1          = alphanumeric | ’_’
IDN          = alphanumeric | special1 | special2
alphanumeric = lowercase | uppercase | numeric | MUTF8
lowercase    = ’a’|’b’|’c’|’d’|’e’|’f’|’g’|’h’|’i’|’j’|’k’|’l’|’m’|
               ’n’|’o’|’p’|’q’|’r’|’s’|’t’|’u’|’v’|’w’|’x’|’y’|’z’
uppercase    = ’A’|’B’|’C’|’D’|’E’|’F’|’G’|’H’|’I’|’J’|’K’|’L’|’M’|
               ’N’|’O’|’P’|’Q’|’R’|’S’|’T’|’U’|’V’|’W’|’X’|’Y’|’Z’
numeric      = ’0’|’1’|’2’|’3’|’4’|’5’|’6’|’7’|’8’|’9’
                             // special1 chars have traditionally been
                             // permitted in netCDF names.
special1     = ’_’|’.’|’@’|’+’|’-’
                             // special2 chars are recently permitted in
                             // names (and require escaping in CDL).
                             // Note: ’/’ is not permitted.
special2     = ’ ’ | ’!’ | ’"’ | ’#’  | ’$’ | ’%’ | ’&’ | ’\’’ |
               ’(’ | ’)’ | ’*’ | ’,’  | ’:’ | ’;’ | ’<’ | ’=’  |
               ’>’ | ’?’ | ’[’ | ’\\’ | ’]’ | ’^’ | ’‘’ | ’{’  |
               ’|’ | ’}’ | ’~’
MUTF8        = <multibyte UTF-8 encoded, NFC-normalized Unicode character>
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_list  nc_type  vsize  begin
                             // nelems is the dimensionality (rank) of the
                             // variable: 0 for scalar, 1 for vector, 2
                             // for matrix, ...
dimid        = NON_NEG       // Dimension ID (index into dim_list) for
                             // variable shape.  We say this is a "record
                             // variable" if and only if the first
                             // dimension is the record dimension.
vatt_list    = att_list      // Variable-specific attributes
```

```
vsize        = NON_NEG       // Variable size.  If not a record variable,
                             // the amount of space in bytes allocated to
                             // the variable's data.  If a record variable,
                             // the amount of space per record.  See "Note
                             // on vsize", below.
begin        = OFFSET        // 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     = [vardata ...] // The data for all non-record variables,
                             // stored contiguously for each variable, in
                             // the same order the variables occur in the
                             // header.
vardata      = [values ...]  // All data for a non-record variable, as a
                             // block of values of the same type as the
                             // variable, in row-major order (last
                             // dimension varying fastest).
recs         = [record ...]  // The data for all record variables are
                             // stored interleaved at the end of the
                             // file.
record       = [varslab ...] // Each record consists of the n-th slab
                             // from each record variable, for example
                             // x[n,...], y[n,...], z[n,...] where the
                             // first index is the record number, which
                             // is the unlimited dimension index.
varslab      = [values ...]  // One record of data for a variable, a
                             // block of values all of the same type as
                             // the variable in row-major order (last
                             // index varying fastest).
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>
                             // Header padding uses null (\x00) bytes.  In
                             // data, padding uses variable's fill value.
                             // See "Note on padding", below, for a special
                             // case.
NON_NEG      = <non-negative INT>
STREAMING    = \xFF \xFF \xFF \xFF   // Indicates indeterminate record
                                     // count, allows streaming data
OFFSET       = <non-negative INT> |  // For classic format or
               <non-negative INT64>  // for 64-bit offset format
```

