CDF
User's Guide

Version 2.2.0*, March 4, 1992

National Space Science Data Center

National Space Science Data Center
NASA - Goddard Space Flight Center
Greenbelt, Maryland 20771 (USA)

NSINET — NSSDCA::CDFSUPPORT
Internet — CDFSUPPORT@NSSDCA.GSFC.NASA.GOV

Contents

List of Figures

List of Tables

Preface

About this document

This document is intended to be both a user's guide and reference manual for the Common Data Format (CDF). As such, it provides a primer for introducing the novice reader to the concepts of CDF as well as a reference manual for the advanced user and programmer. However, it does not serve as a cookbook for the proper methods of architecturing a CDF.

The very first questions usually asked by a reader are: What is CDF?, How is CDF used?, and How is CDF useful for me?. Although the reader will find the answers to these questions in this document, we provide here a brief description of the conceptual basis of CDF in order to provide a proper perspective when reading the remainder of this document.

What is CDF?

CDF, in its most basic terms, is a conceptual data abstraction for storing, manipulating, and accessing multi-dimensional data sets. We refer to CDF as a data abstraction because we never discuss the actual physical format in which data sets are stored. Instead, we describe the form of the data sets and the means (interface) by which they may be manipulated. This important difference from traditional physical file formats is reflected in the orientation of the document towards defining form and function as opposed to a specification of the bits and bytes in an actual physical format. It is important to state here that the use of a data abstraction in no way inhibits access to physical data or necessarily makes such access inefficient. It merely provides a way of generalizing the data model and makes possible the specification of a uniform interface for manipulating it. The data abstraction allows future extensibility and provides for conceptual simplicity while isolating machine and device dependence.

The contents of a CDF fall into two catagories. The first is a series of records comprising a collection of variables consisting of scalars, vectors, and n-dimensional arrays. The second is a set of attribute entires (metadata) describing the CDF in global terms or specifically for a single variable. This dual function of CDF is what provides it's "data set independence". Both the data dictionary (attributes) and the data objects (variables) are combined into an integrated data set. An important element of the CDF conceptual data abstraction is the "virtual" dimensional layer that allows data objects that share a subset of the overall CDF dimensionality to be projected into the full dimensional space. This capability is made available through the use of logical dimensional variances which indicate the subset of CDF dimensions that are applicable. A few simple examples of this...

1. Making time ordered measurements on a grid of data. Since the grid is fixed, it needs to be stored only once while still being projected at different time instances.
2. A two dimensional spectrogram with nonlinear axes. In this case the two one-dimensional vectors

describing the nonlinear gridding can be expressed as projections of the two-dimensional spectrogram. This allows easier intercomparison of data objects within the CDF while maintaining efficient storage of the actual data.

How is CDF used?

The origins of CDF date back to the development of the NASA Climate Data System at the National Space Science Data Center (NSSDC). As such, it has had three main requirements driving it's development.

1. Facilitate ingestion of data sets and data products into CDF.
2. Utilize standard common terminology (metadata) to describe the data sets.
3. Development of higher level applications (eg. NSSDC Graphics System, NGS).

The above requirements imply two classes of users for CDF. One user class performs primarily data acquisition and is mainly involved in architecturing CDF's and the associated science metadata. The other user class builds high-level applications interacting with CDF at the programming level. CDF has two levels of access: one is through the programming interface layer and the other is through a high-level toolkit written using the programming interface layer.

The toolkit provides utilities for ingesting data into CDF and for browsing existing CDF data sets. These are very useful for architecturing a CDF and describing the metadata without using the programming level interface. The browsing tools allow a quick look at CDF data sets and aid in CDF validation.

The programming layer (CDF library) provides the essential framework on which graphical and data analysis packages can be created. The CDF library allows developers of CDF-based systems to easily create applications which permit users to slice data across multi-dimensional subspaces, access entire structures of data, perform sub-sampling of data, and access one data element independently regardless of its relationship to any other data element. CDF data sets are portable across any platform supported by CDF. These currently consist of: VAX (VMS), Sun (SunOS), DECstation (ULTRIX), Silicon Graphics Iris and Power Series (IRIX), IBM RS/6000 series (AIX), HP 9000 series (HP-UX), and IBM PC (MS-DOS).

How is CDF useful to me?

Hopefully, the answers to the first two questions have provided a basis for answering this question. It is important to understand that CDF has been designed to solve a number of data storage problems and has shown itself to be quite flexible in storing a wide variety of data sets.

Chapter 1

CDF Primer

1.1 Introduction

The CDF Primer is designed for scientists, researchers, programmers, and managers who want to learn about CDF without reading through this entire document. The Primer will address what CDF is and how it can be used for storing and managing different types of data. A brief description of the tools and utilities available with CDF, in addition to program and toolkit examples, will be given. More detailed descriptions of the concepts presented herein are provided in the accompanying chapters of this document.

1.2 Why Use CDF?

When someone first hears the term CDF they intuitively think of data formats in the traditional sense of the word (ie. messy/convoluted storage of data on disk or tape). CDF is more than just a format. CDF is a "self-describing" format for managing data. In addition to the actual data being stored, CDF also stores user-supplied descriptions of the data, known as metadata. This self-describing property allows CDF to be a generic, data independent format that can store data from a wide variety of disciplines.

In addition to being a self-describing data format, CDF is also a software library. The library routines are callable from C and Fortran and allow the user to randomly access and manage data and metadata without regard to their physical storage. This completely relieves the user of low-level I/O operations allowing more time for data analysis. The actual format used to store the data and metadata is completely transparent to the user.

The term "CDF" is also used to refer to the physical files which the CDF Library generates.

1.3 Conceptual Organization

An important feature of CDF is that it can handle data sets which are inherently multi-dimensional in addition to data sets which are scalar. To do this, CDF groups data by "variables" whose values are conceptually organized into arrays. The dimensionality of these variable arrays depends upon the data and is specified by the user when the CDF is created. For scalar data, as an example, the array of values would degenerate to a single value, whereas for image data the array would be two-dimensional. Similarly, the array for volume data would be three-dimensional. CDF allows users to specify arrays of up to ten dimensions. The array for a particular variable is called a "variable record". A collection of arrays for each variable is referred to as a "CDF record". A CDF can, and quite often does, contain multiple CDF records. This is useful for data where there are repeated observations at different times, altitudes, or pressure levels for example. In its current implementation, every variable array in a CDF must have the same number of dimensions and dimension sizes.[1] Because of this, we say that the CDF itself has a certain number of dimensions and dimension sizes (its dimensionality). The number of dimensions in a CDF is referred to as its rank. In the scalar data example the CDF rank would be zero, whereas for the image data example the CDF rank would be two (see Figures 1.1 and 1.2).

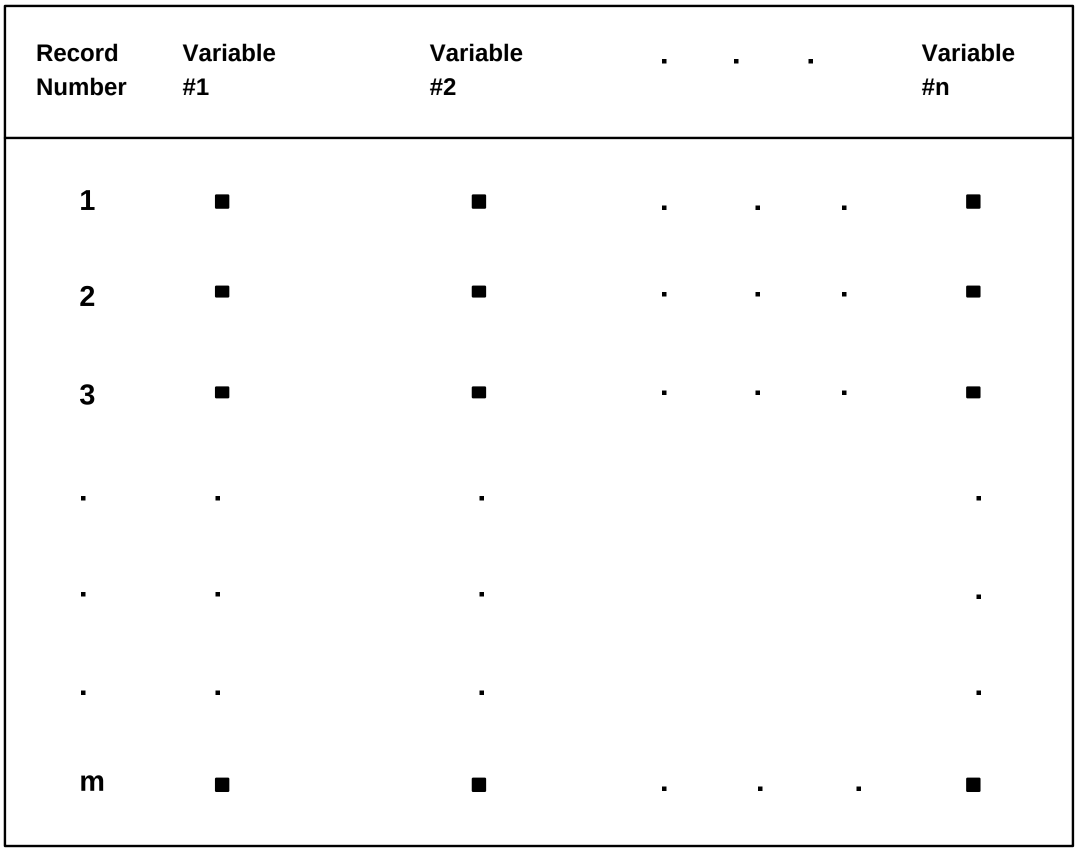

Figure 1.1: Conceptual View of a CDF with Rank = 0

It is important to note that there is no single "correct" way to store data in a CDF. The user has complete control over how the data is stored in the CDF (within the confines of the variable array structure) depending on how the user views the data. This is the advantage of CDF. Data is organized in whatever way makes sense to the user.

[1] This will change in a future release of CDF.

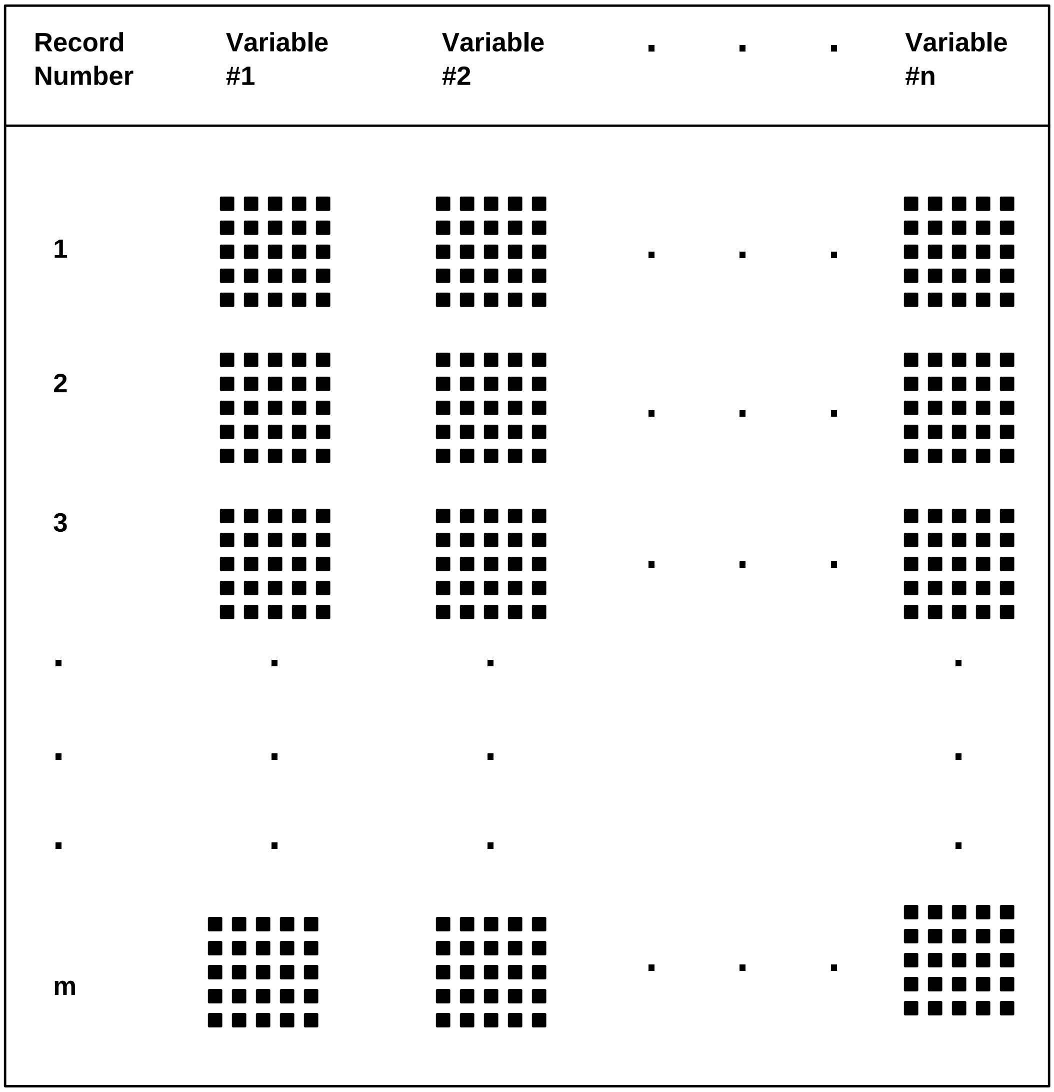

Figure 1.2: Conceptual View of a CDF with Rank = 2

1.4 Features of the CDF Library

The CDF library is a flexible and extensible software package which give the user many options for creating and accessing a CDF.

1.4.1 File Format Option

The CDF library gives the user the option to choose from one of two file formats in which to store the data and metadata. The first option is the traditional CDF multi-file format which is also the default. This file format is illustrated in Figure 1.3 (assuming a CDF containing four variables). The `example.cdf` file contains all of the control information and metadata for the CDF. In addition to the `.cdf` file, a file exists for each variable in the CDF and contains only the data associated with that variable. This is illustrated by the files `example.v0` through `example.v3`. The second option is the single-file format which, as illustrated in Figure 1.4, consists of only the `example.cdf` file. This file contains the control information, metadata, and the data values for each of the variables in the CDF. Both formats allow direct access. The advantage of the single-file format is that it minimizes the number of files one has to manage and makes it easier to transport CDFs across a network. The organization of the data within the single file may, however, become somewhat convoluted, slightly increasing the data access time. The multi-file format, on the other hand, clearly delimits the data from the metadata and is organized in a consistent fashion within the files. Updating, appending, and accessing data are done with optimum efficiency.

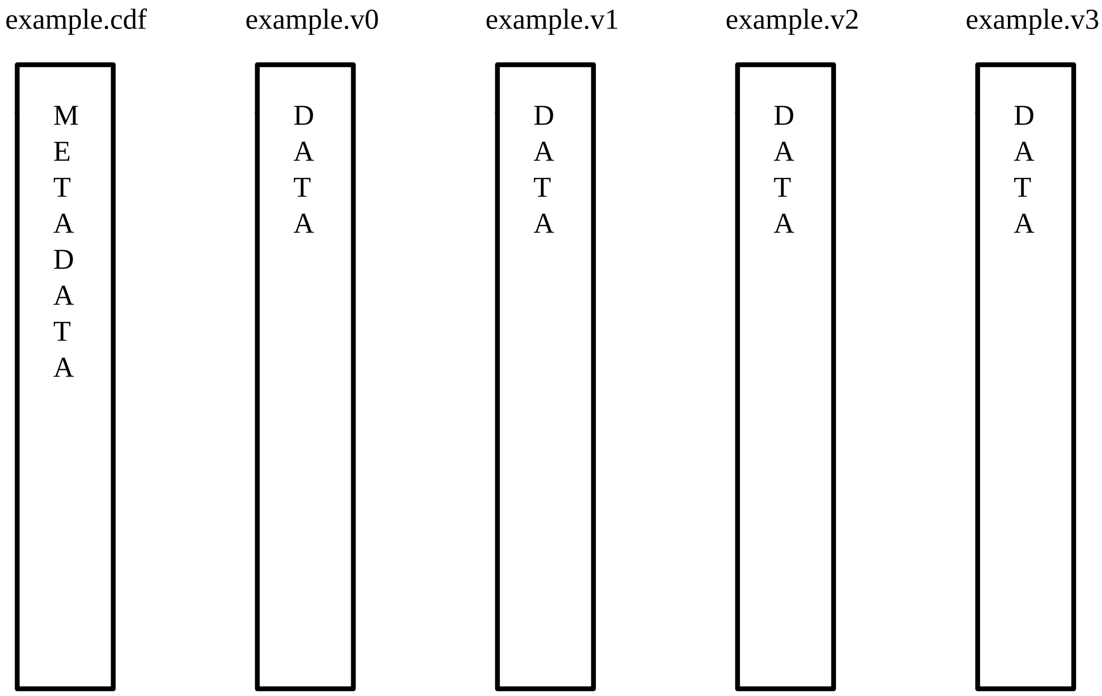

Figure 1.3: Multi-File Format

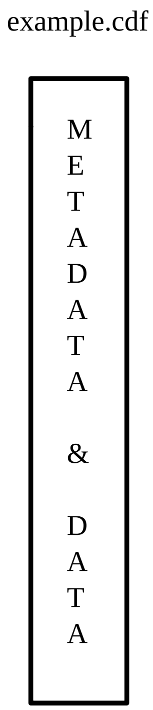

Figure 1.4: Single-File Format

1.4.2 Data Encoding Option

When creating a CDF, the programmer has the option to choose from one of two data encoding schemes. The first is the host encoding scheme which is also the default. Using the host encoding scheme, the data in the CDF will be stored in the physical representation used by the machine on which the CDF library is running (VAX, Sun, SGi Personal Iris and Power Series, DECstation, IBM-RS/6000 series, HP 9000 series, or IBM PC). The second encoding scheme is network encoding. When using the network encoding scheme, the data values are transparently converted from the host representation to the XDR data representation when writing the data and from the XDR representation to the host representation when reading the data. This gives the user the ability to transfer the CDFs to other platforms on which the CDF library runs and to read the data without having to be concerned with the data conversion.

1.4.3 Data Access Option

A program can access variable data one value at a time or it can access an entire multi-dimensional array structure or sub-structure spanning contiguous or non-contiguous record boundaries. The latter feature allows the user to perform aggregate access or uniform sub-sampling of the data at greatly increased rates over value by value access.

1.4.4 CDF Library Interface Option

The CDF library supports two programming interfaces, the Standard interface and the Internal interface. The Standard interface is similar to the interface provided with Version 1 of CDF (which is now known as the Obsolete interface) with several additions for new features. The Internal interface is provided to allow additional functionality to be added to the CDF library without the need to modify the Standard interface. Those features not available using the Standard interface are made avaiable using the Internal interface. The Internal interface makes CDF an easily extensible software package.

Both the Standard and Internal interfaces are callable from both C and FORTAN (the Obsolete interface is callable only from FORTRAN).

1.5 Organizing Your Data in a CDF

1.5.1 Variables

The first component of a CDF is the actual data, organized into arrays for the individual variables. CDF can accommodate any type of data which can be organized into arrays. An example of the type of data set which may be stored in CDF is shown in Table 1.1. Each record holds one value for each of the four variables: `Time`, `Longitude`, `Latitude`, and `Temperature`. CDF can store scalar data in a "flat" (rank = 0) representation such as this, but storage in this manner may hide fundamental relationships among the data values. Consistent repetitions found in the data for this example suggest another way to organize the data set. Note that records one and five are observations of the same point on earth at two different times. That fact is not immediately clear from this representation of the data. Looking more closely, we note that only two differing values are recorded for `Longitude` and, similarly, only two differing values are recorded for `Latitude`. This repetition suggests a two-dimensional array structure whose dimensions are defined by `Longitude` and `Latitude`. For each of the two `Longitude` values there are two `Latitude` values. `Time` repeats for each `Longitude/Latitude` pair — the observations were taken simultaneously at the longitude/latitude locations. Because of `Time`'s repetition for `Longitude/Latitude` pairs, the number of `Time` values specifies the number of records needed in the CDF. Each record contains a two-dimensional array per variable (Table 1.2). The array structure defines the dimensionality of the CDF. Although there are four variables, the array dimensions and the sizes of those dimensions are determined only by `Longitude` and `Latitude`. `Temperature` varies across the entire array while `Time` tells us how many records to expect. Therefore, the example, when reduced as described, defines a two-dimensional CDF. The number of discrete values for each variable which defines a dimension generates the size of that dimension. For example, `Longitude` has two unique values so the dimension defined by `Longitude` has a size of two.

Adding another independent variable, for instance `Pressure`, poses no difficulty for the example. `Temperature` would then be dependent on a specific `Longitude`, `Latitude`, and `Pressure` — a three-dimensional array structure. In this three-dimensional example, the `Longitude`, `Latitude`, and `Pressure` variables define the number of dimensions in the CDF, where the size of each dimension is determined by the number of discrete values contained in each of those variables. Additional dependent variables would be stored in the same way as `Temperature`.

Although conceptually there is a two-dimensional array structure for each variable in each record of the CDF, this would not be an efficient way to store the data. For instance, the time for each record need only be stored once as opposed to being stored four times as shown in each two-dimensional array (Table 1.2). This problem is circumvented by specifying "variances". For each variable there are variances associated with the array

Record Number	Variables Time	Longitude	Latitude	Temperature
1	0000	-165.	+40.	20.0
2	0000	-165.	+30.	21.7
3	0000	-150.	+40.	19.2
4	0000	-150.	+30.	20.7
5	0100	-165.	+40.	18.2
6	0100	-165.	+30.	19.3
7	0100	-150.	+40.	22.0
8	0100	-150.	+30.	19.2
9	0200	-165.	+40.	19.9
10	0200	-165.	+30.	19.3
11	0200	-150.	+40.	19.6
12	0200	-150.	+30.	19.0
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
93	2300	-165.	+40.	21.0
94	2300	-165.	+30.	19.5
95	2300	-150.	+40.	18.4
96	2300	-150.	+30.	22.0

Table 1.1: Example Data Set - "Flat" Representation (Rank = 0)

<table>
<tr><th>Record Number</th><th>Variables Time</th><th>Longitude</th><th>Latitude</th><th>Temperature</th></tr>
<tr><td></td><td>0000 — 0000</td><td>-165 — -150</td><td>+40 — +40</td><td>20.0 — 19.2</td></tr>
<tr><td>1</td><td>| |</td><td>| |</td><td>| |</td><td>| |</td></tr>
<tr><td></td><td>0000 — 0000</td><td>-165 — -150</td><td>+30 — +30</td><td>21.7 — 20.7</td></tr>
<tr><td></td><td>0100 — 0100</td><td>-165 — -150</td><td>+40 — +40</td><td>18.2 — 22.0</td></tr>
<tr><td>2</td><td>| |</td><td>| |</td><td>| |</td><td>| |</td></tr>
<tr><td></td><td>0100 — 0100</td><td>-165 — -150</td><td>+30 — +30</td><td>19.3 — 19.2</td></tr>
<tr><td></td><td>0200 — 0200</td><td>-165 — -150</td><td>+40 — +40</td><td>19.9 — 19.6</td></tr>
<tr><td>3</td><td>| |</td><td>| |</td><td>| |</td><td>| |</td></tr>
<tr><td></td><td>0200 — 0200</td><td>-165 — -150</td><td>+30 — +30</td><td>19.3 — 19.0</td></tr>
<tr><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td></td><td>2300 — 2300</td><td>-165 — -150</td><td>+40 — +40</td><td>21.0 — 18.4</td></tr>
<tr><td>24</td><td>| |</td><td>| |</td><td>| |</td><td>| |</td></tr>
<tr><td></td><td>2300 — 2300</td><td>-165 — -150</td><td>+30 — +30</td><td>19.5 — 22.0</td></tr>
</table>

Table 1.2: Example CDF - Two-Dimensional Representation (Conceptual)

dimensions as well as the records. "Record variance" indicates whether or not a variable has unique values from record to record in the CDF. `Time` changes for each record so the record variance for `Time` is [TRUE]. One could also say that `Time` is record-variant. `Latitude` and `Longitude` repeat their values from record to record so the record variance for each is [FALSE]. `Latitude` and `Longitude` are non-record-variant (NRV). The `Temperature` values change from record to record so they are record-variant. The record variances for this example are shown in Table 1.3.

Similarly, the term "dimension variance" indicates whether or not a variable changes with respect to the CDF dimensions. In the two-dimensional CDF example above, the `Longitude` variable defines the first dimension of the CDF with its values repeating along the second dimension so its dimension variances would be [TRUE,false]. The `Latitude` variable defines the second dimension of the CDF with its values repeating along the first dimension so its dimension variances would be [false,TRUE]. Because the `Temperature` values change for each latitude/longitude location, its dimension variances are [TRUE,TRUE]. `Time` does not change from one latitude/longitude location to another so its values are the same along both dimensions. The dimension variances for `Time` would be [false,false]. The dimension variances for the above example are shown in Table 1.3.

	Variables			
	`Time`	`Longitude`	`Latitude`	`Temperature`
Record Variance	TRUE	false	false	TRUE
First Dimension Variance	false	TRUE	false	TRUE
Second Dimension Variance	false	false	TRUE	TRUE

Table 1.3: Example CDF - Specification for Two-Dimensional Representation

When the record and dimension variances have been defined appropriately, the amount of physical storage needed for the CDF is drastically reduced. In the above example, two-dimensional arrays are not physically stored for each variable in a CDF record. Instead, the physical storage for each variable consists of just one value for `Time` in each CDF record, a single one-dimensional array of values for the `Longitude` and `Latitude` variables (in only the first CDF record), and a full two-dimensional array of values for `Temperature` in each CDF record. The actual physical storage is shown in Table 1.4. The conceptual view of the CDF, however, is still that of one two-dimensional array per variable in each CDF record as shown in Table 1.2 (the physically stored values are shown in boldface type).

1.6 Attributes

The second component of a CDF is the metadata. Metadata consists of user-supplied descriptive information about the CDF and the variables in the CDF and are referred to as attributes. Attributes can be divided into two categories: attributes of global scope and attributes of variable scope. Global scope attributes describe the CDF as a whole, while variable scope attributes describe some property of each variable in the CDF. Any number of attributes may be stored in a single CDF. Section 2.9 describes standard CDF attributes used by the NSSDC.

Global scope attributes can include any information regarding the CDF and all of its variables collectively. Such descriptions could include a title for the CDF, data set documentation, or a CDF modification history. Global scope attributes may contain multiple entries. An example of this would be a modification history kept in the optional global scope attribute, `MODS`. This attribute could be specified at CDF creation time and an entry made regarding creation date. Any subsequent changes made to the CDF, including additional variables, changes in min/max values, or modifications to variable values could be documented by adding

Record Number	Time	Longitude	Latitude	Temperature
				Variables
			+40	**20.0 — 19.2**
1	**0000**	**-165 — -150**	|	| |
			+30	**21.7 — 20.7**
				18.2 — 22.0
2	**0100**			| |
				19.3 — 19.2
				19.9 — 19.6
3	**0200**			| |
				19.3 — 19.0
.	.			.
.	.			.
.	.			.
				21.0 — 18.4
24	**2300**			| |
				19.5 — 22.0

Table 1.4: Example CDF - Two-Dimensional Representation (Physical)

entries to `MODS`.

Variable scope attributes further describe the individual variables and their values. Examples of variable attributes would include such things as a field name for the variable, the valid minimum and maximum, the units in which the variable data values are stored, the format in which the data values are to be displayed, a fill value for errant or missing data, and a description of the expected order of data values: increasing or decreasing (monotonicity). Sample attribute entries for the `Temperature` variable from the example above are shown in Table 1.5.

Attribute	Entry value
FIELDNAM	"Recorded temperature"
VALIDMIN	-40.0
VALIDMAX	50.0
SCALEMIN	17.0
SCALEMAX	24.0
UNITS	"deg C "
FORMAT	"F4.1 "
MONOTON	"FALSE "
FILLVAL	-999.9

Table 1.5: Attribute Entries for the `Temperature` Variable

1.7 CDF Toolkit

A set of utility programs are provided with the CDF distribution which allow a user to perform a variety of operations on CDFs without having to write an application program. Each toolkit program is described in detail in Chapter 3.

The available toolkit programs are as follows...