```
BYTE         = <8-bit byte>          // See "Note on byte data", below.
CHAR         = <8-bit byte>          // See "Note on char data", below.
SHORT        = <16-bit signed integer, Bigendian, two’s complement>
INT          = <32-bit signed integer, Bigendian, two’s complement>
INT64        = <64-bit signed integer, Bigendian, two’s complement>
FLOAT        = <32-bit IEEE single-precision float, Bigendian>
DOUBLE       = <64-bit IEEE double-precision float, Bigendian>
                             // following type tags are 32-bit integers
NC_BYTE      = \x00 \x00 \x00 \x01       // 8-bit signed integers
NC_CHAR      = \x00 \x00 \x00 \x02       // text characters
NC_SHORT     = \x00 \x00 \x00 \x03       // 16-bit signed integers
NC_INT       = \x00 \x00 \x00 \x04       // 32-bit signed integers
NC_FLOAT     = \x00 \x00 \x00 \x05       // IEEE single precision floats
NC_DOUBLE    = \x00 \x00 \x00 \x06       // IEEE double precision floats
                             // Default fill values for each type, may be
                             // overridden by variable attribute named
                             // ‘_FillValue’. See "Note on fill values",
                             // below.
FILL_CHAR    = \x00                      // null byte
FILL_BYTE    = \x81                      // (signed char) -127
FILL_SHORT   = \x80 \x01                 // (short) -32767
FILL_INT     = \x80 \x00 \x00 \x01       // (int) -2147483647
FILL_FLOAT   = \x7C \xF0 \x00 \x00       // (float) 9.9692099683868690e+36
FILL_DOUBLE  = \x47 \x9E \x00 \x00 \x00 \x00 //(double)9.9692099683868690e+36
```

Note on vsize: This number is the product of the dimension lengths (omitting the record dimension) and the number of bytes per value (determined from the type), increased to the next multiple of 4, for each variable. If a record variable, this is the amount of space per record. The netCDF “record size” is calculated as the sum of the vsize’s of all the record variables.

The vsize field is actually redundant, because its value may be computed from other information in the header. The 32-bit vsize field is not large enough to contain the size of variables that require more than $2^{3}2 - 4$ bytes, so $2^{3}2 - 1$ is used in the vsize field for such variables.

Note on names: Earlier versions of the netCDF C-library reference implementation enforced a more restricted set of characters in creating new names, but permitted reading names containing arbitrary bytes. This specification extends the permitted characters in names to include multi-byte UTF-8 encoded Unicode and additional printing characters from the US-ASCII alphabet. The first character of a name must be alphanumeric, a multi-byte UTF-8 character, or ’_’ (reserved for special names with meaning to implementations, such as the “_FillValue” attribute). Subsequent characters may also include printing special characters, except for ’/’ which is not allowed in names. Names that have trailing space characters are also not permitted.

Implementations of the netCDF classic and 64-bit offset format must ensure that names are normalized according to Unicode NFC normalization rules during encoding as UTF-8 for storing in the file header. This is necessary to ensure that gratuitous differences in the representation of Unicode names do not cause anomalies in comparing files and querying data objects by name.

Note on streaming data: The largest possible record count, $2^{3}2-1$, is reserved to indicate an indeterminate number of records. This means that the number of records in the file must be determined by other means, such as reading them or computing the current number of records from the file length and other information in the header. It also means that the numrecs field in the header will not be updated as records are added to the file. [This feature is not yet implemented].

Note on padding: In the special case of only a single record variable of character, byte, or short type, no padding is used between data values.

Note on byte data: It is possible to interpret byte data as either signed (-128 to 127) or unsigned (0 to 255). When reading byte data through an interface that converts it into another numeric type, the default interpretation is signed. There are various attribute conventions for specifying whether bytes represent signed or unsigned data, but no standard convention has been established. The variable attribute "_Unsigned" is reserved for this purpose in future implementations.

Note on char data: Although the characters used in netCDF names must be encoded as UTF-8, character data may use other encodings. The variable attribute "_Encoding" is reserved for this purpose in future implementations.

Note on fill values: Because data variables may be created before their values are written, and because values need not be written sequentially in a netCDF file, default "fill values" are defined for each type, for initializing data values before they are explicitly written. This makes it possible to detect reading values that were never written. The variable attribute "_FillValue", if present, overrides the default fill value for a variable. If _FillValue is defined then it should be scalar and of the same type as the variable.

Fill values are not required, however, because netCDF libraries have traditionally supported a "no fill" mode when writing, omitting the initialization of variable values with fill values. This makes the creation of large files faster, but also eliminates the possibility of detecting the inadvertent reading of values that haven't been written.

Notes on Computing File Offsets

The offset (position within the file) of a specified data value in a classic format or 64-bit offset data file is completely determined by the variable start location (the offset in the `begin` field), the external type of the variable (the `nc_type` field), and the dimension indices (one for each of the variable's dimensions) of the value desired.

The external size in bytes of one data value for each possible netCDF type, denoted `extsize` below, is:

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

The record size, denoted by `recsize` below, is the sum of the `vsize` fields of record variables (variables that use the unlimited dimension), using the actual value determined by dimension sizes and variable type in case the `vsize` field is too small for the variable size.

To compute the offset of a value relative to the beginning of a variable, it is helpful to precompute a "product vector" from the dimension lengths. 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 as the product vector for each variable.

For example:

Non-record variable:

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

Record variable:

dimension lengths: [0 2 9 4] product vector: [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 the array of coordinates (dimension indices, zero-based) of the desired data value. Then the offset of the value from the beginning of the file 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.

A special case: Where there is exactly one record variable, we drop the requirement 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. It begins with the four-byte “magic number” that identifies it as a netCDF version 1 file: ‘`C`’, ‘`D`’, ‘`F`’, ‘`\x01`’. Following are seven 32-bit integer zeros representing the number of records, an empty list of dimensions, an empty list of global attributes, and an empty list of variables.

Below is an (edited) dump of the file produced 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 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 ]
```

C.2 The 64-bit Offset Format

The netCDF 64-bit offset format differs from the classic format only in the VERSION byte, '`\x02`' instead of '`\x01`', and the OFFSET entity, a 64-bit instead of a 32-bit offset from the beginning of the file. This small format change permits much larger files, but there are still some practical size restrictions. Each fixed-size variable and the data for one record's worth of each record variable are still limited in size to a little less that 4 GiB. The rationale for this limitation is to permit aggregate access to all the data in a netCDF variable (or a record's worth of data) on 32-bit platforms.

C.3 The NetCDF-4 Format

The netCDF-4 format implements and expands the netCDF-3 data model by using an enhanced version of HDF5 as the storage layer. Use is made of features that are only available in HDF5 version 1.8 and later.

Using HDF5 as the underlying storage layer, netCDF-4 files remove many of the restrictions for classic and 64-bit offset files. The richer enhanced model supports user-defined types and data structures, hierarchical scoping of names using groups, additional primitive types including strings, larger variable sizes, and multiple unlimited dimensions. The underlying HDF5 storage layer also supports per-variable compression, multidimensional tiling, and efficient dynamic schema changes, so that data need not be copied when adding new variables to the file schema.

Creating a netCDF-4/HDF5 file with netCDF-4 results in an HDF5 file. The features of netCDF-4 are a subset of the features of HDF5, so the resulting file can be used by any existing HDF5 application.

Although every file in netCDF-4 format is an HDF5 file, there are HDF5 files that are not netCDF-4 format files, because the netCDF-4 format intentionally uses a limited subset of the HDF5 data model and file format features. Some HDF5 features not supported in the netCDF enhanced model and netCDF-4 format include non-hierarchical group structures, HDF5 reference types, multiple links to a data object, user-defined atomic data types, stored property lists, more permissive rules for data object names, the HDF5 date/time type, and attributes associated with user-defined types.

A complete specification of HDF5 files is beyond the scope of this document. For more information about HDF5, see the HDF5 web site: `http://hdf.ncsa.uiuc.edu/HDF5/`.

The specification that follows is sufficient to allow HDF5 users to create files that will be accessable from netCDF-4.

C.3.1 Creation Order

The netCDF API maintains the creation order of objects that are created in the file. The same is not true in HDF5, which maintains the objects in alphabetical order. Starting in version 1.8 of HDF5, the ability to maintain creation order was added. This must be explicitly turned on in the HDF5 data file in several ways.