`CDFbrowse`	Allows the display of attribute and variable data.
`CDFlist`	Allows a sequential "flat" view of filtered variable data. `CDFlist` also allows the subsetting of a CDF (creating a new, smaller CDF).
`CDFwalk`	Allows direct and sequential access to related variable values.
`CDFconvert`	Converts CDFs created by previous releases of the CDF library to be updated to the current release. `CDFconvert` also allows the format, encoding, and majority of a CDF to be changed. In all cases, a new CDF is created (the original CDF is not modified).
`CDFskeleton`	Reads a specially formatted text file (called a skeleton table) and creates a CDF. The created CDF is complete except for record-variant data.
`SkeletonTable`	Reads a CDF and produces a specially formatted text file called a skeleton table. The skeleton table may be modified and then input to `CDFskeleton` to create a new CDF.
`CDFinquire`	Produces a report on the contents of a CDF.
`CDFstats`	Produces a report containing various statistics about the variables in a CDF.
`CDFcompare`	Reports the differences between two CDFs.
`CDFscope`	Corrects the attributes in a CDF that have an assumed scope.
`CDFdir`	Produces a directory listing of a CDF's files. For a multi-file CDF, the variable files are listed in ascending numerical order.

1.8 Standard Interface Routines

The CDF software library allows a user to create CDFs and access data randomly or sequentially using the fundamental concepts described previously without knowledge of low-level file format details. Three categories of software functions are utilized to manipulate the components which make up a CDF: general CDF functions, variable functions, and attribute functions.

The general CDF functions are:

Callable from C	Callable from FORTRAN	Purpose
`CDFcreate()`	`CDF_create()`	Creates a new CDF.
`CDFopen()`	`CDF_open()`	Opens an existing CDF.
`CDFdoc()`	`CDF_doc()`	Inquires version/release and copyright notice.
`CDFinquire()`	`CDF_inquire()`	Inquires dimensionality, etc.
`CDFclose()`	`CDF_close()`	Closes a CDF.
`CDFdelete()`	`CDF_delete()`	Deletes a CDF.
`CDFerror()`	`CDF_error()`	Inquires error (status) code meaning.

The variable functions are:

Callable from C	Callable from FORTRAN	Purpose
`CDFvarCreate()`	`CDF_var_create()`	Creates a variable.
`CDFvarNum()`	`CDF_var_num()`	Determines a variable number.
`CDFvarRename()`	`CDF_var_rename()`	Renames a variable.
`CDFvarInquire()`	`CDF_var_inquire()`	Inquires about a variable.
`CDFvarPut()`	`CDF_var_put()`	Writes a variable value.
`CDFvarGet()`	`CDF_var_get()`	Reads a variable value.
`CDFvarHyperPut()`	`CDF_var_hyper_put()`	Writes more than one variable value.
`CDFvarHyperGet()`	`CDF_var_hyper_get()`	Reads more than one variable value.

The attribute functions are:

Callable from C	Callable from FORTRAN	Purpose
`CDFattrCreate()`	`CDF_attr_create()`	Creates an attribute.
`CDFattrNum()`	`CDF_attr_num()`	Determines an attribute number.
`CDFattrRename()`	`CDF_attr_rename()`	Renames an attribute.
`CDFattrInquire()`	`CDF_attr_inquire()`	Inquires about an attribute.
`CDFattrEntryInquire()`	`CDF_attr_entry_inquire()`	Inquires about an attribute entry.
`CDFattrPut()`	`CDF_attr_put()`	Writes an attribute entry.
`CDFattrGet()`	`CDF_attr_get()`	Reads an attribute entry.

1.9 Examples

In this section several examples of how to use the CDF library and toolkit will be presented. The same CDF will be created two different ways: by using just the CDF library and by using the CDF library with the `SkeletonTable` toolkit program.

1.9.1 Creating a CDF

The first example program, written in C, creates a two-dimensional CDF using only CDF library function calls. The CDF created will contain the data and metadata values used in the example presented earlier in this chapter. An input file, `example.dat`, whose format is similar to that of Table 1.1 will be read and its data values written into the CDF.

```
/******************************************************************************
*
*  NSSDC/CDF          Create an example CDF (without using a skeleton table).
```

```
*
*  Version 1.1, 10-Feb-92, Hughes STX
*
*  Modification history:
*
*   V1.0  13-Nov-91, J Love    Original version (for CDF V2.1 Primer chapter).
*   V1.1  10-Feb-92, J Love    Typo city.
*
******************************************************************************/

/******************************************************************************
* Necessary include files.
******************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include "cdf.h"

/******************************************************************************
* Function prototypes.
******************************************************************************/

void StatusHandler ();

/******************************************************************************
* MAIN.
******************************************************************************/

main () {
CDFid id;             /* CDF identifier. */
CDFstatus status;     /* CDF completion status. */

FILE *fp;             /* File pointer - used to read input data file. */

long numDims = 2;                   /* Number of dimensions for the CDF. */
static long dimSizes[2] = {2,2};    /* Dimension sizes. */

long dimVarys[2];     /* Dimension variances. */
long indices[2];      /* Dimension indices. */
long recNum;          /* Record number. */
long attrNum;         /* Attribute number. */

long TimeVarNum;      /* 'Time' variable number. */
long LonVarNum;       /* 'Longitude' variable number. */
long LatVarNum;       /* 'Latitude' variable number. */
long TmpVarNum;       /* 'Temperature' variable number. */

long Time;            /* 'Time' variable value. */
float Lat;            /* 'Latitude' variable value. */
float Lon;            /* 'Longitude' variable value. */
```

```
float Tmp;           /* 'Temperature' variable value. */

long TimeValidMin = 0;          /* 'Time' valid minimum (0000). */
long TimeValidMax = 2359;       /* 'Time' valid maximum (2359). */
long TimeScaleMin = 100;        /* 'Time' scale minimum (0100). */
long TimeScaleMax = 200;        /* 'Time' scale maximum (0200). */
long TimeFillValue = 9999;      /* 'Time' fill value. */

float LonValidMin = -180.0;     /* 'Longitude' valid minimum. */
float LonValidMax = 180.0;      /* 'Longitude' valid maximum. */
float LonScaleMin = -165.0;     /* 'Longitude' scale minimum. */
float LonScaleMax = -150.0;     /* 'Longitude' scale maximum. */
float LonFillValue = -999.9;    /* 'Longitude' fill value. */

float LatValidMin = -90.0;      /* 'Latitude' valid minimum. */
float LatValidMax = 90.0;       /* 'Latitude' valid maximum. */
float LatScaleMin = 30.0;       /* 'Latitude' scale minimum. */
float LatScaleMax = 40.0;       /* 'Latitude' scale maximum. */
float LatFillValue = -999.9;    /* 'Latitude' fill value. */

float TmpValidMin = -40.0;      /* 'Temperature' valid minimum. */
float TmpValidMax = 50.0;       /* 'Temperature' valid maximum. */
float TmpScaleMin = 17.0;       /* 'Temperature' scale minimum. */
float TmpScaleMax = 24.0;       /* 'Temperature' scale maximum. */
float TmpFillValue = -999.9;    /* 'Temperature' fill value. */

/******************************************************************************
* Create the CDF.
******************************************************************************/

status = CDFcreate ("example1", numDims, dimSizes, NETWORK_ENCODING, ROW_MAJOR,
                    &id);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Create variables.
******************************************************************************/

dimVarys[0] = NOVARY;
dimVarys[1] = NOVARY;
status = CDFvarCreate (id, "Time", CDF_INT4, 1, VARY, dimVarys,
                       &TimeVarNum);
if (status != CDF_OK) StatusHandler (status);

dimVarys[0] = VARY;
dimVarys[1] = NOVARY;
status = CDFvarCreate (id, "Longitude", CDF_REAL4, 1, NOVARY, dimVarys,
                       &LonVarNum);
if (status != CDF_OK) StatusHandler (status);

dimVarys[0] = NOVARY;
```

```
dimVarys[1] = VARY;
status = CDFvarCreate (id, "Latitude", CDF_REAL4, 1, NOVARY, dimVarys,
                       &LatVarNum);
if (status != CDF_OK) StatusHandler (status);

dimVarys[0] = VARY;
dimVarys[1] = VARY;
status = CDFvarCreate (id, "Temperature", CDF_REAL4, 1, VARY, dimVarys,
                       &TmpVarNum);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Create attributes.
******************************************************************************/

status = CDFattrCreate (id, "TITLE", GLOBAL_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "FIELDNAM", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "VALIDMIN", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "VALIDMAX", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "SCALEMIN", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "SCALEMAX", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "UNITS", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "FORMAT", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "MONOTON", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrCreate (id, "FILLVAL", VARIABLE_SCOPE, &attrNum);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Write TITLE attribute entry.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"TITLE"), 0, CDF_CHAR, 50,
                     "An example CDF (1).                               ");
```

```
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Write attribute entries for 'Time' variable.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"FIELDNAM"),
                     CDFvarNum(id,"Time"), CDF_CHAR, 20,
                     "Time of observation ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMIN"),
                     CDFvarNum(id,"Time"), CDF_INT4, 1, &TimeValidMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMAX"),
                     CDFvarNum(id,"Time"), CDF_INT4, 1, &TimeValidMax);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMIN"),
                     CDFvarNum(id,"Time"), CDF_INT4, 1, &TimeScaleMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMAX"),
                     CDFvarNum(id,"Time"), CDF_INT4, 1, &TimeScaleMax);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"UNITS"),
                     CDFvarNum(id,"Time"), CDF_CHAR, 20,
                     "Hour/Minute         ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FORMAT"),
                     CDFvarNum(id,"Time"), CDF_CHAR, 8, "I4      ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"MONOTON"),
                     CDFvarNum(id,"Time"), CDF_CHAR, 8, "INCREASE");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FILLVAL"),
                     CDFvarNum(id,"Time"), CDF_INT4, 1, &TimeFillValue);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Write attribute entries for 'Longitude' variable.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"FIELDNAM"),
                     CDFvarNum(id,"Longitude"), CDF_CHAR, 20,
                     "Longitude           ");
```

```
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMIN"),
                     CDFvarNum(id,"Longitude"), CDF_REAL4, 1, &LonValidMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMAX"),
                     CDFvarNum(id,"Longitude"), CDF_REAL4, 1, &LonValidMax);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMIN"),
                     CDFvarNum(id,"Longitude"), CDF_REAL4, 1, &LonScaleMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMAX"),
                     CDFvarNum(id,"Longitude"), CDF_REAL4, 1, &LonScaleMax);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"UNITS"),
                     CDFvarNum(id,"Longitude"), CDF_CHAR, 20,
                     "Degrees east        ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FORMAT"),
                     CDFvarNum(id,"Longitude"), CDF_CHAR, 8, "F6.1    ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"MONOTON"),
                     CDFvarNum(id,"Longitude"), CDF_CHAR, 8, "INCREASE");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FILLVAL"),
                     CDFvarNum(id,"Longitude"), CDF_REAL4, 1, &LonFillValue);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Write attribute entries for 'Latitude' variable.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"FIELDNAM"),
                     CDFvarNum(id,"Latitude"), CDF_CHAR, 20,
                     "Latitude            ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMIN"),
                     CDFvarNum(id,"Latitude"), CDF_REAL4, 1, &LatValidMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMAX"),
                     CDFvarNum(id,"Latitude"), CDF_REAL4, 1, &LatValidMax);
if (status != CDF_OK) StatusHandler (status);
```

```
status = CDFattrPut (id, CDFattrNum(id,"SCALEMIN"),
                     CDFvarNum(id,"Latitude"), CDF_REAL4, 1, &LatScaleMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMAX"),
                     CDFvarNum(id,"Latitude"), CDF_REAL4, 1, &LatScaleMax);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"UNITS"),
                     CDFvarNum(id,"Latitude"), CDF_CHAR, 20,
                     "Degrees north       ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FORMAT"),
                     CDFvarNum(id,"Latitude"), CDF_CHAR, 8, "F6.1    ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"MONOTON"),
                     CDFvarNum(id,"Latitude"), CDF_CHAR, 8, "DECREASE");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FILLVAL"),
                     CDFvarNum(id,"Latitude"), CDF_REAL4, 1, &LatFillValue);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Write attribute entries for 'Temperature' variable.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"FIELDNAM"),
                     CDFvarNum(id,"Temperature"), CDF_CHAR, 20,
                     "Temperature value   ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMIN"),
                     CDFvarNum(id,"Temperature"), CDF_REAL4, 1, &TmpValidMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"VALIDMAX"),
                     CDFvarNum(id,"Temperature"), CDF_REAL4, 1, &TmpValidMax);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMIN"),
                     CDFvarNum(id,"Temperature"), CDF_REAL4, 1, &TmpScaleMin);
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"SCALEMAX"),
                     CDFvarNum(id,"Temperature"), CDF_REAL4, 1, &TmpScaleMax);
if (status != CDF_OK) StatusHandler (status);
```

```
status = CDFattrPut (id, CDFattrNum(id,"UNITS"),
                     CDFvarNum(id,"Temperature"), CDF_CHAR, 20,
                     "Degrees celsius     ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FORMAT"),
                     CDFvarNum(id,"Temperature"), CDF_CHAR, 8, "F4.1    ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"MONOTON"),
                     CDFvarNum(id,"Temperature"), CDF_CHAR, 8, "FALSE   ");
if (status != CDF_OK) StatusHandler (status);

status = CDFattrPut (id, CDFattrNum(id,"FILLVAL"),
                     CDFvarNum(id,"Temperature"), CDF_REAL4, 1, &TmpFillValue);
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Read input values for variables and write them to the CDF.  Not every value
* must be written to the CDF - many of the values are redundant.  The 'Time'
* value only has to be written once per CDF record (every 4 input records).
* The 'Longitude' and 'Latitude' values are only written to the first CDF
* record (and only at the appropriate indices).  Each 'Temperature' value read
* is written to the CDF.
******************************************************************************/

fp = fopen ("example.dat", "r");
if (fp == NULL) {
  printf ("Error opening input file.\n");
  exit (-1);
}

for (recNum = 0; recNum < 24; recNum++) {
   for (indices[0] = 0; indices[0] < 2; indices[0]++) {
      for (indices[1] = 0; indices[1] < 2; indices[1]++) {
         fscanf (fp, "%d %f %f %f", &Time, &Lon, &Lat, &Tmp);

         if (indices[0] == 0 && indices[1] == 0) {
           status = CDFvarPut (id, TimeVarNum, recNum, indices, &Time);
           if (status != CDF_OK) StatusHandler (status);
         }

         if (recNum == 0 && indices[1] == 0) {
           status = CDFvarPut (id, LonVarNum, recNum, indices, &Lon);
           if (status != CDF_OK) StatusHandler (status);
         }

         if (recNum == 0 && indices[0] == 0) {
           status = CDFvarPut (id, LatVarNum, recNum, indices, &Lat);
           if (status != CDF_OK) StatusHandler (status);
         }
```

```
        status = CDFvarPut (id, TmpVarNum, recNum, indices, &Tmp);
        if (status != CDF_OK) StatusHandler (status);
      }
   }
}

fclose (fp);

/******************************************************************************
* Close CDF.
******************************************************************************/

status = CDFclose (id);
if (status != CDF_OK) StatusHandler (status);

return;
}

/******************************************************************************
* Status handler.
******************************************************************************/

void StatusHandler (status)
CDFstatus status;
{
char message[CDF_ERRTEXT_LEN+1];

if (status < CDF_WARN) {
  printf ("An error has occurred, halting...\n");
  CDFerror (status, message);
  printf ("%s\n", message);
  exit (status);
}
else
  if (status < CDF_OK) {
    printf ("Warning, function may not have completed as expected...\n");
    CDFerror (status, message);
    printf ("%s\n", message);
  }
  else
    if (status > CDF_OK) {
      printf ("Function completed successfully, but be advised that...\n");
      CDFerror (status, message);
      printf ("%s\n", message);
    }
return;
}
```

1.9.2 Creating a CDF, an easier way

The CDF toolkit program `CDFskeleton` is provided to make the task of creating a CDF easier for a programmer. `CDFskeleton` reads a specially formatted text file call a skeleton table and generates a CDF. Everything about a CDF can be specified in a skeleton table except data values for variables that vary from record to record (record-variant). The toolkit program `SkeletonTable` is also provided. It reads an existing CDF and produces a skeleton table. The skeleton table for the CDF created using only the CDF library in Section 1.9.1 would be as follows.

```
! Skeleton table for the "example" CDF.
! Generated: Mon 10 Feb 1992 10:53:57

#header

                 CDF NAME: example1
            DATA ENCODING: NETWORK
                 MAJORITY: ROW
                   FORMAT: SINGLE

! Variables G.Attributes V.Attributes Records  Dims  Sizes
! --------- ------------ ------------ -------  ----  -----
      4           1            9         1       2    2 2


#GLOBALattributes

! Attribute         Entry       Data
! Name              Number      Type        Value
! ---------         ------      ----        -----

  "TITLE"               1:    CDF_CHAR     { "An example CDF (1).        " -
                                              "                        " } .


#VARIABLEattributes

  "FIELDNAM"
  "VALIDMIN"
  "VALIDMAX"
  "SCALEMIN"
  "SCALEMAX"
  "UNITS"
  "FORMAT"
  "MONOTON"
  "FILLVAL"


#variables

! Variable            Data       Number     Record    Dimension
```

```
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "Time"            CDF_INT4       1          T          F F

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "FIELDNAM"      CDF_CHAR     { "Time of observation " }
    "VALIDMIN"      CDF_INT4     { 0 }
    "VALIDMAX"      CDF_INT4     { 2359 }
    "SCALEMIN"      CDF_INT4     { 100 }
    "SCALEMAX"      CDF_INT4     { 200 }
    "UNITS"         CDF_CHAR     { "Hour/Minute         " }
    "FORMAT"        CDF_CHAR     { "I4      " }
    "MONOTON"       CDF_CHAR     { "INCREASE" }
    "FILLVAL"       CDF_INT4     { 9999 } .


! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "Longitude"       CDF_REAL4      1          F          T F

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "FIELDNAM"      CDF_CHAR     { "Longitude           " }
    "VALIDMIN"      CDF_REAL4    { -180.0 }
    "VALIDMAX"      CDF_REAL4    { 180.0 }
    "SCALEMIN"      CDF_REAL4    { -165.0 }
    "SCALEMAX"      CDF_REAL4    { -150.0 }
    "UNITS"         CDF_CHAR     { "Degrees east        " }
    "FORMAT"        CDF_CHAR     { "F6.1    " }
    "MONOTON"       CDF_CHAR     { "INCREASE" }
    "FILLVAL"       CDF_REAL4    { -999.9 } .

  ! NRV values follow...

  [ 1, 1 ] = -165.0
  [ 2, 1 ] = -150.0


! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "Latitude"        CDF_REAL4      1          F          F T
```

```
  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "FIELDNAM"      CDF_CHAR     { "Latitude            " }
    "VALIDMIN"      CDF_REAL4    { -90.0 }
    "VALIDMAX"      CDF_REAL4    { 90.0 }
    "SCALEMIN"      CDF_REAL4    { 30.0 }
    "SCALEMAX"      CDF_REAL4    { 40.0 }
    "UNITS"         CDF_CHAR     { "Degrees north       " }
    "FORMAT"        CDF_CHAR     { "F6.1    " }
    "MONOTON"       CDF_CHAR     { "DECREASE" }
    "FILLVAL"       CDF_REAL4    { -999.9 } .

  ! NRV values follow...

  [ 1, 1 ] = 40.0
  [ 1, 2 ] = 30.0


! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "Temperature"     CDF_REAL4       1          T         T T

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "FIELDNAM"      CDF_CHAR     { "Temperature value   " }
    "VALIDMIN"      CDF_REAL4    { -40.0 }
    "VALIDMAX"      CDF_REAL4    { 50.0 }
    "SCALEMIN"      CDF_REAL4    { 17.0 }
    "SCALEMAX"      CDF_REAL4    { 24.0 }
    "UNITS"         CDF_CHAR     { "Degrees celsius     " }
    "FORMAT"        CDF_CHAR     { "F4.1    " }
    "MONOTON"       CDF_CHAR     { "FALSE   " }
    "FILLVAL"       CDF_REAL4    { -999.9 } .

#end
```

Assuming that `CDFskeleton` was used to create a CDF containing the metadata and data in the above skeleton table, the following FORTRAN program would be used to complete the creation of the CDF.

```
        PROGRAM exampleSKT

C-----------------------------------------------------------------------------
C
```

```
C  NSSDC/CDF                    Create an example CDF (using skeleton table).
C
C  Version 1.0, 13-Nov-91, Hughes STX
C
C  Modification history:
C
C   V1.0  13-Nov-91, J Love    Original version (for CDF V2.1 Primer chapter).
C
C-----------------------------------------------------------------------------

        INCLUDE '../../include/cdf.inc'

        INTEGER*4 id            ! CDF identifier.
        INTEGER*4 status        ! CDF completion status.

        INTEGER*4 lun           ! Logical unit number for input data file.

        INTEGER*4 indices(2)    ! Dimension indices.
        INTEGER*4 rec_num       ! Record number.

        INTEGER*4 time_var_num     ! 'Time' variable number.
        INTEGER*4 tmp_var_num      ! 'Temperature' variable number.

        INTEGER*4 time          ! 'Time' variable value.
        REAL*4    lat           ! 'Latitude' variable value.
        REAL*4    lon           ! 'Longitude' variable value.
        REAL*4    tmp           ! 'Temperature' variable value.

        DATA lun/1/

C-----------------------------------------------------------------------------
C Open the CDF.
C-----------------------------------------------------------------------------

        CALL CDF_open ('example2', id, status)
        IF (status .NE. CDF_OK) CALL StatusHandler (status)

C-----------------------------------------------------------------------------
C Determine variable numbers.
C-----------------------------------------------------------------------------

        time_var_num = CDF_var_num (id, 'Time')
        IF (time_var_num .LT. CDF_OK) CALL StatusHandler (status)

        tmp_var_num = CDF_var_num (id, 'Temperature')
        IF (tmp_var_num .LT. CDF_OK) CALL StatusHandler (status)

C-----------------------------------------------------------------------------
C Read input values for variables and write them to the CDF.  Not every value
C must be written to the CDF - many of the values are redundant.  The 'Time'
C value only has to be written once per CDF record (every 4 input records).
```

```
C The 'Longitude' and 'Latitude' values are not written at all because they
C had been specified in the skeleton table.  Each 'Temperature' value read
C is written to the CDF.
C-----------------------------------------------------------------------------

        OPEN (lun, FILE='example.dat', ERR=99)

        DO rec_num = 1, 24
          DO x1 = 1, 2
            DO x2 = 1, 2
              indices(1) = x1
              indices(2) = x2

              READ (lun, *, ERR=99) time, lon, lat, tmp

              IF (indices(1) .EQ. 1 .AND. indices(2) .EQ. 1) THEN
                CALL CDF_var_put (id, time_var_num, rec_num, indices,
     .                            time, status)
                IF (status .NE. CDF_OK) CALL StatusHandler (status)
              END IF

              CALL CDF_var_put (id, tmp_var_num, rec_num, indices,
     .                          tmp, status)
              IF (status .NE. CDF_OK) CALL StatusHandler (status)
            END DO
          END DO
        END DO

        CLOSE (lun, ERR=99)

C-----------------------------------------------------------------------------
C Close CDF.
C-----------------------------------------------------------------------------

        CALL CDF_close (id, status)
        IF (status .NE. CDF_OK) CALL StatusHandler (status)

        STOP

C-----------------------------------------------------------------------------
C Input file error handler.
C-----------------------------------------------------------------------------

 99     WRITE (6,101)
 101    FORMAT (' ','Error reading input file')
        STOP

        END
```

```
C-----------------------------------------------------------------------------
C Status handler.
C-----------------------------------------------------------------------------

        SUBROUTINE StatusHandler (status)
        INTEGER*4 status

        INCLUDE '../../include/cdf.inc'

        CHARACTER message*(CDF_ERRTEXT_LEN)

        IF (status .LT. CDF_WARN) THEN
          WRITE (6,10)
10        FORMAT (' ','Error (halting)...')
          CALL CDF_error (status, message)
          WRITE (6,11) message
11        FORMAT (' ',A)
          STOP
        ELSE
          IF (status .LT. CDF_OK) THEN
            WRITE (6,12)
12          FORMAT (' ','Warning...')
            CALL CDF_error (status, message)
            WRITE (6,13) message
13          FORMAT (' ',A)
          ELSE
            IF (status .GT. CDF_OK) THEN
              WRITE (6,14)
14            FORMAT (' ','Be advised that...')
              CALL CDF_error (status, message)
              WRITE (6,15) message
15            FORMAT (' ',A)
            END IF
          END IF
        END IF

        RETURN
        END
```

The CDF was opened (since it already existed) and the values for only the `Time` and `Temperature` variables were written to the CDF. All of the other functions performed by the program in Section 1.9.1 were done by the `CDFskeleton` program when it read the skeleton table.

Chapter 2

General CDF Concepts

2.1 CDF Interfaces

Several interfaces to the CDF library exist. They are described in the following sections.

2.1.1 Standard Interface

The Standard interface[1] provides a standard set of routines with which to access a CDF. Not all CDF features are available with the Standard interface. The Internal interface must be used to perform operations not available with the Standard interface routines. The Standard interface is callable from both C and FORTRAN applications. Table 2.1 lists the routines available when using the Standard interface. Each routine is described in detail in the corresponding programming reference chapter.

2.1.2 Internal Interface

The Internal interface may be used to perform all supported CDF operations. The Internal interface must be used to perform those operations not available with the Standard interface. Table 2.2 lists the routines available when using the Internal interface. Each is described in detail in the corresponding programming reference chapter.

2.1.3 Obsolete Interface, VMS Systems Only

FORTRAN applications written for CDF Version 1 on VMS systems may still be used, without modification, via the Obsolete interface. While we don't encourage the use of the Obsolete interface, it is provided for use while existing applications are being updated for the new Standard interface routines. To use the Obsolete interface, simply relink the application with the Obsolete interface module (`CDF_O_IF.OBJ`) and the new CDF

[1] The Standard interface was previously known as the C/FORTRAN interface(s). That was an unfortunate choice of terms since the Internal interface is also callable from C and FORTRAN.

library (`LIBCDF.OLB`). An example of this (assuming that `CDF$LIB` has been properly defined) would be as follows...

```
$ LINK file1, file2, ...fileN, CDF$LIB:CDF_O_IF, CDF$LIB:LIBCDF/LIBRARY
```

where `file1, file2, ...fileN` are the application's object modules. Note that `CDF_O_IF` must appear before `LIBCDF` in the `LINK` command.

There are several conditions which apply to the use of the Obsolete interface.

1. The functions (and their associated arguments) available for use are exactly the same as in CDF Version 1 (refer to the CDF Version 1 Implementer's Guide for details). New CDF features are not available.
2. CDFs created will be Version 2 CDFs. To create a Version 1 CDF you must use the CDF Version 1 library.
3. Version 2 CDFs may be accessed except in those cases where the Obsolete interface does not support a new Version 2 feature. In those cases an error code indicating the incompatibility will be returned. These incompatibilities deal mainly with attribute entries (in Version 1 the data type was specified for the attribute — in Version 2 the data type is specified per entry).
4. Version 1 CDFs may be accessed but the only modification allowed is the updating of existing variable values. No other modifications are allowed to Version 1 CDFs (with the Version 2 CDF library).

2.2 Access to CDFs

Both Version 1 and Version 2 CDFs may be accessed with the Version 2 CDF library. Restrictions are described in the following sections.

2.2.1 Version 2 CDFs

Any operation on a Version 2 CDF may be performed with the Standard and Internal interfaces.

On VMS systems, when using the Obsolete interface, all operations are allowed except when an incompatibility exists between the contents of the Version 2 CDF and the capabilities of the Obsolete interface (see Section 2.1.3).

2.2.2 Version 1 CDFs

All types of read access are allowed on Version 1 CDFs using either the Standard or Internal interface. The only write access allowed is the modification of existing variable values.

2.3 CDF Format

There are two CDF formats: multi-file and single-file. The choice of which format to use will depend on how the CDF is to be accessed. Note that the `CDFconvert` toolkit program can be used to change the format of a CDF.

A CDF's format is normally specified when the CDF is created. In a user application, the Internal interface must be used to specify a single-file CDF (the default using the Standard interface is multi-file). The format of an existing CDF can be changed (with the Internal interface) if no variables have been created in the CDF. If the `CDFskeleton` toolkit program is used to create a CDF, the format is specified in the skeleton table (see Section 3.9).

2.3.1 Multi-file CDFs

A multi-file CDF (`MULTI_FILE`) consists of one file (with extension `.cdf`) containing control information and attribute entry data and a separate file for each variable defined in the CDF (with extensions `.v0`,`.v1`,...). Each variable file contains the data values for the corresponding variable (the control information for each variable is stored in the `.cdf` file).

2.3.2 Single-file CDFs

A single-file CDF (`SINGLE_FILE`) consists of only one file (with extension `.cdf`). This file contains the control information for the entire CDF, the attribute entry data, and all of the variable data. An indexing scheme is used to provide efficient access to variable data values.

2.3.3 Performance

The most efficient access to CDF variables will usually occur when the CDF has the multi-file format. The extra overhead involved with the indexing scheme used in single-file CDFs is small so the difference may not be significant (especially if hyper reads/writes are used). The drawback to using the multi-file format is that more than one file is associated with a CDF (which may or may not be a problem for your system management).

There is a case in which the single-file format may be more efficient. If a CDF has a large number of variables (larger than the number of files which may be open at once by an application) and the variables values are accessed variable-by-variable (rather than value-by-value within a variable), the multi-file format may be much slower than the single-file format. This is because the CDF library will have to close one variable file and then open another as each variable value is accessed by the application (since the operating system's open file limit will be reached). If the application was to access every value for a variable before going on to the next variable, this would not occur (but it might create complications for your application).

Note that the format of a CDF can always be converted using the `CDFconvert` toolkit program (which creates a new CDF with the specified format). Section 3.4 describes `CDFconvert`.

2.4 CDF Encoding

The encoding of a CDF determines how attribute entry data and variable data values are stored on disk in the CDF file(s). An application program never has to concern itself with the encoding of the CDF being accessed. The CDF library performs all of the encoding and decoding of data values for the application. The application always deals with data values in the host encoding of the computer being used.

A CDF's encoding is specified when the CDF is created. The encoding of an existing CDF may be changed with the Internal interface if no variable values or attribute entries have been written (variables and attributes may exist however). If the `CDFskeleton` toolkit program is used to create a CDF, the encoding is specified in the skeleton table (see Section 3.9).

The encoding specified when creating a CDF may be either host encoding or network encoding.

2.4.1 Host Encoding

Host encoding (`HOST_ENCODING`) specifies that variable and attribute entry data values be written to the CDF in the encoding of the computer being used. The following host encodings (computers) are supported.

`VAX_ENCODING`	VAX and microVAX computers.
`SUN_ENCODING`	Sun computers.
`MIPSEB_ENCODING`	Silicon Graphics Iris and Power Series computers.
`MIPSEL_ENCODING`	DECstation computers.
`IBMRS_ENCODING`	IBM RS-6000 series computers.
`HP_ENCODING`	HP 9000 series computers.
`IBMPC_ENCODING`	IBM PC personal computers.

A host encoded CDF is only readable on the host computer that created the CDF. If the CDF must be transferred to another type computer, the toolkit program `CDFconvert` must be used to first convert the CDF to network encoding (a new CDF is created by `CDFconvert`). It is not possible to convert from one computer's encoding directly to another computer's encoding. When the network encoded CDF is transferred to the other computer, it may then be converted (with `CDFconvert`) to the host encoding of that computer (if desired — a CDF can always be used in network encoding).

2.4.2 Network Encoding

Network encoding (`NETWORK_ENCODING`) specifies that variable and attribute entry data values be written to the CDF in a modified XDR (External Data Representation) format.[2] As values are written to the CDF, the CDF library encodes them into network encoding. Likewise, as values are read from the CDF, the CDF library decodes them from network encoding to the host computer encoding. Network encoded CDFs are readable on any platform supported by CDF.

[2] Modified because data values never increase in size.

2.4.3 Performance

The best performance when accessing a CDF will occur when that CDF is in the host encoding of the computer being used because no encoding or decoding has to be performed by the CDF library. A CDF which must be portable between two or more different types of computers must, however, be network encoded. Obviously there are trade-offs as to which encoding should be used in any one particular case. Keep in mind that a CDF can always be converted to host encoding (with `CDFconvert`) before being accessed.

2.5 CDF Data Types

CDF supports a variety of data types consistent with the types available with C and FORTRAN compilers on most computers. All data types are based on an 8-bit byte.

2.5.1 Integer Data Types

`CDF_BYTE`	1-byte, signed integer.
`CDF_INT1`	1-byte, signed integer.
`CDF_UINT1`	1-byte, unsigned integer.
`CDF_INT2`	2-byte, signed integer.
`CDF_UINT2`	2-byte, unsigned integer.
`CDF_INT4`	4-byte, signed integer.
`CDF_UINT4`	4-byte, unsigned integer.

2.5.2 Floating Point Data Types

`CDF_REAL4`	4-byte, single precision floating point.
`CDF_FLOAT`	4-byte, single precision floating point.
`CDF_REAL8`	8-byte, double precision floating point.
`CDF_DOUBLE`	8-byte, double precision floating point.

2.5.3 Character Data Types

`CDF_CHAR`	1-byte, character.
`CDF_UCHAR`	1-byte, unsigned character.

Character data types are unique for variables in that they are the only data types for which more than one element per value is allowed. Each variable value consists of a character string with the number of elements being the number of characters. More than one element is allowed for any of the data types when dealing with attribute entries.

2.5.4 EPOCH Data Type

CDF_EPOCH	8-byte, double precision floating point.

The EPOCH data type is used to store time values referenced from a particular epoch. For NSSDC applications that epoch is `01-Jan-0000 00:00:00.000`.[3] CDF_EPOCH values are the number of milliseconds since the epoch. CDF toolkit programs will display and expect CDF_EPOCH values in the format...

```
dd-mmm-yyyy hh:mm:ss.ccc
```

where

dd	is the day of the month, 1-31,
mmm	is the month, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec,
yyyy	is the year, AD,
hh	is the hour, 0-23,
mm	is the minute, 0-59,
ss	is the second, 0-59, and
ccc	is the millisecond, 0-999.

Functions exist which compute and breakdown CDF_EPOCH values. These functions are descibed in Section 4.8 for C applications and in Section 5.8 for FORTRAN applications.

2.5.5 Equivalent Data Types

Certain data types are considered equivalent with respect to their representation in memory and in a CDF. The following groups contain equivalent data types.

CDF_CHAR	CDF_INT2	CDF_INT4	CDF_REAL4	CDF_REAL8
CDF_UCHAR	CDF_UINT2	CDF_UINT4	CDF_FLOAT	CDF_DOUBLE
CDF_INT1				CDF_EPOCH
CDF_UINT1				
CDF_BYTE				

Note that while the signed and unsigned forms of a data type are considered equivalent by the CDF library, they must be correctly interpreted by an application to produce the desired results.

[3] I know what you're thinking. The year 0 AD never existed. If it makes you feel better, think of the epoch year as 1 BC rather than 0 AD.

2.6 CDF Limits

Limits within the CDF library are defined in the appropriate include files: `cdf.h` for C applications and `cdf.inc` for FORTRAN applications. The following limits exist.

`CDF_MAX_CDFS`	The maximum number of CDFs which may be open at any one time. The actual number may be less if the operating system being used allows fewer files to be opened at once. This is because the `.cdf` file of each open CDF is kept open by the CDF library until the CDF is closed. For multi-file CDFs, variable files are opened as the variables are accessed. When a new file needs to be opened (for a CDF or variable), a variable file will be closed if necessary.
`CDF_MAX_VARS`	The maximum number of variables allowed in a CDF.
`CDF_MAX_DIMS`	The maximum number of dimensions a CDF may have.
`CDF_VAR_NAME_LEN`	The maximum number of characters in a variable name.
`CDF_ATTR_NAME_LEN`	The maximum number of characters in an attribute name.
`CDF_PATHNAME_LEN`	The maximum number of characters in a pathname used to specify a CDF.

Most of these limits can be raised if necessary. Contact CDF User Support if that becomes necessary.

2.7 Variables

CDF variables are the mechanism for storing data (attributes are used to store metadata). A new variable may be created in a CDF at any time.

2.7.1 Naming

Each variable in a CDF has a unique name. Variable names are case sensitive regardless of the operating system being used. Variable names may consist of any printable characters including blanks. Trailing blanks, however, are ignored when the CDF library compares variable names. `"LAT"` and `"LAT     "` are considered to be the same name so they cannot both exist in the same CDF. This was done because Version 1 of CDF padded variable names on the right with blanks out to eight characters. To allow CDF Version 2 applications to read Version 1 CDFs without having to be concerned with the trailing blanks, the trailing blanks are ignored when comparing variable names. The trailing blanks are returned as part of the name, however, when a variable is inquired by an application program.

2.7.2 Numbering

The variables in a CDF are numbered consecutively starting at one (1) for FORTRAN applications and starting at zero (0) for C applications.

2.7.3 Dimensionality

Variable values are stored in arrays. A variable's dimensionality refers to the number of dimensions and the dimension sizes of these arrays. Each variable in a CDF has the same dimensionality.[4] An array of values exists for each variable at each record in a CDF (the values may not be physically stored but may be virtual, see Sections 2.7.4, 2.7.6, and 2.7.7). The array of values for a variable within a CDF record is also referred to as a variable record (a set of variable records make up a CDF record).

A variable array in a CDF having two or more dimensions also contains subarrays. For instance, in a 3-dimensional CDF with dimension sizes `10,20,30`, each variable array consists of ten 2-dimensional subarrays of size `20,30` and each of those 2-dimensional subarrays consists of twenty 1-dimensional subarrays of size `30`. Subarrays will be referred to when discussing other properties of CDF variables.

2.7.4 Records

A CDF record is a set of variable arrays, one per variable in the CDF. A CDF may contain zero or more records. The variable arrays in a particular record are generally related to each other in some way (often time). A variable record is simply the corresponding variable array within a CDF record.

Physical variable records are actually stored in the CDF. Virtual variable records are not actually stored but do exist in the conceptual view of the variable provided by CDF. Virtual records can occur in a CDF because of the following reasons...

1. If a variable's values do not vary from record to record (record variance of `NOVARY`), all of that variable's records beyond the first one are virtual. If a record has not yet been written to that variable, then all of its records are virtual.
2. If a variable's values do vary from record to record (record variance of `VARY`) but more records have been written to some other variable, then the records beyond the last record actually written are virtual (up to the last record written for the other variable).

Record variance is described in Section 2.7.6.

The maximum record written is maintained by the CDF library for each variable individually and for the CDF as a whole. The maximum record written for the entire CDF is simply the largest of the maximum variable record numbers.

Figure 2.1 illustrates the relationships between physical and virtual variable records. Variable 1 has five records that were physically written (the number of records in the CDF would also be five). Only two records were physically written to variable 2 so the last three records are virtual (containing fill values). Only one record can be physically written to variable 3 because its record variance is `NOVARY`. The other four records are virtual and contain the same values as the first record. Because a record has not been physically written to variable 4, all of its records are virtual containing the fill value for that variable. Likewise, since no records have been written to variable 5, all of its records are also virtual and contain fill values.

Note that a variable's records do not have to be written sequentially starting at the first record. The records may be written in any order. For a variable with `VARY` record variance, if a new record more than one record

[4] This will be changed in a future release.

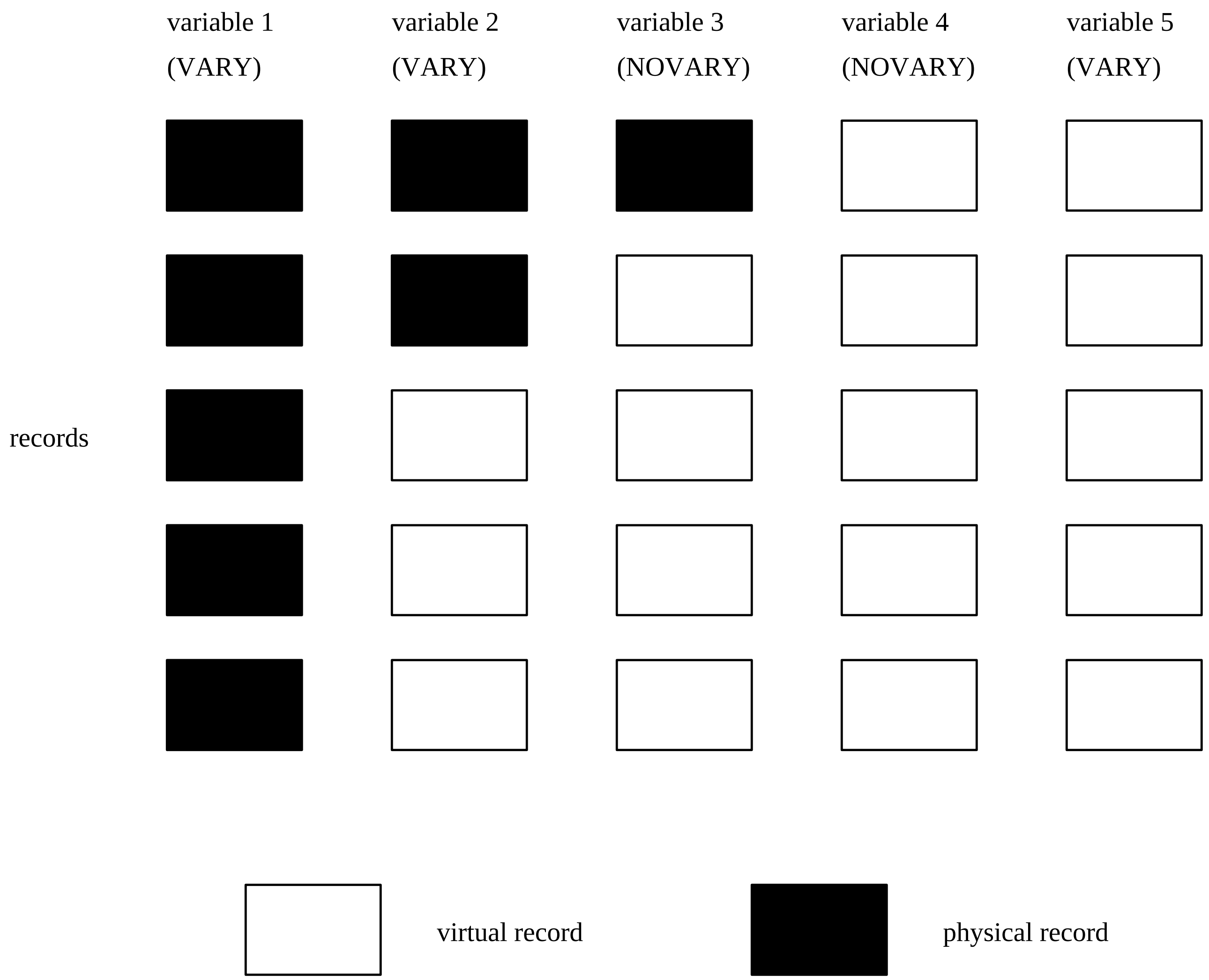

Figure 2.1: Physical vs. Virtual Records

beyond the current maximum record for a variable is written, the intervening records will be physically written and contain fill values. Also, when one or more values are written to a new physical record, the entire record is actually written with fill values being used for the unspecified values. Variable fill values are described in Section 2.7.11.

Initial Records

The Internal interface may be used to specify an initial number of records for a variable. The fill value for the variable is written at each record as if the application had done so itself. The Internal interface allows this to be done more conveniently with only one function call. Note that the default fill value for the variable's data type will be used unless a fill value is specified for the variable (also using the Internal interface). The fill value for a variable must be specified before the number of initial records are specified.

Specifying a number of initial records for a variable would usually only be done for a CDF with the single-file format. Because the records would be allocated contiguously within the CDF file, the indexing scheme would require fewer entries making the access to that variable more efficient.

The number of initial records specified would in most cases be the number of records planned for a variable. Note that additional records may be added to a variable at any time. After the initial records are written with the fill value, the actual data values would be written by the application.

Extend Records

As new records are written to a variable, additional records are allocated at the end of that variable. For CDFs with the multi-file format, only the records actually needed are appended to the corresponding variable file. The records always remain contiguous (allowing the location of a record to be determined very efficiently).

For single-file CDFs, however, the new records are allocated at the end of the `.cdf` file. The new records may not be located contiguously with the other records for the variable. This results in a new entry being created in the indexing scheme used to locate variable records. The indexing scheme is more efficient if fewer entries are needed (resulting from larger blocks of contiguous records being allocated). The CDF library attempts to optimize the indexing scheme by allocating more records than are actually needed with the assumption that they will eventually be used. A minimum number of records to allocate is maintained for each variable.

Because an application knows how records will be added to a variable more so than the CDF library, the Internal interface may be used to explicitly specify the number of records to extend a variable when necessary. This would usually be done only for CDFs with the single-file format.

Note the distinction between records allocated and records actually written. The CDF library may allocate more records than are actually written by an application for the reasons stated above. When an application inquires the number of records for a variable, the number written, not the number allocated, will be returned by the CDF library.

2.7.5 Data Specification

Each variable in a CDF has a defined data specification. A variable's data specification consists of a data type and a number of elements of that data type. A variable's data specification is specified when the variable is created. The data specification of an existing variable may also be changed if either of the following conditions are true.

1. Values have not yet been written to the variable (including an explicitly written fill value, see Section 2.7.11).
2. The old data type and new data type are considered equivalent and the number of elements for the variable are the same. Equivalent data types are described in Section 2.5.5.

Data Type

The supported data types are described in Section 2.5. Variables having any combination of data types may exist in the same CDF.

Number of Elements

In addition to a data type, each variable also has a number of elements. This refers to the number of elements of the data type at each variable value. For character string data types this is the number of characters in each string (a variable value consists of the entire character string). The character string can be thought of as an array of characters. For non-character string data types, this must always be one (1).[5] An array of elements per variable value is not allowed for non-character string data types. Note that for attribute entries, more than one element is allowed for any data type.

2.7.6 Record Variance

A variable's record variance specifies whether or not the variable's values change from record to record. The effect of a variable's record variance is defined as follows...

`VARY` The values do change from record to record. Each variable record is physically written with no gaps between records (ie. if a record more than one beyond the maximum record is written, the intervening records are also physically written and contain fill values). If a record is read beyond the maximum record written to a variable, the fill value for the variable is returned. Variables of this type are referred to as record-variant (RV).

`NOVARY` The values do not change from record to record. Only one record may be physically written to the variable. Each record (in the conceptual view of the variable) contains the same values. Variables of this type are referred to as non-record-variant (NRV).

[5] A future release of CDF may relax this restriction.

Section 2.7.4 describes variable records in more detail.

A variable's record variance is specified when the variable is created. The record variance of an existing variable may be changed only if values have not yet been written to that variable (an explicit fill value may have been specified).

2.7.7 Dimension Variance

A variable's dimension variances specify whether or not the values change along the corresponding dimension. The effects of a dimension variance are defined as follows...

`VARY` — The values do change along the dimension. All of the values for the dimension (or all of the subarrays) are physically stored.

`NOVARY` — The values do not change along the dimension. Only one value (or subarray) is physically written for that dimension. Each value (or subarray) along that dimension (in the conceptual view of the variable) is the same.

Figure 2.2 illustrates the effect of dimension variances on a variable in a 2-dimensional CDF for a particular record. In variable 1 each value in the array is physically stored and therefore unique. Because variable 2 does not vary along the second dimension, each value along that dimension is the same so only one value for that dimension is physically stored (the other values are virtual). Likewise for variable 3 which does not vary along the first dimension. Variable 4 does not vary along either dimension. Only one value is physically stored for the array — all of the other values are the same (they are virtual).

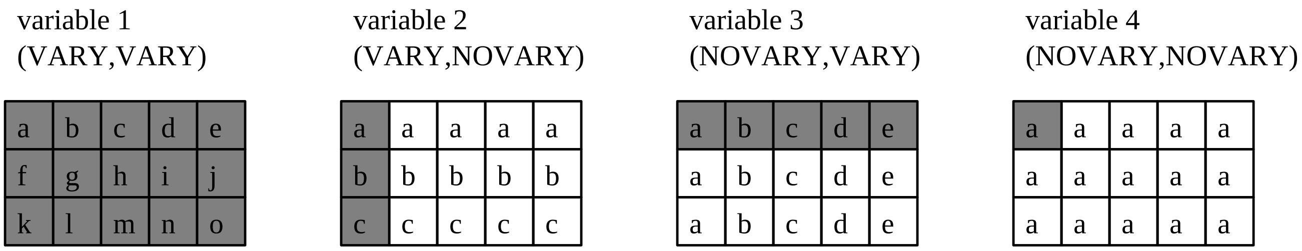

Figure 2.2: Physical vs. Virtual Dimensions

A variable's dimension variances are specified when the variable is created. The dimension variances of an existing variable may be changed only if values have not yet been written to that variable (an explicit fill value may have been specified).

2.7.8 Majority

The majority of a CDF describes how variable values within each variable array (record) are stored. Each variable in a CDF has the same majority. The majority can be either row major or column major.

`ROW_MAJOR` — Row majority. The first dimension changes the slowest.

`COL_MAJOR` — Column majority. The first dimension changes the fastest.

For example, an array for a variable with `VARY,VARY` dimension variances in a 2-dimensional CDF with dimension sizes `2,4` and row majority would be stored as follows...

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

where `v(i,j)` is the value at indices `i,j`. If the CDF had column majority, the array would be stored as follows...

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

In each case, `v(1,1)` is stored at the low address.

An application needs to be concerned with the majority of a CDF in the following cases...

1. When performing a hyper read, the values placed in the buffer by the CDF library will be in the majority of the CDF. The application must process the values according to that majority.

   When performing a hyper write, the CDF library expects the values in the buffer to be in the majority of the CDF. The application must place the values into the buffer in that majority.

2. When using sequential access, the values are read/written in the order imposed by the majority of the CDF.

3. When single value reads/writes are performed, the majority should have little or no effect. The CDF library uses a caching scheme to optimize the random access to variable values. If all of the values of a record are to be read/written, there may be a slight increase in performance if the values are accessed with (rather than against) the majority.[6] For example, if the majority is `ROW_MAJOR`, increment the last index the fastest.

A CDF's majority is specified when the CDF is created. The majority of an existing CDF may be changed using the Internal interface only if variable values have not yet been written (variables may exist and explicit fill values may have been specified).

2.7.9 Hyper Access

Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library. In fact, the entire variable may be accessed at once (if a large enough memory buffer is available to your application). Hyper reads cause the CDF library to read from the variable record(s) in the CDF and place the values into a memory buffer provided by the application. Hyper writes cause the CDF library to take values from a memory buffer provided by the application and write them to the variable records in the CDF. Six parameters are specified when performing a hyper read/write...

`record number`	The record number at which to start the access.
`record count`	The number of records to access.

[6] Hyper reads/writes being the most efficient of course.

`record interval`	The interval between records being accessed. An interval of two (2) would indicate that every other record is to be accessed.
`dimension indices`	The indices within each record at which the access should begin.
`dimension counts`	The number of values along each dimension that should be accessed.
`dimension intervals`	For each dimension, the interval between values being accessed. An interval of three (3) would indicate that every third value is to be accessed.

For 0-dimensional CDFs, the dimension indices, counts, and intervals are not applicable.

A hyper access may or may not read/write a contiguous set of values stored for a variable in the CDF. However, the values in the memory buffer received/provided by the application are contiguous.

Hyper access is based on the conceptual view of a variable. For instance, if a variable has a record variance of `NOVARY` and a hyper read of the first five records for that variable is requested, the CDF library will read the single record that is physically stored and place it five times (contiguously) into the memory buffer provided by the application. The same applies to any dimension variances which are `NOVARY`. For example, if the count for a dimension is three and the dimension variance is `NOVARY`, the one value (or subarray) physically stored will be read by the CDF library and placed into the application's memory buffer three times (contiguously).

Example (FORTRAN application)

Assume a 2-dimensional CDF with sizes `2,4` and row majority, a variable with a record variance of `VARY` and dimension variances of `VARY,VARY`, and hyper read parameters as follows...

record number	`5`
record count	`2`
record interval	`1`
dimension indices	`1,1`
dimension counts	`2,4`
dimension intervals	`1,1`

The values placed in the application's buffer would be as follows (with the first value being in low memory)...

```
5(1,1) 5(1,2) 5(1,3) 5(1,4) 5(2,1) 5(2,2) 5(2,3) 5(2,4)
6(1,1) 6(1,2) 6(1,3) 6(1,4) 6(2,1) 6(2,2) 6(2,3) 6(2,4)
```

where `r(i,j)` is a physically stored value with `r` being the record number, `i` being the first dimension index, and `j` being the second dimension index (`r`, `i`, and `j` are physical record numbers and dimension indices).

If the dimension variances had been `VARY,NOVARY`, the values placed in the buffer would have been...

```
5(1,1) 5(1,1) 5(1,1) 5(1,1) 5(2,1) 5(2,1) 5(2,1) 5(2,1)
6(1,1) 6(1,1) 6(1,1) 6(1,1) 6(2,1) 6(2,1) 6(2,1) 6(2,1)
```

If the record count had been `3` and the record interval `2`, the values placed in the buffer would have been...

```
5(1,1) 5(1,2) 5(1,3) 5(1,4) 5(2,1) 5(2,2) 5(2,3) 5(2,4)
7(1,1) 7(1,2) 7(1,3) 7(1,4) 7(2,1) 7(2,2) 7(2,3) 7(2,4)
9(1,1) 9(1,2) 9(1,3) 9(1,4) 9(2,1) 9(2,2) 9(2,3) 9(2,4)
```

If the dimension counts had been `2,2` and the dimension intervals `1,2`, the values placed in the buffer would have been...

```
5(1,1) 5(1,3) 5(2,1) 5(2,3)
6(1,1) 6(1,3) 6(2,1) 6(2,3)
```

If the CDF majority had been column major, the values placed in the buffer would have been...

```
5(1,1) 5(2,1) 5(1,2) 5(2,2) 5(1,3) 5(2,3) 5(1,4) 5(2,4)
6(1,1) 6(2,1) 6(1,2) 6(2,2) 6(1,3) 6(2,3) 6(1,4) 6(2,4)
```

Had these examples been for hyper writes, the CDF library would have expected to find the values in the application's buffer exactly as they were placed there during the corresponding hyper read. In the case where the record interval was `2`, the records being skipped would be written using the variable's fill value if they did not already exist. If they did already exist, they would not be affected.

2.7.10 Sequential Access

Sequential access provides a way to sequentially read/write the values physically stored for a variable. A physical view is provided rather than the conceptual view. To use sequential access, a starting point must be selected by specifying a record number and dimension indices (which selects the current position). Each sequential read will then return the value at the current position and then automatically increment the current position to the next value. Likewise, a sequential write will store a value at the current position and then increment the current position to the next value. Sequential reads are allowed until the end of the physical values has been reached (not the end of the virtual values). Sequential reading will increment to the beginning of the next physical record if necessary. Sequential writing can be used to extend the physical values for a variable (as well as to overwrite existing values).

Example (FORTRAN application)

Assume a 2-dimensional CDF with sizes `2,3` and column majority, a variable with record variance `VARY`, dimension variances `VARY,VARY` and nine (9) physical records written, and that the current sequential position has been set to record number `7` and indices `2,2`. Consecutive sequential reads would cause the following values to be read and returned to the application...

```
                     7(2,2) 7(1,3) 7(2,3)
8(1,1) 8(2,1) 8(1,2) 8(2,2) 8(1,3) 8(2,3)
9(1,1) 9(2,1) 9(1,2) 9(2,2) 9(1,3) 9(2,3)
END_OF_VAR
```

...where r(i,j) is a physically stored value with r being the record number, i the first dimension index, and j the second dimension index (r, i, and j are physical record numbers and dimension indices). The next sequential read after the last physical value would cause a status code indicating the end of the variable to be returned (END_OF_VAR).

Had the dimension variances been NOVARY,VARY, the values read would have been...

```
       7(1,2) 7(1,3)
8(1,1) 8(1,2) 8(1,3)
9(1,1) 9(1,2) 9(1,3)
END_OF_VAR
```

Note that specifying the virtual value 7(2,2) as the current seqential position caused physical value 7(1,2) to actually be selected (because the first dimension variance is NOVARY).

2.7.11 Variable Fill Values

Variable fill values are used in several situations.

1. When the first value is written to a new record (with CDFvarPut in C, CDF_var_put in FORTRAN), the other values in that record will contain the fill value. This also applies to hyper writes (using CDFvarHyperPut in C, CDF_var_hyper_put in FORTRAN) if less than the entire record is written (the unwritten values will contain the fill value).
2. When a new record is written which is more than one record beyond the last record already written, the intervening records will contain fill values. This will only happen if the record variance is VARY (the variable values vary from record to record).
3. When reading a record beyond the last record for a variable (but within the number of records in the entire CDF), fill values will be returned.

The fill value for a variable may be specified with the Internal interface. It should be specified before any values are read from or written to the variable, otherwise the default fill value will be used. The fill value may be changed at any time (and any number of times) and will be in effect for all subsequent operations.