Each group must have link and attribute creation order set. The following code (from libsrc4/nc4hdf.c) shows how the netCDF-4 library sets these when creating a group.

```
/* Create group, with link_creation_order set in the group
 * creation property list. */
if ((gcpl_id = H5Pcreate(H5P_GROUP_CREATE)) < 0)
   return NC_EHDFERR;
if (H5Pset_link_creation_order(gcpl_id, H5P_CRT_ORDER_TRACKED|H5P_CRT_ORDER_INDE
   BAIL(NC_EHDFERR);
if (H5Pset_attr_creation_order(gcpl_id, H5P_CRT_ORDER_TRACKED|H5P_CRT_ORDER_INDE
   BAIL(NC_EHDFERR);
if ((grp->hdf_grpid = H5Gcreate2(grp->parent->hdf_grpid, grp->name,
                                 H5P_DEFAULT, gcpl_id, H5P_DEFAULT)) < 0)
   BAIL(NC_EHDFERR);
if (H5Pclose(gcpl_id) < 0)
   BAIL(NC_EHDFERR);
```

Each dataset in the HDF5 file must be created with a property list for which the attribute creation order has been set to creation ordering. The H5Pset_attr_creation_order funtion is used to set the creation ordering of attributes of a variable.

The following example code (from libsrc4/nc4hdf.c) shows how the creation ordering is turned on by the netCDF library.

```
      /* Turn on creation order tracking. */
      if (H5Pset_attr_creation_order(plistid, H5P_CRT_ORDER_TRACKED|
                                     H5P_CRT_ORDER_INDEXED) < 0)
         BAIL(NC_EHDFERR);
```

C.3.2 Groups

NetCDF-4 groups are the same as HDF5 groups, but groups in a netCDF-4 file must be strictly hierarchical. In general, HDF5 permits non-hierarchical structuring of groups (for example, a group that is its own grandparent). These non-hierarchical relationships are not allowed in netCDF-4 files.

In the netCDF API, the global attribute becomes a group-level attribute. That is, each group may have its own global attributes.

The root group of a file is named "/" in the netCDF API, where names of groups are used. It should be noted that the netCDF API (like the HDF5 API) makes little use of names, and refers to entities by number.

C.3.3 Dimensions with HDF5 Dimension Scales

Until version 1.8, HDF5 did not have any capability to represent shared dimensions. With the 1.8 release, HDF5 introduced the dimension scale feature to allow shared dimensions in HDF5 files.

The dimension scale is unfortunately not exactly equivilent to the netCDF shared dimension, and this leads to a number of compromises in the design of netCDF-4.

A netCDF shared dimension consists solely of a length and a name. An HDF5 dimension scale also includes values for each point along the dimension, information that is (optionally) included in a netCDF coordinate variable.

To handle the case of a netCDF dimension without a coordinate variable, netCDF-4 creates dimension scales of type char, and leaves the contents of the dimension scale empty. Only the name and length of the scale are significant. To distinguish this case, netCDF-4 takes advantage of the NAME attribute of the dimension scale. (Not to be confused with the name of the scale itself.) In the case of dimensions without coordinate data, the HDF5 dimension scale NAME attribute is set to the string: `"This is a netCDF dimension but not a netCDF variable."`

In the case where a coordinate variable is defined for a dimension, the HDF5 dimscale matches the type of the netCDF coordinate variable, and contains the coordinate data.

A further difficulty arrises when an n-dimensional coordinate variable is defined, where n is greater than one. NetCDF allows such coordinate variables, but the HDF5 model does not allow dimension scales to be attached to other dimension scales, making it impossible to completely represent the multi-dimensional coordinate variables of the netCDF model.

To capture this information, multidimensional coordinate variables have an attribute named _Netcdf4Coordinates. The attribute is an array of H5T_NATIVE_INT, with the netCDF dimension IDs of each of its dimensions.

The _Netcdf4Coordinates attribute is otherwise hidden by the netCDF API. It does not appear as one of the attributes for the netCDF variable involved, except through the HDF5 API.

C.3.4 Dimensions without HDF5 Dimension Scales

Starting with the netCDF-4.1 release, netCDF can read HDF5 files which do not use dimension scales. In this case the netCDF library assigns dimensions to the HDF5 dataset as needed, based on the length of the dimension.

When an HDF5 file is opened, each dataset is examined in turn. The lengths of all the dimensions involved in the shape of the dataset are determined. Each new (i.e. previously unencountered) length results in the creation of a phony dimension in the netCDF API.

This will not accurately detect a shared, unlimited dimension in the HDF5 file, if different datasets have different lengths along this dimension (possible in HDF5, but not in netCDF).

Note that this is a read-only capability for the netCDF library. When the netCDF library writes HDF5 files, they always use a dimension scale for every dimension.

Datasets must have either dimension scales for every dimension, or no dimension scales at all. Partial dimension scales are not, at this time, understood by the netCDF library.

C.3.5 Dimension and Coordinate Variable Ordering

In order to preserve creation order, the netCDF-4 library writes variables in their creation order. Since some variables are also dimension scales, their order reflects both the order of the dimensions and the order of the coordinate variables.

However, these may be different. Consider the following code:

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

/* Define dimensions in order. */
if (nc_def_dim(ncid, DIM0, NC_UNLIMITED, &dimids[0])) ERR;
if (nc_def_dim(ncid, DIM1, 4, &dimids[1])) ERR;

/* Define coordinate variables in a different order. */
if (nc_def_var(ncid, DIM1, NC_DOUBLE, 1, &dimids[1], &varid[1])) ERR;
if (nc_def_var(ncid, DIM0, NC_DOUBLE, 1, &dimids[0], &varid[0])) ERR;
```

In this case the order of the coordinate variables will be different from the order of the dimensions.

In practice, this should make little difference in user code, but if the user is writing code that depends on the ordering of dimensions, the netCDF library was updated in version 4.1 to detect this condition, and add the attribute _Netcdf4Dimid to the dimension scales in the HDF5 file. This attribute holds a scalar H5T_NATIVE_INT which is the (zero-based) dimension ID for this dimension.

If this attribute is present on any dimension scale, it must be present on all dimension scales in the file.

C.3.6 Variables

Variables in netCDF-4/HDF5 files exactly correspond to HDF5 datasets. The data types match naturally between netCDF and HDF5.

In netCDF classic format, the problem of endianness is solved by writing all data in big-endian order. The HDF5 library allows data to be written as either big or little endian, and automatically reorders the data when it is read, if necessary.

By default, netCDF uses the native types on the machine which writes the data. Users may change the endianness of a variable (before any data are written). In that case the specified endian type will be used in HDF5 (for example, a H5T_STD_I16LE will be used for NC_SHORT, if little-endian has been specified for that variable.)

`NC_BYTE`	H5T_NATIVE_SCHAR
`NC_UBYTE`	H5T_NATIVE_SCHAR
`NC_CHAR`	H5T_C_S1
`NC_STRING`	variable length array of H5T_C_S1
`NC_SHORT`	H5T_NATIVE_SHORT
`NC_USHORT`	H5T_NATIVE_USHORT
`NC_INT`	H5T_NATIVE_INT
`NC_UINT`	H5T_NATIVE_UINT
`NC_INT64`	H5T_NATIVE_LLONG
`NC_UINT64`	H5T_NATIVE_ULLONG
`NC_FLOAT`	H5T_NATIVE_FLOAT
`NC_DOUBLE`	H5T_NATIVE_DOUBLE

The NC_CHAR type represents a single character, and the NC_STRING an array of characters. This can be confusing because a one-dimensional array of NC_CHAR is used to represent a string (i.e. a scalar NC_STRING).

An odd case may arise in which the user defines a variable with the same name as a dimension, but which is not intended to be the coordinate variable for that dimension. In this case the string "_nc4_non_coord_" is pre-pended to the name of the HDF5 dataset, and stripped from the name for the netCDF API.

C.3.7 Attributes

Attributes in HDF5 and NetCDF-4 correspond very closely. Each attribute in an HDF5 file is represented as an attribute in the netCDF-4 file, with the exception of the attributes below, which are ignored by the netCDF-4 API.

`_Netcdf4Coordinates`	An integer array containing the dimension IDs of a variable which is a multi-dimensional coordinate variable.
`_nc3_strict`	When this (scalar, H5T_NATIVE_INT) attribute exists in the root group of the HDF5 file, the netCDF API will enforce the netCDF classic model on the data file.
`REFERENCE_LIST`	This attribute is created and maintained by the HDF5 dimension scale API.
`CLASS`	This attribute is created and maintained by the HDF5 dimension scale API.
`DIMENSION_LIST`	This attribute is created and maintained by the HDF5 dimension scale API.
`NAME`	This attribute is created and maintained by the HDF5 dimension scale API.

C.3.8 User-Defined Data Types

Each user-defined data type in an HDF5 file exactly corresponds to a user-defined data type in the netCDF-4 file. Only base data types which correspond to netCDF-4 data types may be used. (For example, no HDF5 reference data types may be used.)

C.3.9 Compression