The default fill values for each data type are as follows.

```
CDF_BYTE        0
CDF_INT1        0
CDF_UINT1       0
CDF_INT2        0
CDF_UINT2       0
CDF_INT4        0
CDF_UINT4       0
CDF_REAL4       0.0
CDF_FLOAT       0.0
CDF_REAL8       0.0
CDF_DOUBLE      0.0
```

```
CDF_EPOCH          01-Jan-0000 00:00:00.000 (0.0)
CDF_CHAR           " " (space character, 40 octal)
CDF_UCHAR          " " (space character, 40 octal)
```

2.8 Attributes

CDF attributes are the mechanism for storing metadata (variables are used to store data). A new attribute may be created in a CDF at any time.

2.8.1 Naming

Each attribute in a CDF has a unique name. Attribute names are case sensitive regardless of the operating system being used. Attribute names may consist of any printable characters including blanks. Trailing blanks, however, are ignored when the CDF library compares attribute names. `"UNITS"` and `"UNITS   "` are considered to be the same name so they cannot both exist in the same CDF. This was done because Version 1 of CDF padded attribute names on the right with blanks out to eight characters. To allow CDF Version 2 applications to read Version 1 CDFs without having to be concerned with the trailing blanks, the trailing blanks are ignored when comparing attribute names. The trailing blanks are returned as part of the name, however, when an attribute is inquired by an application program.

2.8.2 Numbering

The attributes in a CDF are numbered consecutively starting at one (1) for FORTRAN applications and starting at zero (0) for C applications.

2.8.3 Attribute Scopes

Attribute scopes declare the intended purpose of an attribute. Global scoped attributes describe some aspect of the entire CDF. Variable scoped attributes describe some property of each variable (one entry per variable will normally exist for a variable scoped attribute).

The CDF library places no restrictions on the operations which may be performed on an attribute of a certain scope. An attribute's scope only affects the interpretation of its entries by CDF toolkit programs and user applications (eg. entries of a variable scoped attribute should correspond to variables).

CDF Version 1 did not allow the scope of an attribute to be explicitly declared. This led to ambiguities in the interpretation of attribute entries in the toolkit programs and user applications. CDF Version 2 does allow the scope of an attribute to be declared when the attribute is created. However, when the Version 2 CDF library reads a Version 1 CDF it cannot always determine the scope of an attribute. When this occurs the Version 2 CDF library makes a best guess and returns `GLOBAL_SCOPE_ASSUMED` or `VARIABLE_SCOPE_ASSUMED` as the scope of the attribute. When a user application reading a Version 1 CDF encounters an assumed scope, it should take the appropriate action to determine the correct scope. However, when the `CDFconvert` toolkit program (see Section 3.4) encounters an assumed scope it cannot always determine the correct scope. In those cases it will create the attribute in the new CDF with the assumed scope. The assumed scopes may be corrected by using the `CDFscope` toolkit program (see Section 3.5). The assumed scopes may also be corrected (in Version 2 CDFs) by using the `CDFattrCorrectScope` function within a C application or the `CDF_attr_correct_scope` function within a FORTRAN application.

2.8.4 Attribute Entries

Attribute entries are used to actually store metadata values. Each attribute in a CDF may have zero or more associated entries. For variable scoped attributes, each entry describes some property of the corresponding variable in the CDF (note that an entry does not have to exist for each variable). For global scoped attributes, the entries are independent of anything else in the CDF and have meaning only to the application.

Numbering

The entries for an attribute are numbered starting at one (1) for FORTRAN applications and starting at zero (0) for C applications. Entry numbers used need not be contiguous (as do variable and attribute numbers). An application may chose to write any combination of entries for a particular attribute (keeping in mind that the entry numbers used for a variable scope attribute correspond to the existing variables).

Data Type

Each entry for an attribute has a specified data type. The supported data types are described in Section 2.5. The entries for an attribute may have any combination of data types.

Number of Elements

In addition to a data type, each attribute entry has a number of elements. This refers to the number of elements of the specified data type.

For character string data types this is the number of characters in the string. For example, if the value for a global scope attribute named `TITLE` were `"Example CDF Title."` (not including the double quotes), the data type would be `CDF_CHAR` and the number of elements would be 18 (a character string of size 18).

For non-character string data types this is the number of elements in an array of the data type. For example, if the value of a variable scope attribute named `RANGE` were `100.0,900.0`, the data type would be `CDF_REAL4` and the number of elements would be two (an array of two values). Note that the number of elements must be one (1) for variables with a non-character string data type.

2.9 NSSDC Standard

CDFs generated for use by NSSDC applications (eg. the NSSDC Graphics System, NGS) are expected to conform to a specified standard. These applications expect certain attributes and make use of several optional attributes and variables to provide additional features.

2.9.1 Standard Attributes

There are several attributes which are required to exist in a CDF if it is to conform to the NSSDC standard. These required attributes are shown in Table 2.3. Several optional attributes are also shown.

The purpose of each attribute is defined as follows...

`TITLE` A title for the CDF data set. Only one entry should be written. (entry number 0 for C applications, entry number 1 for FORTRAN applications).

`FIELDNAM` The field name for a variable. One entry per variable should be written.

`VALIDMIN` The minimum valid value for a variable. Values below this minimum are treated as invalid or fill data. One entry per variable should be written.

`VALIDMAX` The maximum valid value for a variable. Values above this maximum are treated as invalid or fill data. One entry per variable should be written.

`UNITS` The units of measurement for a variable. One entry per variable should be written.

`FORMAT` The FORTRAN format specification to use when displaying the values for a variable. One entry per variable should be written.

`TEXT` Documentation text about the CDF. Any number of entries may be written.

`MODS` A modification history for the CDF. Each time the CDF is modified a new entry would be added with an explanation of the changes.

`SCALEMIN` The scale minimum for a variable. This is used when scaling the variable's values for display. If omitted, the `VALIDMIN` attribute entry will be used. One entry per variable should be written.

`SCALEMAX` The scale maximum for a variable. This is used when scaling the variable's values for display. If omitted, the `VALIDMAX` attribute entry will be used. One entry per variable should be written.

`MONOTON` The monotonicity of a variable. This specifies if the values strictly increase (`"INCREASE"`), strictly decrease (`"DECREASE"`), or are not monotonic (`"FALSE    "`) along a dimension or from record to record. Specifying a variable's monotonicity can greatly improve the efficiency of certain NSSDC applications (namely the NSSDC Graphics System). One entry per variable should be written.

`CATDESC` A catalog description for a variable. This is used for cataloging purposes by some NSSDC applications. One entry per variable should be written.

`FILLVAL` The fill value for a variable. This value should not be confused with the fill value that may be specified for a variable (with the Internal interface) which the CDF library actually uses. The fill values should agree but this one is not used by the CDF library in any way. It is used only by CDF applications. One entry per variable should be written.

2.9.2 Standard Variables

There are no required CDF variables. The first eight characters of variable names must, however, be unique for NSSDC applications. Names longer than eight characters will be truncated when displayed.

The optional variable **EPOCH** is used by many NSSDC applications to store date/time values. Section 2.5.4 defines an EPOCH value. The record variance should be **VARY**, each dimension variance should be **NOVARY**, the data type should be **CDF_EPOCH**, and the number of elements should be one (1). If a CDF contains an **EPOCH** variable, the following variable names should not be used: **TIME**, **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **MSEC**, **IYEAR**, **IMONTH**, **IDAY**, **IHOUR**, **IMINUTE**, **ISECOND**, and **IMSEC**.

Callable from C	Callable from FORTRAN	Purpose
CDFcreate()	CDF_create()	Creates a new CDF.
CDFopen()	CDF_open()	Opens an existing CDF.
CDFdoc()	CDF_doc()	Inquires version/release and copyright notice.
CDFinquire()	CDF_inquire()	Inquires dimensionality, etc.
CDFclose()	CDF_close()	Closes a CDF.
CDFdelete()	CDF_delete()	Deletes a CDF.
CDFerror()	CDF_error()	Inquires error (status) code meaning.
CDFvarCreate()	CDF_var_create()	Creates a variable.
CDFvarNum()	CDF_var_num()	Determines a variable number.
CDFvarRename()	CDF_var_rename()	Renames a variable.
CDFvarInquire()	CDF_var_inquire()	Inquires about a variable.
CDFvarPut()	CDF_var_put()	Writes a variable value.
CDFvarGet()	CDF_var_get()	Reads a variable value.
CDFvarHyperPut()	CDF_var_hyper_put()	Writes more than one variable value.
CDFvarHyperGet()	CDF_var_hyper_get()	Reads more than one variable value.
CDFattrCreate()	CDF_attr_create()	Creates an attribute.
CDFattrNum()	CDF_attr_num()	Determines an attribute number.
CDFattrRename()	CDF_attr_rename()	Renames an attribute.
CDFattrInquire()	CDF_attr_inquire()	Inquires about an attribute.
CDFattrEntryInquire()	CDF_attr_entry_inquire()	Inquires about an attribute entry.
CDFattrPut()	CDF_attr_put()	Writes an attribute entry.
CDFattrGet()	CDF_attr_get()	Reads an attribute entry.

Table 2.1: Standard Interface Routines

Callable from C	Callable from FORTRAN	Purpose
CDFlib ()	CDF_lib ()	Performs all available operations.

Table 2.2: Internal Interface Routines

	Name	Data Type	Number of Elements
Required Global Scope Attributes:			
	`TITLE`	Character (CDF_CHAR)	50
Required Variable Scope Attributes:			
	`FIELDNAM`	Character (CDF_CHAR)	20
	`VALIDMIN`	(same as variable)	(same as variable)
	`VALIDMAX`	(same as variable)	(same as variable)
	`UNITS`	Character (CDF_CHAR)	20
	`FORMAT`	Character (CDF_CHAR)	8
Optional Global Scope Attributes:			
	`TEXT`	Character (CDF_CHAR)	80
	`MODS`	Character (CDF_CHAR)	80
Optional Variable Scope Attributes:			
	`SCALEMIN`	(same as variable)	(same as variable)
	`SCALEMAX`	(same as variable)	(same as variable)
	`MONOTON`[a]	Character (CDF_CHAR)	8
	`CATDESC`	Character (CDF_CHAR)	40
	`FILLVAL`	(same as variable)	(same as variable)

Table 2.3: NSSDC Standard CDF Attributes

[a] We strongly suggest using the `MONOTON` attribute.

Chapter 3

CDF Toolkit Reference

3.1 Introduction

The CDF toolkit is a set of utility programs which allow the creation, access, and modification of CDFs. Each program is executed at the command line (or may be executed from within your applications using the methods provided by the operating system being used). The following rules apply to the command line syntax.

1. Parameters are required unless noted otherwise. Parameters are shown in `<>`s.
2. All qualifiers are optional.
3. Qualifiers can be abbreviated as long as the abbreviation is unique.
4. Optional parts of a command are shown in `[]`s.
5. Wildcard characters are allowed in CDF names to allow more than one CDF to be specified (where appropriate). Wildcard characters may be used in the CDF name but not the directory path portion of a specification. The wildcard characters supported are similar to those avaiable on the operating system being used.

   NOTE: On UNIX systems, if a CDF specification is to contain a wildcard character, the entire specification must be enclosed in single quote marks (eg. `'/disk3/sst*'`).
6. On VMS systems, qualifiers begin with a '/'. On UNIX and MS-DOS systems, qualifiers begin with a '-'. When describing qualifiers, both notations may be shown (eg. `/[NO]LOG`, `-[no]log`). Use only the notation appropriate for the operating system begin used.

If a program allows message logging to be disabled (using `/NOLOG`,`-nolog`), INFO and WARNING messages (from the CDF library) will not be displayed but ERROR messages will always be displayed.

3.2 CDFdir

3.2.1 Introduction

The `CDFdir` utility is used to display a directory listing of a CDF's files. The `.cdf` file is displayed first followed by the variable files (if they exist for a multi-file CDF) in numerical order.

3.2.2 Executing the CDFdir program

The command line syntax for `CDFdir` is as follows.

Syntax:

VMS:

```
$ CDFDIR <cdf-path>
```

UNIX:

```
% cdfdir <cdf-path>
```

MS-DOS:

```
> cdfdir <cdf-path>
```

Parameter(s):

`<cdf-path>`	The pathname of the CDF for which to display a directory listing (do not specify an extension).

Example(s):

VMS:

```
$ CDFDIR NCDS$DATA:GISS_WETL_CLIMATOLOGY
$ CDFDIR [-.TEMP]FGGE3B
```

UNIX:

```
% cdfdir ../cac_sst_blended
% cdfdir ~/CDFs/giss_wetl_climatology
```

MS-DOS:

```
> cdfdir ..\cac_sst
> cdfdir c:\cdfs\gisswetl
```

Help is displayed when `CDFdir` is executed without any arguments.

3.2.3 Output from the CDFdir program

The format of the output from `CDFdir` is that of a directory listing on the operating system being used.

3.3 CDFcompare

3.3.1 Introduction

The CDFcompare program displays the differences between two CDFs. More than one pair of CDFs can be compared. This program would be used to verify changes made to a CDF (comparing it with the original) or to verify the conversions performed by `CDFconvert` (see Section 3.4).

3.3.2 Executing the CDFcompare program

The command line syntax for `CDFcompare` is as follows.

Usage:

VMS:

```
$ CDFCOMPARE [/[NO]LOG] [/[NO]ATTR] [/[NO]VAR] [/[NO]NUMBER]
             <cdf-spec-1> <cdf-spec-2>
```

UNIX:

```
% cdfcompare [-[no]log] [-[no]attr] [-[no]var] [-[no]number]
             <cdf-spec-1> <cdf-spec-2>
```

MS-DOS:

```
> cdfcompare [-[no]log] [-[no]attr] [-[no]var] [-[no]number]
             <cdf-spec-1> <cdf-spec-2>
```

Parameter(s):

```
<cdf-spec-1> <cdf-spec-2>
```

The specifications of the CDFs to be compared (do not enter extensions). These can be either a pathname specifying a single CDF or a directory/wildcard path specifying more than one CDF. Wildcards are allowed in the CDF name but not in the directory path.

If two directory/wildcard paths are specified, all of the CDFs with matching names will be compared. If a CDF pathname and a directory/wildcard path are specified, the CDF specified will be compared with the CDF in the directory/wildcard path having the same name. If two CDF pathnames are specified, the CDFs are compared (this is the only way to compare two CDFs having different names).

Qualifier(s):

`/[NO]LOG`, `-[no]log` Specifies whether or not messages about the progress of each comparison are displayed. Message logging is disabled by default.

`/[NO]ATTR`, `-[no]attr` Specifies whether or not attributes (and their entries) are to be compared. Attributes are compared by default.

`/[NO]VAR`, `-[no]var` Specifies whether or not variables are to be compared. Variables are compared by default.

`/[NO]NUMBER`, `-[no]number`

Specifies whether or not numbering differences between attributes with the same names and between variables with the same names are to be displayed. Numbering differences are not displayed by default.

Example(s):

VMS:

```
$ CDFCOMPARE GISS_WETL GISS_WETL1
$ CDFCOMPARE/LOG/NOATTR/NUMBER GISS_WETL CDF$SMPL:GISS_WETL
$ CDFCOMPARE/NOVAR NCDS$SMPL: NCDS$DATA:
```

UNIX:

```
% cdfcompare giss_wetl giss_wetl1
% cdfcompare -log -noattr -number giss_wetl ../../giss_wetlx
% cdfcompare -novar /user5/CDFs /user6/CDFs
```

MS-DOS:

```
> cdfcompare gisswetl c:\gisswetl
> cdfcompare -log -noattr -number gisswetl ..\..\giswetlx
> cdfcompare -novar a:\cdfs c:\cdf\cdfs
```

Help is displayed when `CDFcompare` is executed without any arguments.

3.3.3 Output from the CDFcompare program

The output from `CDFcompare` consists of messages indicating the differences found. If message logging is enabled, the progress of each comparison is also displayed.

3.4 CDFconvert

3.4.1 Introduction

The `CDFconvert` program is used to convert various properties of a CDF. In all cases new CDFs are created (existing CDFs are not modified). Any combination of the following properties may be changed when converting a CDF.

1. A CDF created with an earlier release of the CDF library may be converted to the current library release.
2. The format of the CDF may be changed (see Section 2.3).
3. The data encoding of the CDF may be changed (see Section 2.4).
4. The variable majority of the CDF may be changed (see Section 2.7.8).

3.4.2 Executing the CDFconvert program

The command line syntax for `CDFconvert` is as follows.

Usage:

VMS:

```
$ CDFCONVERT [/SKELETON=<cdf-path>] [/[NO]LOG]
             <src-cdf-spec> <dst-cdf-spec>
             [/SINGLE or /MULTI] [/ROW or /COLUMN] [/HOST or /NETWORK]
```

UNIX:

```
% cdfconvert [-skeleton <cdf-path>] [-[no]log]
             <src-cdf-spec> <dst-cdf-spec>
             [-single or -multi] [-row or -column] [-host or -network]
```

MS-DOS:

```
> cdfconvert [-skeleton <cdf-path>] [-[no]log]
             <src-cdf-spec> <dst-cdf-spec>
             [-single or -multi] [-row or -column] [-host or -network]
```

Parameter(s):

`<src-cdf-spec>` The source CDF(s). This can be either a single CDF pathname or a directory/wildcard path in which case all CDFs which match the specification will be converted. Wildcards are allowed in the CDF name but not in the directory path. In either case, do not specify an extension.

`<dst-cdf-spec>` The destination of the converted CDF(s). This may be a single CDF pathname only if `<src-cdf-spec>` is a single CDF pathname. If the directory paths are the same, then a different CDF name must be specified. If `<src-cdf-spec>` is a directory/wildcard path, then this must be a directory path (other than the source directory path). This may also be a directory path if only a single CDF is being converted. In any case, do not specify an extension.

Qualifier(s):

`/SKELETON=<cdf-path>`, `-skeleton <cdf-path>`

`<cdf-path>` is the pathname of a skeleton CDF to be used during the conversions (do not enter an extension). The skeleton CDF is used in the following cases.

1. If an assumed attribute scope is found in a source CDF, then the scope of that attribute in the skeleton CDF (if the attribute exists) will be used in the destination CDF.
2. If a format for the destination CDF was not specified, then the format of the skeleton CDF will be used.
3. If a majority for the destination CDF was not specified, then the majority of the skeleton CDF will be used.
4. If an encoding for the destination CDF was not specified, then the encoding of the skeleton CDF will be used.

Specifying a skeleton CDF is optional.

`/[NO]LOG`, `-[no]log` Specifies whether or not messages about the progress of each conversion are displayed. Message logging is disabled by default.

`/SINGLE`, `-single` or `/MULTI`, `-multi`

The format of the destination CDF(s). This overrides the format of the skeleton CDF (if specified). If neither this qualifier nor a skeleton CDF is specified, then the format of the destination CDF(s) will be the same as that of the source CDF(s).

`/ROW`, `-row` or `/COLUMN`, `-column`

The majority of the destination CDF(s). This overrides the majority of the skeleton CDF (if specified). If neither this qualifier nor a skeleton CDF is specified, then the majority of the destination CDF(s) will be the same as that of the source CDF(s).

/HOST, -host or /NETWORK, -network

The encoding of the destination CDF(s). This overrides the encoding of the skeleton CDF (if specified). If neither this qualifier nor a skeleton CDF is specified, then the encoding of the destination CDF(s) will be the same as that of the source CDF(s).

Example(s):

VMS:

```
$ CDFCONVERT CDF$SMPL:TEMPLATE0 TEMPLATE0X
$ CDFCONVERT/LOG CDF$SMPL: USER_DISK:[USER.CDF]
$ CDFCONVERT CAC_SST_BLENDED CAC_SST_BLENDEDX/SINGLE/NETWORK
$ CDFCONVERT/SKELETON=CDF$SMPL:TEMPLATE3 CAC_SST_BLENDED* [USER.CDF]
```

UNIX:

```
% cdfconvert ../samples/template0 template0x
% cdfconvert -log ../samples /disk4/user/cdf
% cdfconvert cac_sst_blended cac_sst_1 -single -network
% cdfconvert -skeleton template3 '../cdf/cac_sst*' ~user/cdf
```

MS-DOS:

```
> cdfconvert ..\samples\tplate0 tplate0x
> cdfconvert -log ..\samples c:\dir4\user\cdf
> cdfconvert cac_sst cac_sst1 -single -network
> cdfconvert -skeleton tplate3 a:\cdf\cac_sst dir5\cdf
```

Help is displayed when CDFconvert is executed without any arguments.

3.4.3 Output from the CDFconvert program

As CDFconvert executes, the name of each CDF being converted is displayed. If message logging is enabled, the progress of each conversion is also displayed.

3.5 CDFscope

3.5.1 Introduction

The CDFscope program is used to correct assumed attribute scopes found in a CDF. Section 2.8.3 explains attribute scopes and how assumed scopes can arise.

3.5.2 Executing the CDFscope program

The command line syntax for `CDFscope` is as follows.

Usage:

VMS:

```
$ CDFSCOPE [/[NO]CONFIRM] [/[NO]LOG] <cdf-path>
```

UNIX:

```
% cdfscope -[no]confirm -[no]log <cdf-path>
```

MS-DOS:

```
> cdfscope -[no]confirm -[no]log <cdf-path>
```

Parameter(s):

`<cdf-path>` The pathname of the CDF to be corrected (do not specify an extension).

Qualifier(s):

`/[NO]CONFIRM`, `-[no]confirm`

Specifies whether or not `CDFscope` prompts for confirmation before correcting a scope. The valid responses to the confirmation prompts are as follows.

`Y/y`	Yes, correct the scope (assumed to definite).
`N/n`	No, don't correct the scope.
`G/g`	Correct the scope to global.
`V/v`	Correct the scope to variable.

Confirming is disabled by default (all assumed scopes are corrected to the corresponding definite scopes).

`/[NO]LOG`, `-[no]log` Specifies whether or not message logging is enabled. Message logging is disabled by default.

Example(s):

VMS:

```
$ CDFSCOPE DUA5:[CDF.TEST]TOGA_ECMWF
$ CDFSCOPE/CONFIRM/NOLOG TOGA_ECMWF
```

UNIX:

```
% cdfscope /disk5/cdf/toga_ecmwf
% cdfscope -confirm -nolog toga_ecmwf
```

MS-DOS:

```
> cdfscope a:\dir5\cdf\toga
> cdfscope -confirm -nolog toga
```

Help is displayed when `CDFscope` is executed without any arguments.

3.5.3 Output from the CDFscope program

If message logging is enabled, the number of scopes corrected will be displayed.

3.6 CDFstats

3.6.1 Introduction

The `CDFstats` program produces a statistical report on a CDF's variable data. For each variable, it determines the actual minimum and maximum values (in the entire CDF), the minimum and maximum values within the valid range (if the `VALIDMIN` and `VALIDMAX` attributes and corresponding entries are present in the CDF), and the monotonicity. An option exists to allow fill values (specified by the `FILLVAL` attribute) to be filtered out when collecting statistics.

Monotonicity refers to whether or not variable data values increase or decrease from record to record or along a dimension. This property is checked only if the variable varies along just one "dimension" (considering records to be another "dimension"). For example, consider a 2-dimensional CDF with the following variables.

Variable	Record Variance	Dimension Variances
`EPOCH`	`VARY`	`NOVARY,NOVARY`
`LATITUDE`	`NOVARY`	`VARY,NOVARY`
`LONGITUDE`	`NOVARY`	`NOVARY,VARY`
`ELEVATION`	`NOVARY`	`VARY,VARY`
`TEMPERATURE`	`VARY`	`VARY,VARY`

The `EPOCH`, `LATITUDE`, and `LONGITUDE` variables would be checked for monotonicity but the `ELEVATION` and `TEMPERATURE` variables would not be checked.

3.6.2 Executing the CDFstats program

The command line syntax for `CDFstats` is as follows.

Usage:

VMS:

```
$ CDFSTATS [/[NO]RANGE] [/[NO]FILL] [/OUTPUT=<file-path>] <cdf-path>
```

UNIX:

```
% cdfstats [-[no]range] [-[no]fill] [-output <file-name>] <cdf-path>
```

MS-DOS:

```
> cdfstats [-[no]range] [-[no]fill] [-output <file-name>] <cdf-path>
```

Parameter(s):

`<cdf-path>` The pathname of the CDF to check (do not specify an extension).

Qualifier(s):

`/[NO]RANGE`, `-[no]range` Specifies whether or not range checking will be performed. To perform range checking, the CDF must contain `VALIDMIN` and `VALIDMAX` attributes. A variable must also have an entry for each of these attributes in order for range checking to be performed on that variable. Range checking is enabled by default (unless your installation of CDF does not conform to the NSSDC standard).

`/[NO]FILL`, `-[no]fill` Specifies whether or not fill values are to be filtered out when collecting statistics. To filter out fill values, the CDF must contain the `FILLVAL` attribute. A variable must also have an entry for the `FILLVAL` attribute in order for fill value filtering to be performed on that variable. Fill value filtering is enabled by default (unless your installation of CDF does not conform to the NSSDC standard).

`/OUTPUT=<file-path>`, `-output <file-path>`

Redirects the output to a file. The file created will be named `<file-path>` (if `<file-path>` does not have an extension, `.sts` is appended automatically). If this qualifier is not specified, the output is displayed at the terminal.

Example(s):

VMS:

```
$ CDFSTATS GISS_SOIL
$ CDFSTATS/NOFILL/OUTPUT=TEMPLATE3/NORANGE CDF$SMPL:TEMPLATE3
```

UNIX:

```
% cdfstats giss_soil
% cdfstats -range -fill $CDF_SMPL/giss_soil
% cdfstats -norange -output template3 ../../samples/template3
```

MS-DOS:

```
> cdfstats gisssoil
> cdfstats -range -nofill a:\cdfs\gisssoil
> cdfstats -norange -output tplate3 ..\..\samples\tplate3
```

Help is displayed when `CDFstats` is executed without any arguments.

3.6.3 Output from the CDFstats program

The format of the output from `CDFstats` is as follows.

For each variable,

```
<var-number>. <var-name> <rec-variance>/<dim-variances> (<var-datatype>)

           min: <min-value>
  min in range: <min-value-in-range>
     valid min: <valid-min>, <low-values> low value(s)

           max: <max-value>
  max in range: <max-value-in-range>
     valid max: <valid-max>, <high-values> high value(s)

    fill value: <fill-value>, <fill-values> fill value(s)

     monotonic: <monotonicity>
```

If range checking and/or fill value filtering are disabled, the corresponding fields will not be displayed. The fields are defined as follows...

`<var-number>`	The variable number.
`<var-name>`	The variable name.
`<rec-variance>`	The record variance of the variable — either a `T` or `F`.
`<dim-variances>`	The dimension variances of the variable — for each dimension either a `T` or `F`.
`<var-datatype>`	The data type of the variable (eg. `_REAL4`).

`<min-value>`	The minimum value found (regardless of any range checking performed).
`<min-value-in-range>`	The minimum value found within the valid range.
`<valid-min>`	The minimum valid value (`VALIDMIN` attribute entry value).
`<low-values>`	The number of values found that are less than the valid minimum.
`<max-value>`	The maximum value found (regardless of any range checking performed).
`<max-value-in-range>`	The maximum value found with the valid range.
`<valid-max>`	The maximum valid value (`VALIDMAX` attribute entry value).
`<high-values>`	The number of values found that are greater than the valid maximum.
`<fill-value>`	The fill value (`FILLVAL` attribute entry value).
`<fill-values>`	The number of fill values found.
`<monotonicity>`	The monotonicity of the variable.

The `<monotonicity>` field may take on one of the following values.

`Steady (one value)`	The variable has only one value in the CDF.
`Steady (all values the same)`	All values of the variable are the same.
`Increase`	Values strictly increase (with increasing record number/dimension index).
`Decrease`	Values strictly decrease (with increasing record number/dimension index).
`noDecrease (some values the same)`	Consecutive values either increase or are the same (with increasing record number/dimension index).
`noIncrease (some values the same)`	Consecutive values either decrease or are the same (with increasing record number/dimension index).
`False`	Consecutive values both increase and decrease.
`n/a`	The variable was not checked for monotonicity because it varys along more than one "dimension" (if records are considered another "dimension").

3.7 CDFinquire

3.7.1 Introduction

The `CDFinquire` program produces a report on a CDF's contents. It may be used to inquire the following.

1. The version, release, and increment numbers of the creating CDF library.

2. The copyright notice of the creating CDF library.

3. Header information — the CDF's data encoding, variable majority, number of records, number of global scoped and variable scoped attributes, number of variables, number of dimensions, and dimension sizes.

4. Global scope attribute definitions — the name, number, and number of entries for each global scope attribute.

5. Global scope attribute entries — for each global scope attribute entry, the data type, number of elements, and value(s).

6. Variable scope attribute definitions — the name, number, and number of entries for each variable scope attribute.

7. Variable definitions — the name, number, data type, number of elements, record variance, and dimension variances for each variable.

8. Variable scope attribute entries — for each variable scope attribute entry, the data type, number of elements, and value(s).

3.7.2 Executing the CDFinquire program

The command line syntax for `CDFinquire` is as follows.

Usage:

VMS:

```
$ CDFINQUIRE [/DISPLAY=<mask>] [/OUTPUT=<file-path>] [/ID] <cdf-path>
```

UNIX:

```
% cdfinquire [-display <mask>] [-output <file-path>] [-id] <cdf-path>
```

MS-DOS:

```
> cdfinquire [-display <mask>] [-output <file-path>] [-id] <cdf-path>
```

Parameter(s):

`<cdf-path>`	The pathname of the CDF to inquire (do not specify an extension).

Qualifier(s):

`/DISPLAY=<mask>`, `-display <mask>`

The type of inquiry to be done. The `<mask>` consists of one or more of the following.

`i`	CDF copyright and version/release/increment.
`h`	Header information.
`g`	Global scope attribute definitions.
`g+`	Global scope attribute definitions & entries.
`a`	Variable scope attribute definitions.
`v`	Variable definitions.
`v+`	Variable definitions & variable scope attribute entries.

Do not specify both `g` and `g+` or both `v` and `v+`. If this qualifier is not specified, the default `<mask>` is `ihg+av+`.

`/OUTPUT=<file-path>`, `-output <file-path>`

Redirects the output to a file. The file created will be named `<file-path>` (if `<file-path>` does not have an extension, `.inq` is appended automatically). If this qualifier is not specified, the output is displayed at the terminal.

`/ID`,`-id` Causes the version of the CDF distribution (not the CDF data set) to be displayed. All other parameters/qualifiers are ignored.

Example(s):

VMS:

```
$ CDFINQUIRE GISS_VEG
$ CDFINQUIRE/DISPLAY=ihv/OUTPUT=GISS_VEG GISS_VEG
```

UNIX:

```
% cdfinquire giss_veg
% cdfinquire -display ihv -output giss_veg giss_veg
```

MS-DOS:

```
> cdfinquire giss_veg
> cdfinquire c:\cdfs\giss\giss_veg
> cdfinquire -display ihv -output ..\giss_veg giss_veg
```

Help is displayed when `CDFinquire` is executed without any arguments.

3.7.3 Output from the CDFinquire program

The format of the output from `CDFinquire` is as follows (assuming a display mask of `ihg+av+`).

```
INFORMATION BLOCK
-----------------

<copyright>

Created by CDF library version: <version>.<release>.<increment>


HEADER BLOCK
------------

  CDF name..... <cdf-name>
  Encoding..... <encoding>
  Majority..... <majority>
  Format....... <format>
  Records...... <num-records>
  Attributes... <num-attributes> (<num-global-scope> global scope,
                                  <num-variable-scope> variable scope)
  Variables.... <num-variables>
  Dimensions... <num-dimension>, sizes: <dimension-sizes>


GLOBAL SCOPE ATTRIBUTES
-----------------------

  <attr-num>.  Name...... <attr-name>
               Entries... <num-entries>

                  Number...... <entry-num>
                  Data type... <entry-data-type> (<entry-num-elems> elements)
                  Value....... <entry-value(s)>

                     .
                     .
                     .

      .
      .
      .


VARIABLE SCOPE ATTRIBUTES
-------------------------

  <attr-num>.  Name...... <attr-name>
               Entries... <num-entries>
       .
       .
       .
```

```
VARIABLES
---------

  <var-num>.  Name........ <var-name>
              Data type... <var-data-type> (<var-num-elems> elements)
              Variances... <rec-variance>/<dim-variances>

                 Attribute name... <attr-name>
                 Data type........ <entry-data-type> (<entry-num-elems>
                                                              elements)
                 Value............ <entry-value(s)>

                    .
                    .
                    .

    .
    .
    .
```

Each field is described as follows.

`<copyright>`	The CDF copyright notice.
`<version>`	The version number of the CDF library which created/modified the CDF.
`<release>`	The release number of the CDF library which created/modified the CDF.
`<increment>`	The increment number of the CDF library which created/modified the CDF.
`<attr-name>`	The attribute name.
`<attr-num>`	The attribute number.
`<cdf-name>`	The name of the CDF.
`<dim-variances>`	The variable dimension variances. For each dimension, a `T` if values vary along that dimension (`VARY`) or an `F` if they do not (`NOVARY`). Dimension variances are described in Section 2.7.7. For 0-dimensional CDFs, this field is omitted.
`<dimension-sizes>`	The dimension sizes for the CDF. For 0-dimensional CDFs, this field is omitted.
`<encoding>`	The data encoding of the CDF. Data encodings are described in Section 2.4.
`<entry-data-type>`	The attribute entry data type. Data types are described in Section 2.5.
`<entry-num>`	The attribute entry number.

`<entry-num-elems>`	The number of elements of the data type. The number of elements for an attribute entry is explained in Section 2.8.4.
`<entry-value(s)>`	The attribute entry value(s).
`<format>`	The format of the CDF. CDF formats are described in Section 2.3.
`<majority>`	The variable majority of the CDF. Variable majorities are described in Section 2.7.8.
`<num-attributes>`	The number of attributes in the CDF.
`<num-dimensions>`	The number of dimensions for the CDF.
`<num-entries>`	The number of entries for the attribute.
`<num-records>`	The number of records in the CDF.
`<num-global-scope>`	The number of global scope attributes in the CDF.
`<num-variable-scope>`	The number of variable scope attributes in the CDF.
`<num-variables>`	The number of variables in the CDF.
`<var-data-type>`	The variable data type. Data types are described in Section 2.5.
`<rec-variance>`	The variable record variance. A `T` if values do vary from record to record (`VARY`) or an `F` if they do not (`NOVARY`). Record variances are described in Section 2.7.6.
`<var-name>`	The variable name.
`<var-num>`	The variable number.
`<var-num-elems>`	The number of elements of the data type. The number of elements for a variable is explained in Section 2.7.5.

If a display mask specifying less than `ihg+av+` is used, the information not requested will not appear in the output.

3.8 SkeletonTable

3.8.1 Introduction

The `SkeletonTable` program is used to create an ASCII text file called a skeleton table containing information on a given CDF. It reads a CDF and writes into the skeleton table (and optionally another file called an NRV file) the following information about a CDF.

1. Data encoding, majority, and format.
2. Number of dimensions and dimension sizes.
3. Global scope attribute definitions (with entry values).

4. Variable definitions and variable scope attribute definitions (with entry values),
5. Non-record-variant (NRV) data for all of the CDF's NRV variables (if requested). The NRV data values may also be written to another file called an NRV file. NRV variables are described in Section 2.7.6

The above information is written in a format which can be "understood" by the `CDFskeleton` program (see Section 3.9). `CDFskeleton` reads a skeleton table and creates a new CDF.

3.8.2 Executing the SkeletonTable program

The command line syntax for `SkeletonTable` is as follows.

Usage:

VMS:

```
$ SKELETONTABLE [/SKELETON=<skeleton-path>] [/[NO]LOG]
                [/NONRV or /NRVFILE or /NRVTABLE] <cdf-path>
```

UNIX:

```
% skeletontable [-skeleton <skeleton-path>] [-[no]log]
                [-noNRV or -NRVfile or -NRVtable] <cdf-path>
```

MS-DOS:

```
> skeletontable [-skeleton <skeleton-path>] [-[no]log]
                [-noNRV or -NRVfile or -NRVtable] <cdf-path>
```

Parameter(s):

`<cdf-path>` The pathname of the CDF to read (do not enter an extension). It is from this CDF that the skeleton table will be created.

Qualifier(s):

`/SKELETON=<skeleton-path>`, `-skeleton <skeleton-path>`

`<skeleton-path>` is the pathname of the skeleton table to be created. Any valid pathname may be specified (do not enter an extension, `.skt` is appended automatically). If this qualifier is not specified, the default skeleton table name is `<cdf-name>.skt` in the default/current directory (where `<cdf-name>` is the name portion of `<cdf-path>`).

`/NONRV`, `-noNRV` or `/NRVTABLE`, `-NRVtable` or `/NRVFILE`, `-NRVfile`

No more than one of these qualifiers may be specified. The meaning of each is as follows.

`/NONRV`, `-noNRV`

Ignore NRV data (the values are not placed in the skeleton table or NRV file).

`/NRVTABLE`, `-NRVtable`

Put NRV data values in the skeleton table

`/NRVFILE`, `-NRVfile`

Put NRV data values in the NRV file. If the skeleton table pathname was explicitly named (using `/SKELETON`, `-skeleton`), the name of this file will be `<skeleton-path>.nrv`. Otherwise, it will be `<cdf-name>.nrv` in the default/current directory (where `<cdf-name>` is the name portion of `<cdf-path>`).

`/[NO]LOG`, `-[no]log` Specifies whether or not messages are displayed as the program executes. Message logging is disabled by default.

Example(s):

VMS:

```
$ SKELETONTABLE/NOLOG FGGE3B
$ SKELETONTABLE/SKELETON=FGGE3B/noNRV FGGE3B
```

UNIX:

```
% skeletontable -nolog fgge3b
% skeletontable -skeleton fgge3b -noNRV ../cdfs/fgge3b
```

MS-DOS:

```
> skeletontable -nolog fgge3b
> skeletontable -skeleton fgge3b -noNRV c:\temp\fgge3b
```

Help is displayed when `SkeletonTable` is executed without any arguments.

3.8.3 Output from the SkeletonTable program

The format of the skeleton table and NRV file are described in Appendix A.

3.9 CDFskeleton

3.9.1 Introduction

The `CDFskeleton` program is used to make a fully structured CDF by reading an ASCII text file called a skeleton table (an optional file called an NRV file may also exist). The skeleton table contains the information necessary to create a CDF which is complete in all respects except for record-variant (RV) variable values (RV variables vary from record to record). RV values are then written to the CDF by execution of an application program. The `CDFskeleton` program allows a CDF to be created with the following.

1. The necessary header information — the number of dimensions, dimension sizes, format, data encoding, and variable majority.
2. The global scoped attribute definitions and any number of entries for each.
3. The variable definitions.
4. The variable scoped attribute definitions and the entries corresponding to each variable.
5. The data values for those variables that are non-record-variant (NRV). NRV variables do not vary from record to record. These NRV data values may also be contained in a separate file called an NRV file.

The created CDF is often referred to as a skeleton CDF.

3.9.2 Executing the CDFskeleton program

The command line syntax for the `CDFskeleton` program is as follows.

Usage:

VMS:

```
$ CDFSKELETON [/CDF=<cdf-path>] [/[NO]LOG] [/[NO]DELETE] <skeleton-path>
```

UNIX:

```
% cdfskeleton [-cdf <cdf-path>] [-[no]log] [-[no]delete] <skeleton-path>
```

MS-DOS: UNIX:

```
> cdfskeleton [-cdf <cdf-path>] [-[no]log] [-[no]delete] <skeleton-path>
```

Parameter(s):

`<skeleton-path>` The pathname of the skeleton table (do not specify an extension). This skeleton table will be read and its information used to create a new CDF.

Qualifier(s):

`/CDF=<cdf-path>`, `-cdf <cdf-path>`

`<cdf-path>` is the pathname of the CDF that will be created (overriding the pathname in the skeleton table). If this qualifier is not specified, the CDF pathname in the skeleton table is used. Do not specify an extension in the pathname.

`/[NO]LOG`, `-[no]log` Specifies whether or not messages are displayed as the program executes. Message logging is disabled by default.

`/[NO]DELETE`, `-[no]delete`

Specifies whether or not the CDF will be deleted first if it already exists (essentially overwriting it). By default, the CDF is not deleted.

Example(s):

VMS:

```
$ CDFSKELETON FGGE3B
$ CDFSKELETON/NOLOG/CDF=[-.TEMP]FGGE3B_X FGGE3B
```

UNIX:

```
% cdfskeleton fgge3b
% cdfskeleton -nolog -cdf ../fgge3b_x fgge3b
```

MS-DOS:

```
> cdfskeleton fgge3b
> cdfskeleton -nolog -cdf ..\fgge3b_x a:\fgge3b
```

Help is displayed when `CDFskeleton` is executed without any arguments.

3.9.3 Creating the skeleton table

A skeleton table is an ASCII text file having `.skt` as a file extension. CDF V2.0 skeleton tables had file extensions of `.skeleton_table`. They are still readable by `CDFskeleton`. `CDFskeleton` first looks for the specified skeleton table with `.skt` as the extension. If not found, it then looks for `.skeleton_table` as the extension. `SkeletonTable` now generates skeleton tables with the extension `.skt`.

Likewise, CDF V2.0 NRV files had file extensions of `.nrv_data`. They also are still readable by `CDFskeleton`. `CDFskeleton` first looks for the specified NRV file with `.nrv` as the extension. If not found, it then looks for `.nrv_data` as the extension. `SkeletonTable` now generates NRV files with the extension `.nrv`.

The normal method of creating and using a skeleton table (and optional NRV file) would be to use `SkeletonTable` on an existing CDF which is similar to the CDF you want to create. Then edit the created skeleton table (and optional NRV file) to meet your needs and use `CDFskeleton` to create the new CDF. The skeleton table (and optional NRV file) could also be created from scratch with any text editor.

The format of the skeleton table and NRV file are described in Appendix A.

3.10 CDFbrowse

3.10.1 Introduction

The `CDFbrowse` program allows the user to browse through a CDF's metadata and data. More specifically, it allows the user to browse the attribute definitions, variable definitions, attribute entry values, and variable data.

3.10.2 Executing the CDFbrowse program

The command line syntax for `CDFbrowse` is as follows.

Usage:

VMS:

```
$ CDFBROWSE [<cdf-spec>]
```

UNIX:

```
% cdfbrowse [<cdf-spec>]
```

MS-DOS:

```
> cdfbrowse [<cdf-spec>]
```

Parameter(s):

`<cdf-spec>`	The initial specification of the CDF(s) to browse (do not specify an extension). This may be either a single CDF pathname or a directory/wildcard path. Wildcards are allowed in the CDF name but not in the directory path. This parameter is optional.

Example(s):

VMS:

```
$ CDFBROWSE
$ CDFBROWSE NCDS$DATA:GISS_WETL_CLIMATOLOGY
$ CDFBROWSE FGGE3B
```

UNIX:

```
% cdfbrowse
% cdfbrowse ../cac_sst_blended
% cdfbrowse ~/CDFs/giss_wetl_climatology
```

MS-DOS:

```
> cdfbrowse
> cdfbrowse ..\cac_sst
> cdfbrowse b:\cdfs\giss_wetl_climatology
```

3.10.3 Interaction with CDFbrowse

If not specified on the command line, the first window that appears prompts the user for the CDF and/or directory at which to begin browsing. Entering this results in a window displaying the CDF(s) in that location. The user can scroll upwards or downwards (using the up and down arrow keys), to select the CDF to be browsed. Selecting a CDF and pressing the enter key, results in two windows being displayed. The first window displays the header information and the second window displays a menu with the following selections.

```
Attribute Definitions
Variable Definitions
Variable Values
```

The `Attribute Definition` option, when selected, displays a window containing a list of all the attributes, their scopes (global or variable), and the maximum entry number for each. Selecting an attribute from the list causes a window with the entries for that attribute to be displayed. For each entry, the data type, number of elements, and entry value are displayed. If the attribute is variable in scope, the name of the corresponding variable will also be displayed. For non-character string data types with more than one element, `{...}` will be displayed after the value to indicate that there are more elements. Selecting that entry causes a window with the additional elements to be displayed.

The `Variable Definitions` option, when selected, displays a window containing a list of all the variables, their data types (and number of elements if a character string data type), and their record and dimension variances. Selecting one of the variables causes a window to be displayed containing the variable scope attribute entries associated with that variable. Each entry would include the attribute name, entry data type and number of elements, and the value (attribute entry values are displayed as described for the `Attribute Definitions` option).

The Variable Values option, when selected, displays a window containing a list of all the variables as described for the Variable Definitions section. Selecting a variable results in a window prompting for the record number to display. Entering the record number causes a window to be displayed containing the values for the variable at that record. Each value is displayed along with the corresponding record number and dimension indices. If the values do not change from record to record, an asterisk is displayed as the record number. Likewise, if the values do not change along a dimension, an asterisk is displayed as the index (and the value is only displayed once).

Ctrl-K is used to close a window. Ctrl-E may be used to exit CDFbrowse at any point. When in a window, the up and down arrows are used to move from line to line. If the window contains more items than can be displayed at one time, the items are automatically scrolled when the up arrow key is used at the top of the window and when the down arrow key is used at the bottom of the window. Also, d (the "d" key) may be used to move down more than one line at a time within a window (automatically scrolling the window if necessary). Likewise, u may be used to move up more than one line at a time. The number of lines moved using d or u is dependent on the window size.

The user can, at any instant, invoke the "Help" windows. Two types of Help windows are available. Ctrl-I brings up the "Explain" window which provides information on the current window. Ctrl-L brings up the "Main" help window. The "Main" help window contains a listing of the following topics.

```
Explain the current menu or form.
Explain menus in general.
Show menu key usage.
Explain input forms in general.
Show form key usage.
Show copyright text in CDF V2 header.
Show a map of CDFbrowse's menu structure.
```

Selecting any of the above options brings up a window containing detailed information on the associated topic.

Ctrl-W is used to refresh the terminal display. This is sometimes necessary on certain terminals after particular operations.

3.11 CDFlist

3.11.1 Introduction

The CDFlist program allows you to selectively list all (or a subset of) the variable data values from a CDF. The values can be displayed at the screen, output to a text file, or used to create a new CDF (the new CDF may be a subset of the original CDF).

3.11.2 Executing the CDFlist program

The command line syntax for CDFlist is as follows.

Usage:

VMS:

```
$ CDFLIST [<cdf-spec>]
```

UNIX:

```
% cdflist [<cdf-spec>]
```

MS-DOS:

```
> cdflist [<cdf-spec>]
```

Parameter(s):

`<cdf-spec>`	The initial specification of the CDF(s) to list (do not specify an extension). This may be either a single CDF pathname or a directory/wildcard path. Wildcards are allowed in the CDF name but not in the directory path. This parameter is optional.

Example(s):

VMS:

```
$ CDFLIST
$ CDFLIST GISS_WETL
$ CDFLIST NCDS$DATA:FGGE3B
```

UNIX:

```
% cdflist
% cdflist ../cac_sst_blended
% cdflist ~/CDFs
```

MS-DOS:

```
> cdflist
> cdflist ..\cac_sst
> cdflist a:\cdfs
```

3.11.3 Interaction with CDFlist

The user interface to `CDFlist` is contained on a single screen which is divided into several fixed windows and pop-up windows. The `Action Menu` controls the execution of the program. This menu allows you to move to a different window, start the data output, browse a help file, or exit the program. You must return to this menu each time you want to select a new variable, output option, listing type, or input CDF. This menu may be displayed at any time by entering `Ctrl-F`. Use the up and down arrow keys to highlight a choice in the `Action Menu` and then enter a carriage return to execute that function. Two very important windows are the `Keypad Definition` and `Message` windows. They contain information on keypad functionality and program status/error messages.

When CDFlist is executed (and a CDF and/or directory is not entered on the command line), you must first specify the CDF you wish to list. If you enter a full disk/directory specification (without a CDF name), all of the CDFs in the specified directory will appear in a separate pop-up window. Use the up and down arrow keys to choose a CDF and then enter a carriage return to list that CDF.

The `Output` window will then be displayed with the valid choices. Use the up and down arrow keys to scroll up and down within the menu and then enter a carriage return to select the desired output option. If output to an ASCII file or CDF is selected, you will be prompted for a file/CDF name. You may enter any file/CDF name valid for the operating system begin used or accept the default file/CDF name that appears. Do not enter a file extension (it is automatically appended). The file extension for a generated text file is`.dat`.

The second window that remains constant on the screen is the `Variable Selection` window. This window allows the user to select the variables that are to be listed or put into a CDF along with any desired filters on the variables. You will automatically be placed in this window after you have entered an output option. You can also enter this window by entering `Ctrl-F` to bring up the `Action Menu` and then select the `Select Variables` menu option.

There are three fields on each line in the `Variable Selection` window: variable name, minimum filter value, and maximum filter value. You can move forward to the next field by entering a carriage return or backwards to the previous field with a `Ctrl-H`. You may also use the up and down arrow keys to move vertically through the fields. When the cursor is positioned in a variable name field, entering `Ctrl-P` will display the variables in the CDF in a pop-up window. Use the arrow keys to highlight a variable and then enter a carriage return to select it. You can quit the pop-up window without selecting a variable by entering `q` (quit). You may also use the `u` and `d` keys to page up and down in the pop-up window. The first line in this pop-up menu contains `**SELECT ALL**`. Selecting this item causes all of the variables in the CDF to be selected.

After selecting a variable, the cursor is positioned in the minimum filter field. A filter is simply the range of data that you would like to see from the selected variable. If just specifying a minimum filter value is sufficient, a maximum filter value need not be entered. You do not have to enter a filter for each selected variable. For a line of output (or a set of CDF variable values) to be written for a given record/indices, each filter must be passed. There are two ways to enter minimum and maximum filters. The first is by simply entering the desired values in the appropriate fields. The second method can only be used for variables that consist of a discrete set of values. This method involves entering a `Ctrl-P` (when in the desired filter field) which displays a pop-up window that contains the discrete values for the variable. The desired filter value may then be selected.

When all of the desired variables and filters have been selected, enter `Ctrl-F` to display the `Action Menu`. Selecting `Generate Output` will cause the listing or new CDF to be generated. When `CDFlist` completes the generation of the output, the `Action Menu` is redisplayed. It may then be used to select a new CDF to list, respecify the output option, select new variables/filters, get online help, or exit the program.

NOTE:

If you are running `CDFlist` under UNIX, your terminal output will be limited to 132 chars per line. If too many variables have been selected, the output to a terminal screen will be truncated. Output to a file or CDF, however, will not be affected (as many variables as desired may be selected).

3.12 CDFwalk

3.12.1 Introduction

The `CDFwalk` program allows direct access to related variable data values. The data values of up to 27 variables may be displayed at once (the CDF can have more variables but only 27 may be viewed at any one time). The data value for each variable at a given record/indices location within the CDF conceptual view is displayed. The user can "walk" from one record/indices location to another sequentially (in any "direction") or a particular record/indices location can be "jumped" to directly. Variable values may also be used to directly "jump" to a record/indices location.

3.12.2 Executing the CDFwalk program

The command line syntax for `CDFwalk` is as follows.

Usage:

VMS:

```
$ CDFWALK [<cdf-spec>]
```

UNIX:

```
% cdfwalk [<cdf-spec>]
```

MS-DOS:

```
> cdfwalk [<cdf-spec>]
```

Parameter(s):

`<cdf-spec>`	The initial specification of the CDF(s) at which to begin "walking" (do not specify an extension). This may be either a single CDF pathname or a directory/wildcard path. Wildcards are allowed in the CDF name but not in the directory path. This parameter is optional.

Example(s):

VMS:

```
$ CDFWALK
$ CDFWALK [-.TEMP]NOAA7
```

UNIX:

```
% cdfwalk
% cdfwalk ../cac_sst
% cdfwalk /disk3/CDFs/ssclimate
```

MS-DOS:

```
> cdfwalk
> cdfwalk ..\cac_sst
> cdfwalk c:\cdfs
```

3.12.3 Interaction with CDFwalk

The user interface to `CDFwalk` is contained on a single screen which is divided into several fixed windows and pop-up windows. The `Action Menu` controls the execution of the program. This menu allows you to "walk" through a CDF, select a new CDF through which to "walk", browse online help, or exit the program. This menu may be displayed at any time by entering `Ctrl-F`. Use the up and down arrow keys to highlight a choice in the `Action Menu` and then enter a carriage return to execute that function. Two very important windows are the `Keypad Definition` and `Message` windows. They contain information on keypad functionality and program status/error messages.

When `CDFwalk` is executed (and a CDF and/or directory is not entered on the command line), you must first specify the CDF through which you wish to "walk". If you enter a full disk/directory specification (without a CDF name), all of the CDFs in the specified directory will appear in a separate pop-up window. Use the up and down arrow keys to choose a CDF and then enter a carriage return to "walk" throught that CDF.

After the input CDF is selected, the `Record/Indices` window appears. The size of this window depends on the dimensionality of the input CDF. Vertically the window will contain one row for the record number and one row for each dimension of the CDF. Horizontally, each line of the window contains three fields. The first field contains a variable name, the second field contains a data value for the variable, and the third field contains a record number or dimension index.

The variable selected for the record number line will be one that only varies from record to record (not varying along any of the dimensions). If more than one variable of that type exists in the CDF, a pop-up window will be displayed containing the list of varibles. Select one of the variables keeping in mind that you will have an opportunity to use it for positioning by value to a desired record.

The variable selected for each dimension line will be one that only varys along the corresponding dimension (and does not vary from record to record). Again, if more than one variable of that type exists in the CDF, a pop-up window will be displayed to allow you to select the variable to use for that dimension. Keep in mind that you will be able to use it to position by value to a desired index along that dimension.

After the `Record/Indices` window has been specified (which may occur automatically), the `Variable Values` window will be displayed. It contains up to 27 variable names and the associated data values for the record/indices shown in the `Record/Indices` window.

Initially the cursor will be positioned at the data value field of the record number line in the `Record/Indices` window. You may now begin to "walk" through the CDF by using the right arrow to increment and the left arrow to decrement the record number. Each time a new record number is specified the data for each variable is updated in both the `Record/Indices` and `Variable Values` windows. If the CDF contains many records, you may want to quickly advance to a desired record. You may do this by entering the `i` key (the cursor must be on the data value field). `CDFwalk` enters an input mode and prompts the value to which to advance. If the exact value is not in the CDF for the variable, the record number is changed to correspond to the closest possible value. Another method of advancing to a desired value is by entering `Ctrl-P`. This will cause a pop-up window containing all of the values for the variable to be displayed. You may select the value desired as you would with any other pop-up window and `CDFwalk` will automatically position the record number to the corresponding value.

After positioning to a desired record (and the CDF is multi-dimensional) you may now wish to "walk" through the dimension indices. This is done by entering the up and down arrow keys to position the cursor to the desired dimension line. "Walking" through dimensions is done in the same way as "walking" through the record numbers.

You may also wish to "walk" through the CDF by using the actual record number or dimension indice rather than data values. This is done by entering a carriage return which causes the cursor to advance to the next field (ie. from the data value field to the record number/indices field). Entering `Ctrl-H` causes the cursor to move back to the previous field. "Walking" through the CDF using the record number/indices fields is the same as using the data value fields but with a few exceptions. First, no `Ctrl-P` option is available with these fields. Also, when in input mode, you may input either an absolute record number/index to which to move or you may enter a relative offset for the record number/index. Prefixing the value with a `+/-` will increment/decrement the record number/index by that amount.

If the CDF contains more than 27 variables and a particular variable is not currently displayed in the `Variable Values` window, an option is provied to scroll new variables into that window. This is done by entering a `v` key. A small pop-up window is displayed in which you may use the up and down arrow keys to scroll in or scroll out variables. If the CDF contains a very large number of variables, entering a `u` or `d` key will display a new set of 27 variables (moving up or down through the variables in the CDF). When the desired set of variables is displayed, enter a carriage return to continue "walking" thru the CDF.

When you are done "walking" through the CDF, enter `Ctrl-F` to display the `Action Menu`. You may input a new CDF, resume "walking" through the current CDF, read online help, or exit the program.

Chapter 4

CDF C PROGRAMMING REFERENCE

4.1 Introduction

This chapter provides a reference for using the CDF library from C applications. A separate reference exists for writing CDF applications in FORTRAN (Chapter 5).

4.2 Compiling

Each source file which calls the CDF library or references CDF parameters must include `cdf.h`. This may be done using one of two methods.

4.2.1 Specifying `cdf.h` location in the compile command

The first method involves including the following line at the top of each source file...

```
#include "cdf.h"
```

Since the pathname of the disk/directory containing `cdf.h` was not specified, it must be specified when the source file is compiled.