The HDF5 library provides data compression using the zlib library and the szlib library. NetCDF-4 only allows users to create data with the zlib library (due to licensing restrictions on the szlib library). Since HDF5 supports the transparent reading of the data with either compression filter, the netCDF-4 library can read data compressed with szlib (if the underlying HDF5 library is built to support szlib), but has no way to write data with szlib compression.

With zlib compression (a.k.a. deflation) the user may set a deflation factor from 0 to 9. In our measurements the zero deflation level does not compress the data, but does incur the performance penalty of compressing the data. The netCDF API does not allow the user to write a variable with zlib deflation of 0 - when asked to do so, it turns off deflation for the variable instead. NetCDF can read an HDF5 file with deflation of zero, and correctly report that to the user.

C.4 The NetCDF-4 Classic Model Format

Every classic and 64-bit offset file can be represented as a netCDF-4 file, with no loss of information. There are some significant benefits to using the simpler netCDF classic model with the netCDF-4 file format. For example, software that writes or reads classic model data can write or read netCDF-4 classic model format data by recompiling/relinking to a netCDF-4 API library, with no or only trivial changes needed to the program source code. The netCDF-4 classic model format supports this usage by enforcing rules on what functions may be called to store data in the file, to make sure its data can be read by older netCDF applications (when relinked to a netCDF-4 library).

Writing data in this format prevents use of enhanced model features such as groups, added primitive types not available in the classic model, and user-defined types. However performance features of the netCDF-4 formats that do not require additional features of the enhanced model, such as per-variable compression and chunking, efficient dynamic schema changes, and larger variable size limits, offer potentially significant performance improvements to readers of data stored in this format, without requiring program changes.

When a file is created via the netCDF API with a CLASSIC_MODEL mode flag, the library creates an attribute (_nc3_strict) in the root group. This attribute is hidden by the netCDF API, but is read when the file is later opened, and used to ensure that no enhanced model features are written to the file.

C.5 HDF4 SD Format

Starting with version 4.1, the netCDF libraries can read HDF4 SD (Scientific Dataset) files. Access is limited to those HDF4 files created with the Scientific Dataset API. Access is read-only.

Dataset types are translated between HDF4 and netCDF in a straighforward manner.

`DFNT_CHAR`	NC_CHAR
`DFNT_UCHAR, DFNT_UINT8`	NC_UBYTE
`DFNT_INT8`	NC_BYTE
`DFNT_INT16`	NC_SHORT
`DFNT_UINT16`	NC_USHORT
`DFNT_INT32`	NC_INT
`DFNT_UINT32`	NC_UINT
`DFNT_FLOAT32`	NC_FLOAT
`DFNT_FLOAT64`	NC_DOUBLE

Appendix D Internal Dispatch Table

Draft 3: 5/15/2010

netCDF Dispatch Mechanism

This document describes the architecture and details of the new netCDF internal dispatch mechanism. The idea is that when a user opens or creates a netcdf file, that a specific dispatch table is chosen. Subsequent netcdf API calls are then channeled through that dispatch table to the appropriate function for implementing that API call.

Currently, the following four dispatch tables are supported.

1. netcdf classic files (netcdf-3)
2. netcdf enhanced files (netcdf-4)
3. OPeNDAP to netcdf-3
4. OPeNDAP to netcdf-4

The dispatch table represents a distillation of the netcdf API down to a minimal set of internal operations. The format of the dispatch table is defined in the file libdispatch/dispatch.h. Every new dispatch table must define this minimal set of operations.

Adding a New Dispatch Table

In order to make this process concrete, let us assume we plan to add an in-memory implementation of netcdf-3.

Step 1.

Define a –enable flag and an AM_CONFIGURE flag in configure.ac. We will use the flags –enable-netcdfm and USE_NETCDFM respectively.

Step 2

Choose some prefix of characters to identify the new dispatch system. In effect we are defining a name-space. For our in-memory system, we will choose "NCM" and "ncm". NCM is used for non-static procedures to be entered into the dispatch table and ncm for all other non-static procedures.

Step 3.

Modify file libdispatch/dispatch.h as follows.

- Add a index for this implementation:

```
#define NC_DISPATCH_NCM  5
```

- Define an external reference to the in-memory dispatch table.