VMS systems

The command to compile a source file on VMS systems would be as follows...

```
$ CC/INCLUDE_DIRECTORY=<inc-path> <source-file(s)>
```

where `<source-file(s)>` is the name of the source file(s) being compiled and `<inc-path>` is the pathname of the directory containing `cdf.h`. `<inc-path>` may contain a logical name defined by yourself or your system manager. Check with your system manager to determine where on your system `cdf.h` has been installed.

UNIX systems

The command to compile a source file on UNIX systems would be as follows...

```
% cc -c -I<inc-path> <source-file(s)>.c
```

where `<source-file(s)>` is the name of the source file(s) being compiled and `<inc-path>` is the pathname of the directory containing `cdf.h`. `<inc-path>` may contain an environment variable defined by yourself or your system manager. Check with your system manager to determine where on your system `cdf.h` has been installed.

The `-c` option specifies that only an object module(s) is to be produced (the link step is described in Section 4.3).

MS-DOS systems

The command to compile a source file on MS-DOS systems would be as follows (for Microsoft C, Borland C, and Turbo C respectively)...

```
> CL /c /AL /I<inc-path> <source-file(s)>.c
> BCC -c -ml -I<inc-path> <source-file(s)>.c
> TCC -c -ml -I<inc-path> <source-file(s)>.c
```

where `<source-file(s)>` is the name of the source file(s) being compiled and `<inc-path>` is the pathname of the directory containing `cdf.h`. You will need to know where on your system `cdf.h` has been installed.

The `/c`,`-c` option specifies that only an object module(s) is to be produced (the link step is described in Section 4.3).

The `/AL`,`-ml` option specifies that the object module(s) is to be compiled using the large memory model. The huge memory model could also be used if necessary. The CDF library can be built using either the large or huge memory model. All of your application source files must be compiled using the same memory model as that used for the CDF library.

Most C compilers for MS-DOS systems have some type of environment in which you can develop your applications (Integrated Developers Environment in Borland C and Turbo C, Programmer's Workbench in Microsoft C, etc.). The options shown above for the command line compilers are specified in the development environment being used. Consult the documentation for the development environment you are using for the steps needed to compile (and link) your application.

4.2.2 Specifying `cdf.h` location in the source file

The second method involves specifying the pathname of the directory containing `cdf.h` in the actual source file. The following line would be included at the top of each source file...

```
#include "<inc-path>cdf.h"
```

where `<inc-path>` is the pathname of the directory containing `cdf.h`. The source file would then be compiled as shown in Section 4.2.1 but without specifying the location of `cdf.h` on the command line.

NOTE: On VMS systems, `<inc-path>` may contain a logical name defined by yourself or your system manager (on UNIX systems, an environment variable may NOT be used in `<inc-path>`).

Check with your system manager to determine where on your system `cdf.h` has been installed.

4.3 Linking

Your applications must be linked with the CDF library (`LIBCDF.OLB` on VMS systems, `libcdf.a` on UNIX systems, and `libcdf.lib` on MS-DOS systems).

4.3.1 VMS systems

The command to link your application with the CDF library on VMS systems would be as follows...

```
$ LINK <object-file(s)>, <lib-path>LIBCDF/LIBRARY
```

where `<object-file(s)>` is your application's object module(s) and `<lib-path>` is the pathname of the directory containing the CDF library. `<lib-path>` may contain a logical name defined by yourself or your system manager. Check with your system manager to determine where on your system the CDF library has been installed.

The name of the executable created will be the name part of the (first) object file with `.EXE` appended.

4.3.2 UNIX systems

The command to link your application with the CDF library on UNIX systems would be as follows...

```
% cc <object-file(s)>.o <lib-path>libcdf.a
```

where `<object-file(s)>` is your application's object module(s) and `<lib-path>` is the pathname of the directory containing the CDF library. `<lib-path>` may contain an environment variable defined by yourself or your system manager. Check with your system manager to determine where on your system the CDF library has been installed.

The name of the executable created will be `a.out` by default. It may also be explicitly specified using the `-o` option.

Combining the compile and link

On UNIX systems the compile and link may be combined into one step as follows...

```
% cc -I<inc-path> <source-file(s)>.c <lib-path>libcdf.a
```

4.3.3 MS-DOS systems

The command to link your application with the CDF library on MS-DOS systems would be as follows (for Microsoft C, Borland C, and Turbo C respectively)...

```
> LINK <object-file(s)>.obj,,,<lib-path>libcdf.lib
> TLINK /x <object-file(s)>.obj,,,<lib-path>libcdf.lib
> TLINK /x <object-file(s)>.obj,,,<lib-path>libcdf.lib
```

where `<object-file(s)>` is your application's object module(s) and `<lib-path>` is the pathname of the directory containing the CDF library. You will need to know where on your system the CDF library has been installed.

The two omitted parameters (indicated by `,,,`) are the name of the executable to be created and the name of a map file to be created. The name of the executable will be the name part of the (first) object file with `.exe` appended. A different name for the executable may be explicitly specified if desired. A map file is created by default when using the Borland C and Turbo C linkers unless the `/x` option is used (as shown). A map file is only created by the Microsoft C linker if the `/M` option is specified. In either case, the name of the map file will be the name part of the executable file with `.map` appended (unless a name for the map file is explicitly specified).

The same memory model must have been used to compile your application's source files and the CDF library. The CDF library must have been compiled with the large or huge memory model.

Most C compilers for MS-DOS systems have some type of environment in which you can develop your applications (Integrated Developers Environment in Borland C and Turbo C, Programmer's Workbench in Microsoft C, etc.). The options shown above for the command line linkers are specified in the development environment being used. Consult the documentation for the development environment you are using for the steps needed to link (and compile) your application.

4.3.4 Combining the compile and link

On MS-DOS systems the compile and link may be combined into one step as follows (for Microsoft C, Borland C, and Turbo C respectively)...

```
> CL -AL /I<inc-path> <source-file(s)>.c <lib-path>libcdf.lib
> BCC -ml -I<inc-path> <source-file(s)>.c <lib-path>libcdf.lib
> TCC -ml -I<inc-path> <source-file(s)>.c <lib-path>libcdf.lib
```

The name of the executable created will be the name part of the (first) source file with `.exe` appended.

4.4 Item Referencing

For C applications, all items are referenced starting at zero (0). These include variable, attribute, and attribute entry numbers, record numbers, dimensions, and dimension indices.

4.5 CDF Parameters — Predefined Constants and Types

The following sections describe constants and types defined for use by all CDF application programs written in C. These constants and types are defined in `cdf.h`. `cdf.h` should be included in all application source files referencing CDF functions/parameters.

4.5.1 Defined Types

The following typedef's are provided. They should be used when declaring or defining the corresponding items.

`CDFstatus` — All CDF functions except `CDFvarNum` and `CDFattrNum` are of type `CDFstatus`. They return a status code indicating the completion status of the function. The `CDFerror` function can be used to inquire the meaning of any status code. Appendix B lists the possible status codes along with their explanations. Section 4.7 describes how to interpret status codes.

`CDFid` — An identifier (or handle) for a CDF which must be used when referring to a CDF. A new `CDFid` is established whenever a CDF is created or opened, establishing a connection to that CDF on disk. The `CDFid` is used in all subsequent operations on a particular CDF. The `CDFid` must not be altered by an application.

4.5.2 CDFstatus Constants

These constants are of type `CDFstatus`.

`CDF_OK` — A status code indicating the normal completion of a CDF function.

`CDF_WARN` — Threshold constant for testing severity of non-normal CDF status codes.

Section 4.7 describes how to use these constants to interpret status codes.

4.5.3 CDF formats

`SINGLE_FILE`	The CDF consists of only one file.
`MULTI_FILE`	The CDF consists of one header file for control and attribute data and one additional file for each variable in the CDF.

4.5.4 CDF Datatypes

One of the following constants must be used when specifying a CDF data type for an attribute entry or variable.

`CDF_BYTE`	1-byte, signed integer.
`CDF_CHAR`	1-byte, signed character.
`CDF_INT1`	1-byte, signed integer.
`CDF_UCHAR`	1-byte, unsigned character.
`CDF_UINT1`	1-byte, unsigned integer.
`CDF_INT2`	2-byte, signed integer.
`CDF_UINT2`	2-byte, unsigned integer.
`CDF_INT4`	4-byte, signed integer.
`CDF_UINT4`	4-byte, unsigned integer.
`CDF_REAL4`	4-byte, floating point.
`CDF_FLOAT`	4-byte, floating point.
`CDF_REAL8`	8-byte, floating point.
`CDF_DOUBLE`	8-byte, floating point.
`CDF_EPOCH`	8-byte, floating point.

`CDF_CHAR` and `CDF_UCHAR` are considered character string data types. These are significant because only variables of these data types may have more than one element per value (where each element is a character).

4.5.5 Data Encodings

A CDF's data encoding affects how its attribute entry and variable data values are stored (on disk). Attribute entry and variable values in memory are always in the host machine's native encoding.

`HOST_ENCODING`	Indicates host machine data representation (native). This encoding will provide the greatest performance but will not allow the CDF to be transported to other machines that have different encodings.

`NETWORK_ENCODING`	Indicates network transportable data representation (XDR). This encoding allows the CDF to be transported to other machines but with (possibly) reduced performance.
`VAX_ENCODING`	Indicates VAX data representation.
`SUN_ENCODING`	Indicates SUN data representation.
`MIPSEB_ENCODING`	Indicates MIPSEB data representation (Silicon Graphics Iris and Power Series).
`MIPSEL_ENCODING`	Indicates MIPSEL data representation (DECstation).
`IBMRS_ENCODING`	Indicates IBMRS data representation (IBM RS/6000 series).
`HP_ENCODING`	Indicates HP data representation (HP 9000 series).
`IBMPC_ENCODING`	Indicates IBM PC data representation.

When creating a CDF, you must specify `NETWORK_ENCODING`, `HOST_ENCODING`, or the specific encoding for the host machine (eg. `VAX_ENCODING`). Specifying the host machine's encoding explicitly has the same effect as specifying `HOST_ENCODING`.

When inquiring the encoding of a CDF, either `NETWORK_ENCODING` or the specific machine encoding will be returned (`HOST_ENCODING` is never returned).

4.5.6 Variable Majorities

A CDF's variable majority determines the order in which variable values (within the variable arrays) are stored in the CDF file(s).

`ROW_MAJOR`	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest.
`COL_MAJOR`	FORTRAN-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

For C applications, the compiler defined majority for arrays is row major. The first dimension of multi-dimensional arrays varies the slowest in memory.

Knowing the variable majority of a CDF is necessary when performing hyper reads and writes. During a hyper read (using `CDFvarHyperGet`), the CDF library will place the variable data values into the memory buffer in the same majority as that of the CDF. The buffer must then be processed according to that majority. Likewise, during a hyper write (using `CDFvarHyperPut`), the CDF library will expect to find the variable data values in the memory buffer in the same majority as that of the CDF.

4.5.7 Record/Dimension Variances

Record and dimension variances affect how variable data values are physically stored.

`VARY`	True record or dimension variance.
`NOVARY`	False record or dimension variance.

If a variable has a record variance of `VARY`, then each record for that variable is physically stored. If the record variance is `NOVARY`, then only one record is physically stored (all virtual records contain the same values).

If a variable has a dimension variance of `VARY`, then each value along that dimension is physically stored. If the dimension variance is `NOVARY`, then only one value along that dimension is physically stored (all virtual values along that dimension are the same).

4.5.8 Attribute Scopes

Attribute scopes do not affect how the CDF library handles attribute entries. They are simply a way to explicitly declare the intended use of an attribute by user applications.

`GLOBAL_SCOPE`	Indicates that an attribute's scope is global (applies to the CDF as a whole).
`VARIABLE_SCOPE`	Indicates that an attribute's scope is by-variable (each entry corresponds to a variable).
`GLOBAL_SCOPE_ASSUMED`	Indicates that an attribute's scope is assumed global.
`VARIABLE_SCOPE_ASSUMED`	Indicates that an attribute's scope is assumed by-variable.

`GLOBAL_SCOPE` and `VARIABLE_SCOPE` are definite scopes. `GLOBAL_SCOPE_ASSUMED` and `VARIABLE_SCOPE_ASSUMED` are assumed scopes.

Section 2.8.3 explains attribute scopes and how assumed scopes can arise in a CDF.

4.5.9 Operational Limits

These are limits within the CDF library. If you reach one of these limits please contact CDF User Support.

`CDF_MAX_DIMS`	Maximum number of dimensions in a CDF.
`CDF_MAX_VARS`	Maximum number of variables in a CDF.
`CDF_MAX_CDFS`	Maximum number of CDFs that may be open at one time.

The CDF library imposes no limit on the number of attributes or attribute entries that a CDF may have.

4.5.10 Limits of Names and Other Character Strings

`CDF_PATHNAME_LEN`	Maximum length of a CDF pathname (excluding the NUL[1] terminator and the `.cdf` or `.vnn` appended by the CDF library to construct file names). A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating systems being used (including logical names on VMS systems and environment variables on UNIX systems). NOTE: On UNIX systems pathnames are case sensitive.
`CDF_VAR_NAME_LEN`	Maximum length of a variable name (excluding the NUL terminator).
`CDF_ATTR_NAME_LEN`	Maximum length of an attribute name (excluding the NUL terminator).
`CDF_COPYRIGHT_LEN`	Maximum length of the copyright text returned by `CDFdoc` (excluding the NUL terminator).
`CDF_STATUSTEXT_LEN`	Maximum length of the explanation text returned by `CDFerror` (excluding the NUL terminator).

4.6 C Function Reference

The following sections describe the CDF library functions callable from C applications. Most functions return a status code of type `CDFstatus` (see Section 4.7). Each function except `CDFlib` is part of the Standard interface — `CDFlib` is part of the Internal interface. An application can use both interfaces when necessary.

Each section begins with an ANSI-style function prototype.

4.6.1 CDFcreate (Standard interface)

```
CDFstatus CDFcreate(
char  *CDFname,        /* in  -- CDF pathname. */
long  numDims,         /* in  -- Number of dimensions. */
long  dimSizes[],      /* in  -- Dimension sizes. */
long  encoding,        /* in  -- Data encoding. */
long  majority,        /* in  -- Variable majority. */
CDFid *id);            /* out -- CDF identifier. */
```

`CDFcreate` creates a CDF as defined by the arguments. A CDF cannot be created if it already exists (the existing CDF will not be overwritten). If you want to recreate an existing CDF, you must first open it with `CDFopen`, delete it with `CDFdelete`, and then recreate it with `CDFcreate`. If the existing CDF is corrupted, the call to `CDFopen` will fail (an error code will be returned). In this case you must delete the CDF at the command line. Delete the `.cdf` file and if the CDF has the multi-file format, delete all of the `.v0`,`.v1`,... files.

The arguments to `CDFcreate` are defined as follows.

[1] The ASCII null character, 0 octal.

CDFname The pathname of the CDF to create (do not specify an extension). This may be at most CDF_PATHNAME_LEN characters (excluding the NUL). A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

NOTE: On UNIX systems pathnames are case sensitive.

numDims Number of dimensions the CDF is to have. This may be as few as zero (0) and at most CDF_MAX_DIMS.

dimSizes The size of each dimension. dimSizes is a 1-dimensional array containing numDims elements. Each element of dimSizes specifies the corresponding dimension size. Each size must be greater then zero (0). For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

encoding The encoding for variable and attribute entry data. Specify one of the encodings described in Section 4.5.5.

majority The majority for variable data. Specify one of the majorities described in Section 4.5.6.

id The identifier for the created CDF. This identifier must be used in all subsequent operations on the CDF.

When a CDF is created, both read and write access are allowed. The default format for a CDF created with CDFcreate is multi-file (MULTI_FILE). The CDFlib function (Internal interface) must be used to change a CDF's format to single-file (SINGLE_FILE).

CDFclose MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 4.6.5).

Example(s)

The following example will create a 3-dimensional CDF named test1 with network encoding and row majority.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                             /* CDF identifier. */
CDFstatus   status;                         /* Returned status code. */
static long numDims = 3;                    /* Number of dimensions. */
static long dimSizes[3] = {180,360,10};     /* Dimension sizes. */
static long majority = ROW_MAJOR;           /* Variable majority. */
.
.
status = CDFcreate ("test1", numDims, dimSizes, NETWORK_ENCODING,
                    majority, &id);
```

```
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

ROW_MAJOR and NETWORK_ENCODING are defined in cdf.h.

4.6.2 CDFopen (Standard interface)

```
CDFstatus CDFopen(
char  *CDFname,        /* in  -- CDF pathname. */
CDFid *id);            /* out -- CDF identifier. */
```

CDFopen opens an existing CDF. The CDF is initially opened with only read access. This allows multiple applications to read the same CDF simultaneously. When an attempt to modify the CDF is made, it is automatically closed and reopened with read/write access (the function will fail if the application does not have or can not get write access to the CDF).

The arguments to CDFopen are defined as follows.

CDFname — The pathname of the CDF to open (do not specify an extension). This may be at most CDF_PATHNAME_LEN characters (excluding the NUL). A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

NOTE: On UNIX systems pathnames are case sensitive.

id — The identifier for the opened CDF. This identifier must be used in all subsequent operations on the CDF.

CDFclose MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 4.6.5).

Example(s)

The following example will open a CDF named NOAA1.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                           /* CDF identifier. */
CDFstatus   status;                       /* Returned status code. */
static char CDFname[] = { "NOAA1" };      /* Name of CDF. */
```

```
.
.
status = CDFopen (CDFname, &id);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.3 CDFdoc (Standard interface)

```
CDFstatus CDFdoc(
CDFid id,                             /* in  -- CDF identifier. */
long  *version,                       /* out -- Version number. */
long  *release,                       /* out -- Release number. */
char  copyRight[CDF_DOCUMENT_LEN+1]); /* out -- Copyright. */
```

`CDFdoc` is used to inquire general documentation about a CDF. The version/release of the CDF library which created the CDF is provided (eg. CDF V2.1 is version 2, release 1) along with the CDF copyright notice.

The arguments to `CDFdoc` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`version`	The version number of the CDF library which created the CDF.
`release`	The release number of the CDF library which created the CDF.
`copyRight`	The copyright notice of the CDF library which created the CDF. This character string must be large enough to hold `CDF_COPYRIGHT_LEN + 1` characters (including the NUL terminator). This string will contain a newline character after each line of the copyright notice.

The copyright notice is formatted for printing without modification. The version and release are used together (eg. CDF V2.1 is version 2, release 1).

Example(s)

The following example will inquire and display the version/release and copyright notice.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                         /* CDF identifier. */
CDFstatus status;                     /* Returned status code. */
long      version;                    /* CDF version number. */
```

```
long      release;                        /* CDF release number. */
char      copyRight[CDF_COPYRIGHT_LEN+1]; /* Copyright notice -- +1 for
                                             NUL terminator. */
.
.
status = CDFdoc (id, &version, &release, copyRight);

if (status < CDF_OK) /* INFO status codes ignored */
  UserStatusHandler (status);
else {
  printf ("CDF V%d.%d\n", version, release);
  printf("%s\n", copyRight);
}
.
.
```

4.6.4 CDFinquire (Standard interface)

```
CDFstatus CDFinquire(
CDFid id,                        /* in  -- CDF identifier */
long  *numDims,                  /* out -- Number of dimensions. */
long  dimSizes[CDF_MAX_DIMS],    /* out -- Dimension sizes. */
long  *encoding,                 /* out -- Data encoding. */
long  *majority,                 /* out -- Variable majority. */
long  *maxRec,                   /* out -- Maximum record number in the CDF. */
long  *numVars,                  /* out -- Number of variables in the CDF. */
long  *numAttrs);                /* out -- Number of attributes in the CDF. */
```

`CDFinquire` inquires the basic characteristics of a CDF. An application needs to know the number of dimensions and their sizes before it can access variable data. Knowing the variable majority can be used to optimize performance and is necessary to properly access the buffer of data values used with the hyper functions — `CDFvarHyperPut` and `CDFvarHyperGet`.

The arguments to `CDFinquire` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.

`numDims` The number of dimensions in the CDF.

`dimSizes` The dimension sizes. `dimSizes` is a 1-dimensional array containing one element per CDF dimension. Each element of `dimSizes` receives the corresponding dimension size. For 0-dimensional CDFs, this argument is ignored (but a place holder is necessary).

`encoding` The encoding of the variable and attribute entry data. The encodings are defined in Section 4.5.5.

`majority` The majority of the variable data. The majorities are defined in Section 4.5.6.

`maxRec`	The maximum record number written to the CDF. Note that the maximum record number written is kept separately for each variable in the CDF. `maxRec` is the largest of these (the conceptual maximum record written). Some variables may have fewer records actually written. `CDFlib` (Internal interface) may be used to inquire the maximum record written for an individual variable (see Section 4.6.24).
`numVars`	The number of variables in the CDF.
`numAttrs`	The number of attributes in the CDF.

Example(s)

The following example will inquire the basic information about a CDF.

```
.
.
#include "cdf.h"
.
.
CDFid id;                     /* CDF identifier. */
CDFstatus status;             /* Returned status code. */
long numDims;                 /* Number of dimensions. */
long dimSizes[CDF_MAX_DIMS];  /* Dimension sizes (allocate to allow the
                                 maximum number of dimensions). */
long encoding;                /* Data encoding. */
long majority;                /* Variable majority. */
long maxRec;                  /* Maximum record number in CDF. */
long numVars;                 /* Number of variables in CDF. */
long numAttrs;                /* Number of attributes in CDF. */
.
.
status = CDFinquire (id, &numDims, dimSizes, &encoding, &majority,
                     &maxRec, &numVars, &numAttrs);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.5 CDFclose (Standard interface)

```
CDFstatus CDFclose(
CDFid id);              /* in -- CDF identifier. */
```

`CDFclose` closes the specified CDF. The CDF's data buffers are flushed, all of the CDF's open files are closed, and the CDF identifier is made available for reuse.

You MUST close a CDF with `CDFclose` to guarantee that all modifications you have made will actually be written to the CDF's file(s). If your program exits, normally or otherwise, without a successful call to

CDFclose, the CDF's data buffers are left unflushed. If you have only read from a CDF, and not written to it, failure to close it is of no consequence. Of course, it is always good practice to close your CDFs before exiting your application.

The arguments to CDFclose are defined as follows.

id — The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.

Example(s)

The following example will close an open CDF.

```
 .
 .
 #include "cdf.h"
 .
 .
CDFid id;                       /* CDF identifier. */
CDFstatus status;               /* Returned status code. */
.
.
status = CDFclose (id);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.6 CDFdelete (Standard interface)

```
CDFstatus CDFdelete(
CDFid id);            /* in -- CDF identifier. */
```

CDFdelete deletes the specified CDF. The CDF files deleted include the .cdf file and if the CDF consists of multiple files, the .v0,.v1,... files.

You must open a CDF before you are allowed to deleted it. If you have no privilege to delete the CDF files, they will not be deleted. If the CDF is corrupted and cannot be opened, the CDF file(s) must be deleted at the command line.

The arguments to CDFdelete are defined as follows.

id — The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.

Example(s)

The following example will open and then delete an existing CDF.

```
.
.
#include "cdf.h"
.
.
CDFid id;               /* CDF identifier. */
CDFstatus status;       /* Returned status code. */
.
.
status = CDFopen ("test2", &id);
if (status < CDF_OK)               /* INFO status codes ignored. */
  UserStatusHandler (status);
else {
  status = CDFdelete (id);
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

4.6.7 CDFerror (Standard interface)

```
CDFstatus CDFerror(
CDFstatus status,                          /* in  -- Status code. */
char      message[CDF_STATUSTEXT_LEN+1]);  /* out -- Explanation text for
                                                     the status code. */
```

`CDFerror` is used to inquire the explanation of a given status code (not just error codes). Section 4.7 explains how to interpret status codes and Appendix B lists all of the possible status codes.

The arguments to `CDFerror` are defined as follows.

`status`	The status code to check.
`message`	The explanation of the status code. This character string must be large enough to hold `CDF_STATUSTEXT_LEN + 1` characters (including the NUL terminator).

Example(s)

The following example displays the explanation text if an error code is returned from a call to `CDFopen`.

```
.
```

```
.
#include "cdf.h"
.
.
CDFid     id;                     /* CDF identifier. */
CDFstatus status;                 /* Returned status code. */
char text[CDF_STATUSTEXT_LEN+1];  /* Explanation text.  +1 added for
                                     NUL terminator. */
.
.
status = CDFopen ("giss_wetl", &id);
if (status < CDF_WARN) {              /* INFO and WARNING codes ignored. */
  CDFerror (status, text);
  printf ("ERROR> %s\n", text);
}
.
.
```

4.6.8 CDFattrCreate (Standard interface)

```
CDFstatus CDFattrCreate(
CDFid id,            /* in  -- CDF identifier. */
char  *attrName,     /* in  -- Attribute name. */
long  attrScope,     /* in  -- Scope of attribute. */
long  *attrNum);     /* out -- Attribute number. */
```

`CDFattrCreate` creates an attribute in the specified CDF. An attribute with the same name must not already exist in the CDF.

The arguments to `CDFattrCreate` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`attrName`	The name of the attribute to create. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the NUL terminator). Attribute names are case-sensitive.
`attrScope`	The scope of the new attribute. Specify one of the scopes described in Section 4.5.8.
`attrNum`	The number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may be determined with the `CDFattrNum` function.

Example(s)

The following example creates two attributes. The `TITLE` attribute is created with global scope — it applies to the entire CDF (most likely the title of the data set stored in the CDF). The `Units` attribute is created with variable scope — each entry describes some property of the corresponding variable (in this case the units for the data).

```
.
.
#include "cdf.h"
.
.
CDFid     id;                               /* CDF identifier. */
CDFstatus status;                           /* Returned status code. */
static char UNITSattrName[] = {"Units"};    /* Name of "Units" attribute. */
long        UNITSattrNum;                   /* "Units" attribute number. */
long        TITLEattrNum;                   /* "TITLE" attribute number. */
static long TITLEattrScope = GLOBAL_SCOPE;  /* "TITLE" attribute scope. */
.
.
status = CDFattrCreate (id, "TITLE", TITLEattrScope, &TITLEattrNum);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFattrCreate (id, UNITSattrName, VARIABLE_SCOPE, &UNITSattrnum);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.9 CDFattrNum (Standard interface)

```
long CDFattrNum(
CDFid id,            /* in -- CDF id */
char  *attrName);    /* in -- attribute name */
```

`CDFattrNum` is used to determine the attribute number associated with a given attribute name. If the attribute is found, `CDFattrNum` returns its number — which will be equal to or greater than zero (0). If an error occurs (eg. the attribute name does not exist in the CDF), an error code (of type `CDFstatus`) is returned. Error codes are less than zero (0).

The arguments to `CDFattrNum` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.

`attrName` The name of the attribute for which to search. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the NUL terminator). Attribute names are case-sensitive.

`CDFattrNum` may be used as an embedded function call when an attribute number is needed.

Example(s)

In the following example the attribute named `pressure` will be renamed `PRESSURE` with `CDFattrNum` being used as an embedded function call. Note that if the attribute `pressure` did not exist in the CDF, the call to `CDFattrNum` would have returned an error code. Passing that error code to `CDFattrRename` as an attribute number would have resulted in `CDFattrRename` also returning an error code. `CDFattrRename` is described in Section 4.6.10.

```
.
.
#include "cdf.h"
.
.
CDFid     id;        /* CDF identifier. */
CDFstatus status;    /* Returned status code. */
.
.
status = CDFattrRename (id, CDFattrNum(id,"pressure"), "PRESSURE");
if (status != CDF_OK) UserStatusHandler (status);
```

4.6.10 CDFattrRename (Standard interface)

```
CDFstatus CDFattrRename(
CDFid id,             /* in -- CDF identifier. */
long  attrNum,        /* in -- Attribute number. */
char  *attrName);     /* in -- New attribute name. */
```

`CDFattrRename` is used to rename an existing attribute. An attribute with the new name must not already exist in the CDF.

The arguments to `CDFattrRename` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`attrNum`	The number of the attribute to rename. This number may be determined with a call to `CDFattrNum` (see Section 4.6.9).
`attrName`	The new attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the NUL terminator). Attribute names are case-sensitive.

Example(s)

In the following example the attribute named `LAT` is renamed `LATITUDE`.

```
.
```

```
.
#include "cdf.h"
.
.
CDFid     id;       /* CDF identifier. */
CDFstatus status;   /* Returned status code. */
.
.
status = CDFattrRename (id, CDFattrNum(id,"LAT"), "LATITUDE");
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.11 CDFattrInquire (Standard interface)

```
CDFstatus CDFattrInquire(
CDFid id,             /* in  -- CDF identifier. */
long  attrNum,        /* in  -- Attribute number. */
char  *attrName,      /* out -- Attribute name. */
long  *attrScope,     /* out -- Attribute scope. */
long  *maxEntry);     /* out -- Maximum entry number for attribute. */
```

`CDFattrInquire` is used to inquire about the specified attribute. To inquire about a specific attribute entry, use `CDFattrEntryInquire` (Section 4.6.12).