```
#ifdef USE_NETCDFM
extern NC_Dispatch* NCM_dispatch_table;
#endif
```

Step 4.

Modify file libdispatch/netcdf.c as follows.

- Add a ptr to the in-memory dispatch table.

```
#ifdef USE_NETCDFM
NC_Dispatch* NCM_dispatch_table = NULL;
#endif
```

- Add any necessary #include files as needed.

Step 5.

Define the functions necessary to fill in the dispatch table. As a rule, we assume that a new directory is defined, libsrcm, say. Within this directory, we need to define Makefile.am, the source files containing the dispatch table and the functions to be placed in the dispatch table – call them ncmdispatch.c and ncmdispatch.h. Look at libsrc/nc3dispatch.[ch] for an example.

As part of the ncmdispatch.c file, you must define the following.

```
NC_Dispatch NCM_dispatcher = {
NC_DISPATCH_NCM,
NCM_create,
NCM_open,
...
};

int
NCM_initialize(void)
{
    NCM_dispatch_table = &NCM_dispatcher;
    return NC_NOERR;
}
```

Assuming that the in-memory library does not require any external libraries, then the Makefile.am will look something like this.

```
if USE_DISPATCH
NCM_SOURCES = ncmdispatch.c ncmdispatch.h ...
AM_CPPFLAGS +=  -I$(top_srcdir)/libsrc -I$(top_srcdir)/libdispatch
libnetcdfm_la_SOURCES = $(NCM_SOURCES)
noinst_LTLIBRARIES = libnetcdfm.la
endif #USE_DISPATCH
```

Step 6.

Provide for the inclusion of this library in the final libnetcdf library. This is accomplished by modifying liblib/Makefile.am by adding something like the following.

```
if USE_NETCDFM
   libnetcdf_la_LIBADD += $(top_builddir)/libsrcm/libnetcdfm.la
endif
```

Step 7.

Modify the NC_intialize function in liblib/stub.c by adding appropriate references to the NCM dispatch function.

```
#ifdef USE_NETCDFM
extern int NCM_initialize(void);
#endif
...
int NC_initialize(void)
{
...
#ifdef USE_DAP
    if((stat = NCM_initialize())) return stat;
#endif
...
}
```

Step 8.

Add a directory of tests; ncm_test, say. The file ncm_test/Makefile.am will look something like this.

```
# These files are created by the tests.
CLEANFILES = ...
# These are the tests which are always run.
TESTPROGRAMS = test1 test2 ...
test1_SOURCES = test1.c ...
...
# Set up the tests.
check_PROGRAMS = $(TESTPROGRAMS)
TESTS = $(TESTPROGRAMS)
# Any extra files required by the tests
EXTRA_DIST = ...
```

Step 9.

Provide for libnetcdfm to be constructed by adding the following to the top-level Makefile.am.

```
if USE_NETCDFM
NCM=libsrcm
NCMTESTDIR=ncm_test
endif
...
SUBDIRS = ... $(DISPATCHDIR)  $(NCM) ... $(NCMTESTDIR)
```

Choosing a Dispatch Table

The dispatch table is chosen in the NC_create and the NC_open procedures in libdispatch/netcdf.c. The decision is currently based on the following pieces of information.

- The file path – this can be used to detect, for example, a DAP url versus a normal file system file.
- The mode argument – this can be used to detect, for example, what kind of file to create: netcdf-3, netcdf-4, 64-bit netcdf-3, etc.
- For nc_open and when the file path references a real file, the contents of the file can also be used to determine the dispatch table.
- Although currently not used, this code could be modified to also use other pieces of information such as environment variables.

In addition to the above, there is one additional mechanism to force the use of a specific dispatch table. The procedure "NC_set_dispatch_override()" can be invoked to specify a dispatch table.

When adding a new dispatcher, it is necessary to modify NC_create and NC_open in libdispatch/netcdf.c to detect when it is appropriate to use the NCM dispatcher. Some possibilities are as follows.

1. Add a new mode flag: say NC_NETCDFM.
2. Use an environment variable.
3. Define a special file path format that indicates the need to use a special dispatch table.

Special Dispatch Table Signatures.

Several of the entries in the dispatch table are significantly different than those of the external API.

Create/Open

The create table entry and the open table entry have the following signatures respectively.