The arguments to `CDFattrInquire` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`attrNum`	The number of the attribute to inquire. This number may be determined with a call to `CDFattrNum` (see Section 4.6.9).
`attrName`	The attribute's name. This character string must be large enough to hold `CDF_ATTR_NAME_LEN + 1` characters (including the NUL terminator).
`attrScope`	The scope of the attribute. Attribute scopes are defined in Section 4.5.8.
`maxEntry`	The maximum entry number for this attribute. This may not correspond with the number of entries (if some entry numbers were not used). The number of entries actually used may be inquired with the `CDFlib` function (Internal interface, see Section 4.6.24).

Example(s)

The following example displays the name of each attribute in a CDF. The number of attributes in the CDF is first determined using the function `CDFinquire`. Note that attribute numbers start at zero (0) and are consecutive.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                        /* CDF identifier. */
CDFstatus status;                    /* Returned status code. */
long numDims;                        /* Number of dimensions. */
long dimSizes[CDF_MAX_DIMS];         /* Dimension sizes (allocate to allow
                                        the maximum number of dimensions). */
long encoding;                       /* Data encoding. */
long majority;                       /* Variable majority. */
long maxRec;                         /* Maximum record number in CDF. */
long numVars;                        /* Number of variables in CDF. */
long numAttrs;                       /* Number of attributes in CDF. */
long attrN;                          /* Attribute number. */
char attrName[CDF_ATTR_NAME_LEN+1]; /* Attribute name -- +1 for NUL
                                        terminator. */
long attrScope;                      /* Attribute scope. */
long maxEntry;                       /* Maximum entry number. */
.
.
status = CDFinquire (id, &numDims, dimSizes, &encoding, &majority,
                     &maxRec, &numVars, &numAttrs);
if (status != CDF_OK) UserStatusHandler (status);

for (attrN = 0; attrN < numAttrs; attrN++) {
   status = CDFattrInquire (id, attrN, attrName, &attrScope, &maxEntry);
   if (status < CDF_OK)              /* INFO status codes ignored. */
     UserStatusHandler (status);
   else
     printf ("%s\n", attrName);
}
.
.
```

4.6.12 CDFattrEntryInquire (Standard interface)

```
CDFstatus CDFattrEntryInquire(
CDFid id,             /* in  -- CDF identifier. */
long  attrNum,        /* in  -- Attribute number. */
long  entryNum,       /* in  -- Entry number. */
long  *dataType,      /* out -- Entry data type. */
long  *numElements);  /* out -- Number of elements (of the data type). */
```

`CDFattrEntryInquire` is used to inquire about a specific attribute entry. To inquire about the attribute in general, use `CDFattrInquire` (Section 4.6.11). `CDFattrEntryInquire` would normally be called before calling `CDFattrGet` in order to determine the data type and number of elements (of that data type) for an entry. This would be necessary to correctly allocate enough memory to receive the value read by `CDFattrGet`.

The arguments to CDFattrEntryInquire are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
attrNum	The attribute number in which to inquire an entry. This number may be determined with a call to CDFattrNum (see Section 4.6.9).
entryNum	The entry number to inquire. If the attribute is global in scope, this is simply the entry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated variable (the variable being described in some way by this entry).
dataType	The data type of the specified entry. The data types are defined in Section 4.5.4.
numElements	The number of elements of the data type. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string (an array of characters). For all other data types, this is the number of elements in an array of that data type.

Example(s)

The following example inquires each entry for an attribute. Note that entry numbers need not be consecutive — not every entry number between zero (0) and the maximum entry number must exist. For this reason, NO_SUCH_ENTRY is an expected error code. Note also that if the attribute has variable scope, the entry numbers are actually variable numbers.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                            /* CDF identifier. */
CDFstatus status;                        /* Returned status code. */
long      attrN;                         /* Attribute number. */
long      entryN;                        /* Entry number. */
char      attrName[CDF_ATTR_NAME_LEN+1]; /* Attribute name, +1 for NUL
                                            terminator. */
long      attrScope;                     /* Attribute scope. */
long      maxEntry;                      /* Maximum entry number. */
long      dataType;                      /* Entry data type. */
long      numElems;                      /* Number of elements (of the
                                            data type). */
.
.
attrN = CDFattrNum (id, "TMP");
if (attrN < 0) UserStatusHandler (attrN);  /* If less than zero (0), then
                                              it must be a warning/error
                                              code. */
```

```
  status = CDFattrInquire (id, attrN, attrName, &attrScope, &maxEntry);
  if (status != CDF_OK) UserStatusHandler (status);

  for (entryN = 0; entryN <= maxEntry; entryN++) {
     status = CDFattrEntryInquire (id, attrN, entryN, &dataType, &numElems);
     if (status < CDF_OK) {
       if (status != NO_SUCH_ENTRY) UserStatusHandler (status);
     }
     else {
       /* process entries */
       .
       .
     }
  }
```

4.6.13 CDFattrPut (Standard interface)

```
CDFstatus CDFattrPut(
CDFid id,             /* in -- CDF identifier. */
long  attrNum,        /* in -- Attribute number. */
long  entryNum,       /* in -- Entry number. */
long  dataType,       /* in -- Data type of this attribute entry. */
long  numElements,    /* in -- Number of elements (of the data type). */
void  *value);        /* in -- Value. */
```

`CDFattrPut` is used to write an attribute entry to a CDF. The entry may or may not already exist. If it does exist, it is overwritten. The data type and number of elements (of that data type) may be changed when overwriting an existing entry.

The arguments to `CDFattrPut` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`attrNum`	The attribute number. This number may be determined with a call to `CDFattrNum` (see Section 4.6.9).
`entryNum`	The entry number. If the attribute is global in scope, this is simply the entry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated variable (the variable being described in some way by this entry).
`dataType`	The data type of the specified entry. Specify one of the data types defined in Section 4.5.4.
`numElements`	The number of elements of the data type. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (an array of characters). For all other data types, this is the number of elements in an array of that data type.

`value` The value(s) to write. The entry value is written to the CDF from memory address `value`.

`numElements` elements of the data type `dataType` will be written to the CDF starting from memory address `value`.

Example(s)

The following example writes two attribute entries. The first is to entry number zero (0) of the global scope attribute `TITLE`. The second is to the variable scope attribute `VALIDs` for the entry which corresponds to variable `TMP`.

```
.
.
#include "cdf.h"
.
.
#define TITLE_LEN   10               /* Length of CDF title. */
.
.
CDFid        id;                     /* CDF identifier. */
CDFstatus    status;                 /* Returned status code. */
long         entryNum;               /* Entry number. */
long         numElements;            /* Number of elements (of data
                                        type). */
static char  title[TITLE_LEN+1]
                   = {"CDF title."}; /* Value of TITLE attribute,
                                        entry 0. */
static short TMPvalids = {15,30};    /* Value(s) of VALIDs attribute,
                                        entry for TMP variable. */
.
.
entryNum = 0;
status = CDFattrPut (id, CDFattrNum(id,"TITLE"), entryNum, CDF_CHAR,
                     TITLE_LEN, title);
if (status != CDF_OK) UserStatusHandler (status);
.
.
numElements = 2;
status = CDFattrPut (id, CDFattrNum(id,"VALIDs"), CDFvarNum(id,"TMP"),
                     CDF_INT2, numElements, TMPvalids);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.14 CDFattrGet (Standard interface)

```
CDFstatus CDFattrGet(
CDFid id,             /* in  -- CDF identifier. */
long  attrNum,        /* in  -- Attribute number. */
long  entryNum,       /* in  -- Entry number. */
void  *value);        /* out -- Value. */
```

`CDFattrGet` is used to read an attribute entry from a CDF. In most cases it will be necessary to call `CDFattrEntryInquire` before calling `CDFattrGet` in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to `CDFattrGet` are defined as follows.

`id` — The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.

`attrNum` — The attribute number. This number may be determined with a call to `CDFattrNum` (see Section 4.6.9).

`entryNum` — The entry number. If the attribute is global in scope, this is simply the entry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated variable (the variable being described in some way by this entry).

`value` — The value read. This buffer must be large enough to hold the entry value. `CDFattrEntryInquire` would be used to determine the entry data type and number of elements (of that data type). The entry value is read from the CDF and placed into memory at address `value`.

Example(s)

The following example displays the value of the `UNITS` attribute for the entry corresponding to the `PRES_LVL` variable (but only if the data type is `CDF_CHAR`). Note that the CDF library does not automatically NUL terminate character data (when the data type is `CDF_CHAR` or `CDF_UCHAR`) for attribute entries or variable values.

```
.
.
#include "cdf.h"
.
.
CDFid     id;              /* CDF identifier. */
CDFstatus status;          /* Returned status code. */
long      attrN;           /* Attribute number. */
long      entryN;          /* Entry number. */
long      dataType;        /* Data type. */
long      numElems;        /* Number of elements (of data type). */
void      *buffer;         /* Buffer to receive entry value. */
```

```
.
.
attrN = CDFattrNum (id, "UNITS");
if (attrN < 0) UserStatusHandler (attrN);  /* If less than zero (0), then
                                              it must be a warning/error
                                              code. */

entryN = CDFvarNum (id, "PRES_LVL");        /* Variable number is entry
                                               number. */
if (entryN < 0) UserStatusHandler (status);  /* If less than zero (0), then
                                               it must be a warning/error
                                               code. */

status =  CDFattrEntryInquire (id, attrN, entryN, &dataType, &numElems);
if (status != CDF_OK) UserStatusHandler (status);

if (dataType == CDF_CHAR) {
  buffer = (char *) malloc (numElems);
  if (buffer == NULL)...

  status = CDFattrGet (id, attrN, entryN, buffer);
  if (status != CDF_OK) UserStatusHandler (status);

  buffer[numElems] = '\0';         /* NUL terminate. */

  printf ("Units of PRES_LVL variable: %s\n", buffer);

  free (buffer);
}
.
.
```

4.6.15 CDFattrCorrectScope (Standard interface)

```
CDFstatus CDFattrCorrectScope(
CDFid id,             /* in -- CDF identifier. */
long  attrNum,        /* in -- Attribute number. */
long  attrScope)      /* in -- New scope of attribute. */
```

`CDFattrCorrectScope` is used to correct the scope of an attribute. Section 2.8.3 describes attribute scopes and the need for this function.

The arguments to `CDFattrCorrectScope` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`attrNum`	The attribute number. This number may be determined with a call to `CDFattrNum` (see Section 4.6.9).

`attrScope` The new scope for the attribute. Specify one of the definite scopes defined in Section 4.5.8.

IMPORTANT NOTE

This function is not a permanent one — it may be deleted from a future release of the CDF library. Version 1 CDFs should be converted to Version 2 CDFs as soon as possible. Any assumed attribute scopes should then be corrected by using this function or the `CDFscope` toolkit program.

Example(s)

The following example will correct the scope of every attribute in a CDF. Assumed scopes will be corrected to the corresponding definite scope. Note that it may not always be the case that an assumed scope should be corrected to the corresponding definite scope (see Section 2.8.3). The toolkit program `CDFscope` could also be used to correct attribute scopes.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                      /* CDF identifier. */
CDFstatus status;                  /* Returned status code. */
long numDims;                      /* Number of dimensions. */
long dimSizes[CDF_MAX_DIMS];       /* Dimension sizes (allocate to allow
                                      the maximum number of dimensions). */
long encoding;                     /* Data encoding. */
long majority;                     /* Variable majority. */
long maxRec;                       /* Maximum record number in CDF. */
long numVars;                      /* Number of variables in CDF. */
long numAttrs;                     /* Number of attributes in CDF. */
long attrN;                        /* Attribute number. */
char attrName[CDF_ATTR_NAME_LEN+1]; /* Attribute name -- +1 for NUL
                                      terminator. */
long attrScope;                    /* Attribute scope. */
long maxEntry;                     /* Maximum entry number. */
.
.
status = CDFinquire (id, &numDims, dimSizes, &encoding, &majority,
                     &maxRec, &numVars, &numAttrs);
if (status != CDF_OK) UserStatusHandler (status);

for (attrN = 0; attrN < numAttrs; attrN++) {
   status = CDFattrInquire (id, attrN, attrName, &attrScope, &maxEntry);
   if (status != CDF_OK) UserStatusHandler (status);

   switch (attrScope) {
     case GLOBAL_SCOPE_ASSUMED:
```

```
        status = CDFattrCorrectScope (id, attrN, GLOBAL_SCOPE);
        if (status != CDF_OK) UserStatusHandler (status);
        break;
      case VARIABLE_SCOPE_ASSUMED:
        status = CDFattrCorrectScope (id, attrN, VARIABLE_SCOPE);
        if (status != CDF_OK) UserStatusHandler (status);
        break;
      default:
        /* Do nothing --- definite scope. */
    }
  }
  .
  .
```

4.6.16 CDFvarCreate (Standard interface)

```
CDFstatus CDFvarCreate(
CDFid id,               /* in --  CDF identifier. */
char  *varName,         /* in --  Variable name. */
long  dataType,         /* in --  Data type. */
long  numElements,      /* in --  Number of elements (of the data type). */
long  recVariance,      /* in --  Record variance. */
long  dimVariances[],   /* in --  Dimension variances. */
long  *varNum);         /* out -- Variable number. */
```

`CDFvarCreate` is used to create a new variable in a CDF. A variable with the same name must not already exist in the CDF.

The arguments to `CDFvarCreate` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.

`varName` The name of the variable to create. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the NUL terminator). Variable names are case-sensitive.

`dataType` The data type of the new variable. Specify one of the data types defined in Section 4.5.4.

`numElements` The number of elements of the data type at each variable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (each variable value consists of the entire string). For all other data types, this must always be one (1) — multiple elements at each variable value are not allowed for non-character string data types.

`recVariance` The variable's record variance. Specify one of the variances defined in Section 4.5.7.

`dimVariances` The variable's dimension variances. `dimVariances` is a 1-dimensional array containing one element per CDF dimension. Each element of `dimVariances` specifies the corresponding dimension variance. For each dimension, specify one of the variances defined in Section 4.5.7. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

varNum The number assigned to the new variable. This number must be used in subsequent CDF function calls when referring to this variable. An existing variables's number may be determined with the **CDFvarNum** function.

Example(s)

The following example will create several variables in a 2-dimensional CDF. In this case, **EPOCH**, **LAT**, and **LON** are independent variables and **TMP** is a dependent variable.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                               /* CDF identifier. */
CDFstatus status;                           /* Returned status code. */
static long EPOCHrecVary = {VARY};          /* EPOCH record variance. */
static long LATrecVary = {NOVARY};          /* LAT record variance. */
static long LONrecVary = {NOVARY};          /* LON record variance. */
static long TMPrecVary = {VARY};            /* TMP record variance. */
static long EPOCHdimVarys = {NOVARY,NOVARY}; /* EPOCH dimension
                                               variances. */
static long LATdimVarys = {NOVARY,VARY};    /* LAT dimension variances. */
static long LONdimVarys = {VARY,NOVARY};    /* LON dimension variances. */
static long TMPdimVarys = {VARY,VARY};      /* TMP dimension variances. */
long        EPOCHvarNum;                    /* EPOCH variable number. */
long        LATvarNum;                      /* LAT variable number. */
long        LONvarNum;                      /* LON variable number. */
long        TMPvarNum;                      /* TMP variable number. */
.
.
status = CDFvarCreate (id, "EPOCH", CDF_EPOCH, 1,
                       EPOCHrecVary, EPOCHdimVarys, &EPOCHvarNum);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFvarCreate (id, "LATITUDE", CDF_INT2, 1,
                       LATrecVary, LATdimVarys, &LATvarNum);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFvarCreate (id, "LONGITUDE", CDF_INT2, 1,
                       LONrecVary, LONdimVarys, &LONvarNum);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFvarCreate (id, "TEMPERATURE", CDF_REAL4, 1,
                       TMPrecVary, TMPdimVarys, &TMPvarNum);
```

```
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.17 CDFvarNum (Standard interface)

```
long CDFvarNum(
CDFid id,              /* in -- CDF identifier. */
char  *varName);       /* in -- Variable name. */
```

CDFvarNum is used to determine the variable number associated with a given variable name. If the variable is found, **CDFvarNum** returns its number — which will be equal to or greater than zero (0). If an error occurs (eg. the variable name does not exist in the CDF), an error code (of type **CDFstatus**) is returned. Error codes are less than zero (0).

The arguments to **CDFvarNum** are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
varName	The name of the variable for which to search. This may be at most CDF_VAR_NAME_LEN characters (excluding the NUL terminator). Variable names are case-sensitive.

CDFvarNum may be used as an embedded function call when a variable number is needed.

Example(s)

In the following example **CDFvarNum** is used as an embedded function call when inquiring about a variable.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                           /* CDF identifier. */
CDFstatus status;                       /* Returned status code. */
char      varName[CDF_VAR_NAME_LEN+1];  /* Variable name. */
long      dataType;                     /* Data type of the variable. */
long      numElements;                  /* Number of elements (of the
                                           data type). */
long      recVariances;                 /* Record variance. */
long      dimVariances[CDF_MAX_DIMS];   /* Dimension variances. */
.
.
status = CDFvarInquire (id, CDFvarNum(id,"LATITUDE"), varName, &dataType,
                        &numElements, &recVariance, dimVariances);
```

```
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

In this example the variable named LATITUDE was inquired. Note that if the variable LATITUDE did not exist in the CDF, the call to CDFvarNum would have returned an error code. Passing that error code to CDFvarInquire as a variable number would have resulted in CDFvarInquire also returning an error code. Also note that the name written into varName is already known (LATITUDE). In some cases the variable names will be unknown — CDFvarInquire would be used to determine them. CDFvarInquire is described in Section 4.6.19.

4.6.18 CDFvarRename (Standard interface)

```
CDFstatus CDFvarRename(
CDFid id,             /* in -- CDF identifier. */
long  varNum,         /* in -- Variable number. */
char  *varName);      /* in -- New variable name. */
```

CDFvarRename is used to rename an existing variable. A variable with the new name must not already exist in the CDF.

The arguments to CDFvarRename are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
varNum	The number of the variable to rename. This number may be determined with a call to CDFvarNum (see Section 4.6.17).
varName	The new variable name. This may be at most CDF_VAR_NAME_LEN characters (excluding the NUL terminator). Variable names are case-sensitive.

Example(s)

In the following example the variable named TEMPERATURE is renamed TMP (if it exists). Note that if CDFvarNum returns a value less than zero (0), that value is not a variable number but rather a warning/error code.

```
.
.
#include "cdf.h"
.
.
CDFid     id;       /* CDF identifier. */
CDFstatus status;   /* Returned status code. */
long      varNum;   /* Variable number. */
```

```
.
.
varNum = CDFvarNum (id, "TEMPERATURE");
if (varNum < 0) {
  if (varNum != NO_SUCH_VAR) UserStatusHandler (varNum);
}
else {
  status = CDFvarRename (id, varNum, "TMP");
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

4.6.19 CDFvarInquire (Standard interface)

```
CDFstatus CDFvarInquire(
CDFid id,                          /* in  -- CDF identifier. */
long  varNum,                      /* in  -- Variable number. */
char  varName,                     /* out -- Variable name. */
long  *dataType,                   /* out -- Data type. */
long  *numElements,                /* out -- Number of elements (of the
                                             data type). */
long  *recVariance,                /* out -- Record variance. */
long  dimVariances[CDF_MAX_DIMS]); /* out -- Dimension variances. */
```

`CDFvarInquire` is used to inquire about the specified variable. This function would normally be used before reading variable values (with `CDFvarGet` or `CDFvarHyperGet`) to determine the data type and number of elements (of that data type).

The arguments to `CDFvarInquire` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.

`varNum` The number of the variable to inquire. This number may be determined with a call to `CDFvarNum` (see Section 4.6.17).

`varName` The variable's name. This character string must be large enough to hold `CDF_VAR_NAME_LEN + 1` characters (including the NUL terminator).

`dataType` The data type of the variable. The data types are defined in Section 4.5.4.

`numElements` The number of elements of the data type at each variable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (each variable value consists of the entire string). For all other data types, this will always be one (1) — multiple elements at each variable value are not allowed for non-character string data types.

`recVariance` The record variance. The record variances are defined in Section 4.5.7.

`dimVariances` The dimension variances. `dimVariances` is a 1-dimensional array containing one element per CDF dimension. Each element of `dimVariances` receives the corresponding dimension variance. The dimension variances are defined in Section 4.5.7. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

Example(s)

The following example inquires about a variable named `HEAT_FLUX` in a CDF. Note that the variable name returned by `CDFvarInquire` will be the same as that passed in to `CDFvarNum`.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                          /* CDF identifier. */
CDFstatus status;                      /* Returned status code. */
char      varName[CDF_VAR_NAME_LEN+1]; /* Variable name, +1 for NUL
                                          terminator. */
long      dataType;                    /* Data type. */
long      numElems;                    /* Number of elements (of data
                                          type). */
long      recVary;                     /* Record variance. */
long      dimVarys[CDF_MAX_DIMS];      /* Dimension variances (allocate
                                          to allow the maximum number of
                                          dimensions). */
.
.
status = CDFvarInquire (id, CDFvarNum(id,"HEAT_FLUX"), varName, &dataType,
                        &numElems, &recVary, dimVarys);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.20 CDFvarPut (Standard interface)

```
CDFstatus CDFvarPut(
CDFid id,             /* in -- CDF identifier. */
long  varNum,         /* in -- Variable number. */
long  recNum,         /* in -- Record number. */
long  indices[],      /* in -- Dimension indices. */
void  *value);        /* in -- Value. */
```

`CDFvarPut` is used to write a single value to a variable. `CDFvarHyperPut` may be used to write more than one variable value with a single call (see Section 4.6.22).

The arguments to `CDFvarPut` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`varNum`	The number of the variable to which to write. This number may be determined with a call to `CDFvarNum` (see Section 4.6.17).
`recNum`	The record number at which to write.
`indices`	The array indices within the specified record at which to write. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
`value`	The value to write. The value is written to the CDF from memory address `value`.

Example(s)

The following example writes values to the `LATITUDE` variable of a 2-dimensional CDF having dimension sizes `360,181`. For `LATITUDE`, the record variance is `NOVARY`, the dimension variances are `NOVARY,VARY`, and the data type is `CDF_INT2`.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                         /* CDF identifier. */
CDFstatus   status;                     /* Returned status code. */
short       lat;                        /* Latitude value. */
long        varN;                       /* Variable number. */
static long recNum = 0;                 /* Record number. */
static long indices[2] = {0,0};         /* Dimension indices. */
.
.
varN = CDFvarNum (id, "LATITUDE");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), not a
                                            variable number but rather a
                                            warning/error code. */

for (lat = -90; lat <= 90; lat += 1) {
   indices[1] = 90 + lat;
   status = CDFvarPut (id, varN, recNum, indices, &lat);
   if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

Since the record variance is `NOVARY`, the record number (`recNum`) is set to zero (0). Also note that because the dimension variances are `NOVARY,VARY`, only the second dimension is varied as values are written (the values written are the same at each index of the first dimension in the conceptual view of the variable).

4.6.21 CDFvarGet (Standard interface)

```
CDFstatus CDFvarGet(
CDFid id,            /* in  -- CDF identifier. */
long  varNum,        /* in  -- Variable number. */
long  recNum,        /* in  -- Record number. */
long  indices[],     /* in  -- Dimension indices. */
void  *value);       /* out -- Value. */
```

`CDFvarGet` is used to read a single value from a variable. `CDFvarHyperGet` may be used to read more than one variable value with a single call (see Section 4.6.23).

The arguments to `CDFvarGet` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`varNum`	The number of the variable from which to read. This number may be determined with a call to `CDFvarNum` (see Section 4.6.17).
`recNum`	The record number at which to read.
`indices`	The array indices within the specified record at which to read. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
`value`	The value read. This buffer must be large enough to hold the variable value. `CDFvarInquire` would be used to determine the variable's data type and number of elements (of that data type) at each variable value. The value is read from the CDF and placed at memory address `value`.

Example(s)

The following example will read and hold an entire record of data from a variable. The CDF is 3-dimensional with sizes `180,91,10`. For the variable, the record variance is `VARY`, the dimension variances are `VARY,VARY,VARY`, and the data type is `CDF_REAL4`.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                /* CDF identifier. */
CDFstatus status;            /* Returned status code. */
float     tmp[180][91][10];  /* Temperature values. */
```

```
long      indices[3];        /* Dimension indices. */
long      varN;              /* Variable number. */
long      recNum;            /* Record number. */
long      d0, d1, d2;        /* Dimension index values. */
.
.
varN = CDFvarNum (id, "Temperature");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), then
                                            it is actually a warning/error
                                            code. */

recNum = 13;

for (d0 = 0; d0 < 180; d0++) {
   indices[0] = d0;
   for (d1 = 0; d1 < 91; d1++) {
      indices[1] = d1;
      for (d2 = 0; d2 < 10; d2++) {
         indices[2] = d2;
         status = CDFvarGet (id, varN, recNum, indices, &tmp[d0][d1][d2]);
         if (status != CDF_OK) UserStatusHandler (status);
      }
   }
}
.
.
```

4.6.22 CDFvarHyperPut (Standard interface)

```
CDFstatus CDFvarHyperPut(
CDFid id,             /* in -- CDF identifier. */
long  varNum,         /* in -- Variable number. */
long  recStart,       /* in -- Starting record number. */
long  recCount,       /* in -- Number of records. */
long  recInterval,    /* in -- Interval between records. */
long  indices[],      /* in -- Dimension indices of starting value. */
long  counts[],       /* in -- Number of values along each dimension. */
long  intervals[],    /* in -- Interval between values along each dimension. */
void  *buffer);       /* in -- Buffer of values. */
```

`CDFvarHyperPut` is used to write a buffer of one or more values to a variable. It is important to know the variable majority of the CDF before using `CDFvarHyperPut` because the values in the buffer to be written must be in the same majority. `CDFvarInquire` is used to determine the variable majority of a CDF. Section 2.7.8 describes the variable majorities.

The arguments to `CDFvarHyperPut` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.

varNum	The number of the variable to which to write. This number may be determined with a call to CDFvarNum (see Section 4.6.17).
recStart	The record number at which to start writing.
recCount	The number of records to write.
recInterval	The interval between records for subsampling[2] (eg. an interval of 2 means write to every other record).
indices	The indices (within each record) at which to start writing. indices is a 1-dimensional array containing one element per CDF dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
counts	The number of values along each dimension to write. counts is a 1-dimensional array containing one element per CDF dimension. Each element of count specifies the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
intervals	For each dimension, the interval between values for subsampling[3] (eg. an interval of 2 means write to every other value). intervals is a 1-dimensional array containing one element per CDF dimension. Each element of intervals specifies the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
buffer	The buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address buffer are written to the CDF.

Example(s)

The following example writes values to the LATITUDE variable of a 2-dimensional CDF having dimension sizes 360,181. For LATITUDE, the record variance is NOVARY, the dimension variances are NOVARY,VARY, and the data type is CDF_INT2. This example is similar to the example in Section 4.6.20 except that is uses a single call to CDFvarHyperPut rather than numerous calls to CDFvarPut.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                         /* CDF identifier. */
CDFstatus   status;                     /* Returned status code. */
short       lat;                        /* Latitude value. */
short       lats[181];                  /* Buffer of latitude values. */
long        varN;                       /* Variable number. */
long recStart = 0;                      /* Record number. */
long recCount = 1;                      /* Record counts. */
```

[2] "Subsampling" is not the best term to use when writing data but you should know what we mean.
[3] Again, not the best term.

```
long recInterval = 1;                 /* Record interval. */
static long indices[2] = {0,0};       /* Dimension indices. */
static long counts[2] = {1,181};      /* Dimension counts. */
static long intervals[2] = {1,1};     /* Dimension intervals. */
.
.
varN = CDFvarNum (id, "LATITUDE");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), not a
                                            variable number but rather a
                                            warning/error code. */

for (lat = -90; lat <= 90; lat += 1) lats[90+lat] = lat;

status = CDFvarHyperPut (id, varN, recStart, recCount, recInterval,
                         indices, counts, intervals, lats);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.6.23 CDFvarHyperGet (Standard interface)

```
CDFstatus CDFvarHyperGet(
CDFid id,            /* in --  CDF identifier. */
long  varNum,        /* in --  Variable number. */
long  recStart,      /* in --  Starting record number. */
long  recCount,      /* in --  Number of records. */
long  recInterval,   /* in --  Subsampling interval between records. */
long  indices[],     /* in --  Dimension indices of starting value. */
long  counts[],      /* in --  Number of values along each dimension. */
long  intervals[],   /* in --  Subsampling intervals along each dimension. */
void  *buffer);      /* out -- Buffer of values. */
```

`CDFvarHyperGet` is used to read a buffer of one or more values from a variable. It is important to know the variable majority of the CDF before using `CDFvarHyperGet` because the values placed into the buffer will be in that majority. `CDFvarInquire` is used to determine the variable majority of a CDF. Section 2.7.8 describes the variable majorities.