```
int (*create)(const char *path, int cmode,
          size_t initialsz, int basepe, size_t *chunksizehintp,
          int useparallel, MPI_Comm comm, MPI_Info info,
          struct NC_Dispatch*, struct NC** ncp);

int (*open)(const char *path, int mode,
        int basepe, size_t *chunksizehintp,
        int use_parallel, MPI_Comm comm, MPI_Info info,
        NC_Dispatch*, NC** ncp);
```

The key difference is that these are the union of all the possible create/open signatures from the netcdf.h API. Note especially the last two parameters. The dispatch table is included in case the create function (e.g. NCM_create) needs to invoke other dispatch functions. The very last parameter is a pointer to a pointer to an NC instance. It is expected that the create function will allocate and fill in an instance of an "NC" object and return a pointer to it in the ncp parameter.

Notes:

- As with the existing code, and when MPI is not being used, the comm and info parameters should be passed in as 0. This is taken care of in the nc_open and nc_create API procedures in libdispatch/netcdf.c.

- In fact, the object returned in the ncp parameter does not actually have to be an instance of struct NC. It only needs to "look like it for the first few fields. This is, in effect, a fake version of subclassing. Let us suppose that the NCM_create function uses a struct NCM object. The initial part of the definition of NCM must match the fields at the beginning of struct NC between the comments BEGIN_COMMON and END_COMMON. So, we would have the following.

  ```
  typedef struct NCM {
  /*BEGIN COMMON*/
          int ext_ncid; /* uid  16 */
          int int_ncid; /* unspecified other id */
          struct NC_Dispatch* dispatch;
  #ifdef USE_DAP
          struct NCDRNO* drno;
  #endif
  /*END COMMON*/
  ...
  } NCM;
  ```

  This allows the pointer to the NCM object to be cast as an instance of NC* and its pointer returned in the ncp file. Eventually, this will be replaced with a separate structure containing the common fields.

put_vara/get_vara

```
int (*put_vara)(int ncid, int varid, const size_t *start, const size_t *count,
                     const void *value, nc_type memtype);

int (*get_vara)(int ncid, int varid, const size_t *start, const size_t *count,
                void *value, nc_type memtype);
```

Most of the parameters are similar to the netcdf API parameters. The last parameter, however, is the type of the data in memory. Additionally, instead of using an "int islong" parameter, the memtype will be either NC_INT or NC_INT64, depending on the value of sizeof(long). This means that even netcdf-3 code must be prepared to encounter the NC_INT64 type.

put_attr/get_attr

```
int (*get_att)(int ncid, int varid, const char *name,
                    void *value, nc_type memtype);

int (*put_att)(int ncid, int varid, const char *name, nc_type datatype, size_t len,
               const void *value, nc_type memtype);
```

Again, the key difference is the memtype parameter. As with put/get_vara, it used NC_INT64 to encode the long case.

NetCDF Library Assembly

The assembly of the final libnetcdf library occurs in the directory liblib. The Makefile uses all of the available configuration flags to decide which component libraries will be added to libnetcdf to produce the final library. In addition, the proper version of netcdf.h will have been placed in liblib: either the version from libsrc or the version from libsrc4 depending on the USE_NETCDF4 flag.

Utility Construction

All of the utilities and the test directories (nctest, nc_test, ...) are expected to obtain their libnetcdf library and their netcdf.h from the ones in liblib.

Miscellaneous Notes

1. It may be desirable to include a few test cases in the libsrcm directory. Libsrc4, for example, has quite a number of such tests. In order to do this, it is necessary to create a number of stub definitions so that the library will compile and load with the test cases. The file libsrc/stub3.c shows a typical stub file.
2. In order avoid adding massive numbers of #ifdef USE_DISPATCH conditionals, I have duplicated certain files in libsrc, libsrc4, libncdap3 and libncdap4. The files for use with dispatch are prefixed with `"d_"`. Thus for libsrc4/nc4file.c, there is a corresponding libsrc4/d_nc4file.c If you make changes to one file, you should see if they need to also be made to the other file.

Index