The arguments to `CDFvarHyperGet` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDFcreate` or `CDFopen`.
`varNum`	The number of the variable from which to read. This number may be determined with a call to `CDFvarNum` (see Section 4.6.17).
`recStart`	The record number at which to start reading.
`recCount`	The number of records to read.
`recInterval`	The interval between records for subsampling (eg. an interval of 2 means read

every other record).

`indices` The indices (within each record) at which to start reading. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`counts` The number of values along each dimension to write. `counts` is a 1-dimensional array containing one element per CDF dimension. Each element of `counts` specifies the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`intervals` For each dimension, the interval between values for subsampling (eg. an interval of 2 means read every other value). `intervals` is a 1-dimensional array containing one element per CDF dimension. Each element of `intervals` specifies the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`buffer` The buffer of values read. The majority of the values in this buffer will be the same as that of the CDF. This buffer must be large to hold the values. `CDFvarInquire` would be used to determine the variable's data type and number of elements (of that data type) at each value. The values are read from the CDF and placed into memory starting at address `buffer`.

Example(s)

The following example will read an entire record of data from a variable. The CDF is 3-dimensional with sizes `180,91,10`. The CDF's variable majority is `ROW_MAJOR`. For the variable, the record variance is `VARY`, the dimension variances are `VARY,VARY,VARY`, and the data type is `CDF_REAL4`. This example is similar to the example in Section 4.6.21 except that it uses a single call to `CDFvarHyperGet` rather than numerous calls to `CDFvarGet`.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                         /* CDF identifier. */
CDFstatus   status;                     /* Returned status code. */
float       tmp[180][91][10];           /* Temperature values. */
long        varN;                       /* Variable number. */
long recStart = 13;                     /* Record number. */
long recCount = 1;                      /* Record counts. */
long recInterval = 1;                   /* Record interval. */
static long indices[3] = {0,0,0};       /* Dimension indices. */
static long counts[3] = {180,91,10};    /* Dimension counts. */
static long intervals[3] = {1,1,1};     /* Dimension intervals. */
.
.
varN = CDFvarNum (id, "Temperature");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), then
```

```
                                            it is actually a warning/error
                                            code. */

  status = CDFvarHyperGet (id, varN, recStart, recCount, recInterval,
                           indices, counts, intervals, tmp);
  if (status != CDF_OK) UserStatusHandler (status);
  .
  .
```

Note that if the CDF's variable majority had been `COL_MAJOR`, the `tmp` array would have been declared `float tmp[10][91][180]` to achieve the same results.

4.6.24 CDFlib (Internal interface)

```
CDFstatus CDFlib (...);
```

The Internal interface consists of only one function, `CDFlib`. `CDFlib` can be used to perform all possible operations on a CDF. (In fact, all of the Standard interface functions are implemented using the Internal interface). `CDFlib` must be used to perform operations not possible with the Standard interface functions. These operations would involve CDF features added after the Standard interface functions had been defined (eg. specifying a single-file format for a CDF, specifying a fill value for a variable, or allocating initial records for a variable). Note that `CDFlib` can also be used to perform certain operations more efficiently than with the Standard interface functions.

`CDFlib` takes a variable number of arguments which specify one or more operations to be performed. The operations are performed according to the order of the arguments. Each operation is performed on an item. An item may be either an object (eg. a CDF, variable, or attribute) or a state (eg. a CDF's format or a variable's data type/number of elements). The possible operations and corresponding items (on which to perform those operations) are described in detail later in this section.

Example(s)

The easiest way to explain how to use `CDFlib` would be to start with a few examples. The following example will show how a CDF would be created with the single-file format.

```
  .
  .
  #include "cdf.h"
  .
  .
  CDFid       id;                          /* CDF identifier (handle). */
  CDFstatus   status;                      /* Status returned from CDF
                                              library. */
  static char CDFname[] = {"test1"};       /* Name of the CDF. */
  long        numDims = 2;                 /* Number of dimensions. */
  static long dimSizes[2] = {100,200};     /* Dimension sizes. */
```

```
long        encoding = HOST_ENCODING;   /* Data encoding. */
long        majority = ROW_MAJOR;       /* Variable data majority. */
long        format = SINGLE_FILE;       /* Format of CDF. */
.
.
status = CDFcreate (CDFname, numDims, dimSizes, encoding, majority, &id);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (SELECT_, CDF_, id,
                 PUT_, CDF_FORMAT_, format,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

The call to `CDFcreate` created the CDF as expected but the default format is `MULTI_FILE` (a separate file for each variable in the CDF). The call to `CDFlib` is then used to change the format to `SINGLE_FILE` (which must be done before any variables are created in the CDF).

The arguments to `CDFlib` in this example are explained as follows.

`SELECT_`	The first operation to be performed. In this case an item is going to be selected. `SELECT_` is defined in `cdf.h` (as are all CDF constants).
`CDF_`	The item to be selected — the current CDF (the CDF on which subsequent operations are to be performed).
`id`	The identifier of the CDF being selected.
`PUT_`	The next operation to be performed. In this case an item is going to be put (a new CDF format).
`CDF_FORMAT_`	The item to be put. In this case it is the CDF's format.
`format`	The actual format for the CDF. Depending on the item being put, one or more arguments would have been necessary. In this case only one argument is necessary. `format` contains the value `SINGLE_FILE`.
`NULL_`	This argument could have been one of two things. It could have been another item to put (followed by the arguments required for that item) or it could have been a new operation to perform. In this case it is a new operation to perform — the `NULL_` operation. `NULL_` indicates the end of the call to `CDFlib`. Specifying `NULL_` at the end of the argument list is required because not all compilers/operating systems provide the ability for a called function to determine how many arguments were passed in by the calling function.

The next example shows how the same CDF could have been created using only one call to `CDFlib` (the declarations would be the same).

```
.
.
```

```
status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id,
                 PUT_, CDF_ENCODING_, encoding,
                       CDF_MAJORITY_, majority,
                       CDF_FORMAT_, format,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

The purpose of each argument is as follows.

`CREATE_`	The first operation to be performed. In this case something will be created.
`CDF_`	The item to be created — a CDF in this case. There are four required arguments which must follow. When a CDF is created (with `CDFlib`), the format defaults to multi-file (`MULTI_FILE`), the data encoding defaults to host (`HOST_ENCODING`), and the variable majority default to row major (`ROW_MAJOR`).
`CDFname`	The name of the CDF.
`numDims`	The number of dimensions in the CDF.
`dimSizes`	The dimension sizes.
`id`	The identifier to be used when referencing the created CDF in subsequent operations.
`PUT_`	This argument could have been one of two things. Another item to create or a new operation to perform. In this case it is another operation to perform — something will be put.
`CDF_ENCODING_`	The item to be put — in this case the CDF's encoding. Note that the CDF did not have to be selected. It was automatically selected as the current CDF when it was created.
`encoding`	The encoding to be put to the CDF.
`CDF_MAJORITY_`	This argument could have been one of two things. Another item to put or a new operation to perform. In this case it is another item to put — the CDF's majority.
`majority`	The majority to be put to the CDF.
`CDF_FORMAT_`	Once again this argument could have been either another item to put or a new operation to perform. It is another item to put — the CDF's format.
`format`	The format to be put to the CDF.
`NULL_`	This argument could have been either another item to put or a new operation to perform. Here it is another operation to perform — the `NULL_` operation which ends the call to `CDFlib`.

Note that the operations are performed in the order that they appear in the argument list. The CDF had to be created before the encoding, majority, and format could be specified (put).

Current Objects/States (Items)

The use of CDFlib requires that an application be aware of the current objects/states maintained by the CDF library. The following objects/states are used by the CDF library when performing operations.

CDF

A CDF operation is always performed on the current CDF. The current CDF is implicitly selected whenever a CDF is opened or created. The current CDF may be explicitly selected using the <SELECT_,CDF_>[4] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.

NOTE

The current CDF is not maintained by the CDF library from one call to the next of CDFlib. A current CDF (or CDFs) must be selected in each call to CDFlib (if an operation is to be performed on a CDF).

variable

A variable operation is always performed on the current variable in the current CDF. For each open CDF a current variable is maintained. This current variable is implicitly selected when a variable is created (in the current CDF) or it may be explicitly selected with the <SELECT_,VAR_> operation. There is no current variable in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

attribute

An attribute operation is always performed on the current attribute in the current CDF. For each open CDF a current attribute is maintained. This current attribute is implicitly selected when an attribute is created (in the current CDF) or it may be explicitly selected with the <SELECT_,ATTR_> operation. There is no current attribute in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

entry number

An attribute entry operation is always performed on the current entry number in the current CDF for the current attribute in that CDF. For each open CDF a current entry number is maintained. This current entry number must be explicitly selected with the <SELECT_,ENTRY_> operation (there is no implicit or default selection of the current entry number for a CDF). Note that the current entry number is maintained for a CDF (not each attribute) — it applies to all of the attributes in that CDF.

record number

A variable read or write operation is always performed at (for regular reads and writes) or starting at (for hyper reads and writes) the current record number for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current record number is initialized to zero (0). It may be explicitly selected using the <SELECT_,CDF_RECNUMBER_> operation. Note that the current record number is maintained for a CDF (not each

[4] This notation is used to specify an operation to be performed on an item. The syntax is: <operation,item>.

variable) — it applies to all of the variables in that CDF.

record count A variable hyper read or write operation is always performed using the current record count for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current record count is initialized to one (1). It may be explicitly selected using the <SELECT_,CDF_RECCOUNT_> operation. Note that the current record count is maintained for a CDF (not each variable) — it applies to all of the variables in that CDF.

record interval A variable hyper read or write operation is always performed using the current record interval for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current record interval is initialized to one (1). It may be explicitly selected using the <SELECT_,CDF_RECINTERVAL_> operation. Note that the current record interval is maintained for a CDF (not each variable) — it applies to all of the variables in that CDF.

dimension indices A variable read or write operation is always performed at (for regular reads and writes) or starting at (for hyper reads and writes) the current dimension indices for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current dimension indices are initialized to zeroes (0,0,...). They may be explicitly selected using the <SELECT_,CDF_DIMINDICES_> operation. Note that the current dimension indices are maintained for a CDF (not each variable) — they apply to all of the variables in that CDF. For 0-dimensional CDFs the current dimension indices are not applicable.

dimension counts A variable hyper read or write operation is always performed using the current dimension counts for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current dimension counts are initialized to the CDF's dimension sizes (which specifies the entire record array). They may be explicitly selected using the <SELECT_,CDF_DIMCOUNTS_> operation. Note that the current dimension counts are maintained for a CDF (not each variable) — they apply to all of the variables in that CDF. For 0-dimensional CDFs the current dimension counts are not applicable.

dimension intervals A variable hyper read or write operation is always performed using the current dimension intervals for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current dimension intervals are initialized to ones (1,1,...). They may be explicitly selected using the <SELECT_,CDF_DIMINTERVALS_> operation. Note that the current dimension intervals are maintained for a CDF (not each variable) — they apply to all of the variables in that CDF. For 0-dimensional CDFs the current dimension intervals are not applicable.

status code When inquiring the explanation of a CDF status code, the text returned is always for the current status code. One current status code is maintained for the entire CDF library (regardless of the number of open CDFs). The current status code may be selected using the <SELECT_,CDF_STATUS_> operation. There is no default current status code. Note that the current status code is NOT the status code from the last operation performed).

NOTE

The current status code is not maintained by the CDF library from one call to the next of `CDFlib`.

Returned Status

`CDFlib` returns a status code of type `CDFstatus`. Since more than one operation may be performed with a single call to `CDFlib`, the following rules apply.

1. The first error detected aborts the call to `CDFlib` and the corresponding status code is returned.
2. In the absence of any errors, the status code for the last warning detected is returned.
3. In the absence of any errors or warnings, the status code for the last informational condition is returned.
4. In the absence of any errors, warnings, or informational conditions, `CDF_OK` is returned.

Section 4.7 explains how to interpret status codes and Appendix B lists the possible status codes.

Indentation/Style

Indentation should be used to make calls to `CDFlib` readable. The following example shows a call to `CDFlib` using proper indentation.

```
status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id,
                 PUT_, CDF_FORMAT_, format,
                       CDF_MAJORITY_, majority,
                 CREATE_, ATTR_, attrName, scope, &attrNum,
                          VAR_, varName, dataType, numElements,
                                recVary, dimVarys, &varNum,
                 NULL_);
```

Note that the operations (`CREATE_`, `PUT_`, and `NULL_`) are indented the same and that the items (`CDF_`, `CDF_FORMAT_`, `CDF_MAJORITY_`, `ATTR_`, and `VAR_`) are indented the same under their corresponding operations.

The following example shows the same call to `CDFlib` without the proper indentation.

```
status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id, PUT_,
                 CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
                 ATTR_, attrName, scope, &attrNum, VAR_, varName, dataType,
                 numElements, recVary, dimVarys, &varNum, NULL_);
```

The need for proper indentation to ensure the readability of your applications should be obvious.[5]

Syntax

`CDFlib` has a variable number of arguments. There must always be at least one argument. The maximum number of arguments is not limited by CDF but rather the C compiler and operating system being used. Under normal circumstances that limit would never be reached (or even approached). Note also that a call to `CDFlib` with a large number of arguments can always be broken up into two or more calls to `CDFlib` with fewer arguments.

Operations/items/arguments...

The syntax for `CDFlib` is as follows...

```
status = CDFlib (op1, item1, arg1, arg2, ...argN,
                      item2, arg1, arg2, ...argN,
                      .
                      .
                      itemN, arg1, arg2, ...argN,
                 op2, item1, arg1, arg2, ...argN,
                      item2, arg1, arg2, ...argN,
                      .
                      .
                      itemN, arg1, arg2, ...argN,
                 .
                 .
                 .
                 opN, item1, arg1, arg2, ...argN,
                      item2, arg1, arg2, ...argN,
                      .
                      .
                      itemN, arg1, arg2, ...argN,
                 NULL_);
```

where `op`x is an operation to perform, `item`x is the item on which to perform the operation, and `arg`x is a required argument for the operation. The `NULL_` operation must be used to end the call to `CDFlib`. The completion status is returned by `CDFlib`.

The supported operations are as follows.

`CREATE_`	Used to create an item.
`OPEN_`	Used to open an item.
`DELETE_`	Used to delete an item.
`CLOSE_`	Used to close an item.
`SELECT_`	Used to select the value of an item.

[5] If it isn't, call us, we can help you.

CONFIRM_	Used to confirm the value of an item.
PUT_	Used to put (write) something to an item.
GET_	Used to get (read) something from an item.

For each operation, the supported items and required arguments are as follows.

CREATE_ The following items may be created.

CDF_ A new CDF will be created. It is illegal to create a CDF that already exists. The created CDF implicitly becomes the current CDF. Required arguments are...

in: char *CDFname	Pathname of the CDF to be created. This can be at most CDF_PATHNAME_LEN characters (excluding the NUL).
in: long numDims	Number of dimensions. This can be as few as zero (0) and at most CDF_MAX_DIMS.
in: long dimSizes[]	Dimension sizes. dimSizes is a 1-dimensional array containing numDims elements. Each element of dimSizes specifies the corresponding dimension size. Each dimension size must be greater than zero (0). For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
out: CDFid *id	CDF identifier to be used in subsequent operations on the CDF.

A CDF is created with a default format of multi-file (MULTI_FILE), a default data encoding of host (HOST_ENCODING), and a default variable majority of row major (ROW_MAJOR). These defaults can be changed with the corresponding PUT_ operations (of CDF_FORMAT_, CDF_ENCODING_, and CDF_MAJORITY_ respectively).

A CDF must be closed with <CLOSE_,CDF_> to ensure that the CDF will be correctly written to disk.

VAR_ A new variable will be created in the current CDF. A variable with the same name must not already exist in the CDF. The created variable implicitly becomes the current variable (in the current CDF). Required arguments are...

in: char *varName	Name of the variable to be created. This can be at most CDF_VAR_NAME_LEN characters (excluding the NUL). Variable names are case-sensitive.
in: long dataType	Data type of the new variable. Specify one of the data types described in Section 4.5.4.

in: long numElements	Number of elements of the data type at each variable value. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in each string (an array of characters). A string exists at each variable value. For the non-character string data types this must be one (1) — arrays of values are not allowed for non-character string data types.
in: long recVary	Record variance. Specify one of the variances described in Section 4.5.7.
in: long dimVarys[]	Dimension variances. dimVarys is a 1-dimensional array containg one element per CDF dimension. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.5.7.
out: long *varNum	Number assigned to the new variable. This number must be used in subsequent CDF function calls when referring to this variable. An existing variable's number may also be determined with the <GET_,VAR_NUMBER_> operation.

ATTR_ A new attribute will be created in the current CDF. An attribute with the same name must not already exist in the CDF. The created attribute implicitly becomes the current attribute (in the current CDF). Required arguments are...

in: char *attrName	Name of the attribute to be created. This can be at most CDF_ATTR_NAME_LEN characters (excluding the NUL). Attribute names are case-sensitive.
in: long scope	Scope of the new attribute. Specify one of the scopes described in Section 4.5.8.
out: long *attrNum	Number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may also be determined with the <GET_,ATTR_NUMBER_> operation.

OPEN_ The follow item(s) may be opened.

CDF_ Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are...

in: char *CDFname	Pathname of the CDF to be opened. This can be at most CDF_PATHNAME_LEN

	characters (excluding the NUL).
`out: CDFid *id`	CDF identifier to be used in subsequent operations on the CDF.

`DELETE_` The following item(s) may be deleted.

`CDF_` Deletes the current CDF. A CDF must be opened before it can be deleted. When a CDF is deleted, there is no longer a current CDF. There are no arguments.

`CLOSE_` The following items may be closed.

`CDF_` Closes the current CDF. When a CDF is closed, there is no longer a current CDF. A CDF must be closed to ensure that it will be properly written to disk. There are no arguments.

`VAR_` Closes the current variable (in the current CDF). There are no arguments.

`SELECT_` The following items may be selected.

`CDF_` Explicitly selects the current CDF. Required arguments are...

`in: CDFid id`	Identifier of the CDF. This identifier must have been initialized by a successful `<CREATE_,CDF_>` or `<OPEN_,CDF_>` operation.

`CDF_STATUS_` Selects the current status code. Required arguments are...

`in: CDFstatus status`	CDF status code.

`VAR_` Explicitly selects the current variable (in the current CDF) by number. Required arguments are...

`in: long varNum`	Variable number.

`VAR_NAME_` Explicitly selects the current variable (in the current CDF) by name. Required arguments are...

`in: char *varName`	Variable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the NUL).

`CDF_RECNUMBER_` Selects the current record number for all variables in the current CDF. Required arguments are...

`in: long recNum`	Record number.

`CDF_RECCOUNT_` Selects the current record count for all variables in the current CDF. Required arguments are...

`in: long recCount`	Record count.

CDF_RECINTERVAL_ Selects the current record interval for all variables in the current CDF. Required arguments are...

in:	long recInterval	Record interval.

CDF_DIMINDICES_ Selects the current dimension indices for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	long indices[]	Dimension indices. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_DIMCOUNTS_ Selects the current dimension counts for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	long counts[]	Dimension counts. `counts` is a 1-dimensional array containing one element per CDF dimension. Each element of `counts` specifies the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_DIMINTERVALS_ Selects the current dimension intervals for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	long intervals[]	Dimension intervals. `intervals` is a 1-dimensional array containing one element per CDF dimension. Each element of `intervals` specifies the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

VAR_SEQPOS_ Selects the current position for sequential access for the current variable (in the current CDF). Note that the current position is selected for each variable individually. Required arguments are...

in:	long recNum	Record number.
in:	long indices[]	Dimension indices. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

ATTR_ Explicitly selects the current attribute (in the current CDF) by number. Required arguments are...

in: `long attrNum`	Attribute number.

ATTR_NAME_ Explicitly selects the current attribute (in the current CDF) by name. Required arguments are...

in: `char *attrName`	Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the NUL).

ENTRY_ Selects the current entry number for all attributes in the current CDF. Required arguments are...

in: `long entryNum`	Entry number.

CONFIRM_ The following items may be confirmed.

CDF_NAME_ Confirms the pathname of the current CDF. Required arguments are...

out: `char *CDFname`	Pathname of the CDF. This character string must be large enough to hold `CDF_PATHNAME_LEN + 1` characters (including the NUL terminator).

VAR_ Confirms the current variable (in the current CDF). Required arguments are...

out: `long *varNum`	Variable number.

CDF_RECNUMBER_ Confirms the current record number for all variables in the current CDF. Required arguments are...

out: `long *recNum`	Record number.

CDF_RECCOUNT_ Confirms the current record count for all variables in the current CDF. Required arguments are...

out: `long *recCount`	Record count.

CDF_RECINTERVAL_ Confirms the current record interval for all variables in the current CDF. Required arguments are...

out: `long *recInterval`	Record interval.

CDF_DIMINDICES_ Confirms the current dimension indices for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

out: `long indices[]`	Dimension indices. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices`

receives the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`CDF_DIMCOUNTS_` Confirms the current dimension counts for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

`out: long counts[]`	Dimension counts. `counts` is a 1-dimensional array containing one element per CDF dimension. Each element of `counts` receives the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`CDF_DIMINTERVALS_` Confirms the current dimension intervals for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

`out: long intervals[]`	Dimension intervals. `intervals` is a 1-dimensional array containing one element per CDF dimension. Each element of `intervals` receives the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`VAR_SEQPOS_` Confirms the current position for sequential access for the current variable (in the current CDF). Required arguments are...

`out: long *recNum`	Record number.
`out: long indices[]`	Dimension indices. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` receives the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`ATTR_` Confirms the current attribute (in the current CDF). Required arguments are...

`out: long *attrNum`	Attribute number.

`ENTRY_` Confirms the current entry number for all attributes in the current CDF. Required arguments are...

`out: long *entryNum`	Entry number.

`PUT_` The following items may be written (put).

CDF_ARRAY_ Respecifies the dimensionality of the current CDF. A CDF's dimensionality may not be changed after any variables have been created. Required arguments are...

in:	`long numDims`	Number of dimensions. The number of dimensions must be in the range from zero (0) to `CDF_MAX_DIMS` (inclusive).
in:	`long dimSizes[]`	Dimension sizes. `dimSizes` is a 1-dimensional array containing one element per CDF dimension. Each element of `dimSizes` specifies the corresponding dimension size. Each dimension size must be greater than zero (0). For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_ENCODING_ Respecifies the data encoding of the current CDF. A CDF's data encoding may not be changed after any variable values (including the fill value) or attribute entries have been written. Required arguments are...

in:	`long encoding`	Data encoding. Specify one of the encodings described in Section 4.5.5.

CDF_MAJORITY_ Respecifies the variable majority of the current CDF. A CDF's variable majority may not be changed after any variable values have been written. Required arguments are...

in:	`long majority`	Variable majority. Specify one of the majorities described in Section 4.5.6.

CDF_FORMAT_ Respecifies the format of the current CDF. A CDF's format may not be changed after any variables have been created. Required arguments are...

in:	`long format`	CDF format. Specify one of the formats described in Section 4.5.3.

VAR_NAME_ Renames the current variable (in the current CDF). A variable with the same name must not already exist in the CDF. Required arguments are...

in:	`char *varName`	Name of the variable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the NUL).

VAR_DATASPEC_ Respecifies the data specification (data type and number of elements) of the current variable (in the current CDF). A variable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any variable values (including the fill value) have been written. Data specifications are considered equivalent if the data types are equivalent (see Section 2.5.5) and the number of elements are the same. Required arguments are...

in:	`long dataType`	Variable data type. Specify one of the data types described in Section 4.5.4.
in:	`long numElements`	Number of elements of the data type at each variable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in each string (an array of characters). A string exists at each variable value. For the non-character string data types this must be one (1) — arrays of values are not allowed for non-character string data types.

`VAR_RECVARY_` Respecifies the record variance of the current variable (in the current CDF). A variable's record variance may not be changed if any values have been written to that variable (except for an explicit fill value — it may have been written). Required arguments are...

in:	`long recVary`	Record variance. Specify one of the variances described in Section 4.5.7.

`VAR_DIMVARYS_` Respecifies the dimension variances of the current variable (in the current CDF). A variable's dimension variances may not be changed if any values have been written to that variable (except for an explicit fill value — it may have been written). For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	`long dimVarys[]`	Dimension variances. `dimVarys` is a 1-dimensional array containing one element per CDF dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension, specify one of the variances described in Section 4.5.7.

`VAR_FILLVALUE_` Specifies the fill value for the current variable (in the current CDF). A variable's fill value may be specified (or respecified) at any time without affecting already written values (including where fill values were used). Section 2.7.11 describes variable fill values. Required arguments are...

in:	`void *value`	Fill value. The fill value is written to the CDF from memory address `value`.

`VAR_INITIALRECS_` Specifies the number of records to initially write to the current variable (in the current CDF). This may be specified only once per variable and before any other records have been written to that variable. If a fill value has not yet been specified, the default is used (see Section 2.7.11). If a fill value has been specified, that value is written to the records. Section 2.7.4 describes initial records. Required arguments are...

in:	`long NinitialRecs`	Number of records to write.

VAR_EXTENDRECS_ Specifies the number of physical records to extend the current variable (in the current CDF) whenever the variable needs to be extended. Section 2.7.4 describes extend records. Required arguments are...

`in:`	`long NextendRecs`	Number of records to extend.

VAR_DATA_ Writes one value to the current variable (in the current CDF). The value is written at the current record number and current indices (for the current CDF). Required arguments are...

`in:`	`void *value`	Value. The value is written to the CDF from memory address `value`.

VAR_HYPERDATA_ Writes one or more values to the current variable (in the current CDF). The values are written based on the current record number, count, and interval and the current dimension indices, counts, and intervals (for the current CDF). Required arguments are...

`in:`	`void *buffer`	Values. The values starting at memory address `buffer` are written to the CDF.

VAR_SEQDATA_ Writes one value to the current variable (in the current CDF) at the current sequential position for that variable. After the write, the sequential position is automatically incremented to the next value (crossing a record boundary if necessary). If the sequential position is past the last record for the variable, the variable is extended as necessary. Required arguments are...

`in:`	`void *value`	Value. The value is written to the CDF from memory address `value`.

ATTR_SCOPE_ Respecifies the scope for the current attribute (in the current CDF). Required arguments are...

`in:`	`long scope`	Attribute scope. Specify one of the scopes described in Section 4.5.8.

ATTR_NAME_ Renames the current attribute (in the current CDF). An attribute with the same name must not already exist in the CDF. Required arguments are...

`in:`	`char *attrName`	Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the NUL).

ENTRY_DATA_ Writes an entry to the current attribute at the current entry number (in the current CDF). An existing entry may be overwritten with a new entry having the same data specification (data type and number of elements) or a different data specification. Required arguments are...

`in:`	`long dataType`	Entry data type. Specify one of the data types described in Section 4.5.4.

in: long numElements	Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string (an array of characters). For all other data types, this is the number of elements in an array of that data type.
in: void *value	Value(s). The entry value is written to the CDF from memory address value.

GET_ The following items may be read (gotten).

CDF_NUMDIMS_ Inquires the number of dimensions in the current CDF. Required arguments are...

out: long *numDims	Number of dimensions.

CDF_DIMSIZES_ Inquires the size of each dimension in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

out: long dimSizes[]	Dimension sizes. dimSizes is a 1-dimensional array containing one element per CDF dimension. Declaring dimSizes as long dimSizes[CDF_MAX_DIMS] would work in all cases. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_ENCODING_ Inquires the data encoding of the current CDF. Required arguments are...

out: long *encoding	Data encoding. The encodings are described in Section 4.5.5.

CDF_MAJORITY_ Inquires the variable majority of the current CDF. Required arguments are...

out: long *majority	Variable majority. The majorities are described in Section 4.5.6.

CDF_FORMAT_ Inquires the format of the current CDF. Required arguments are...

out: long *format	CDF format. The formats are described in Section 4.5.3.

CDF_COPYRIGHT_ Reads the copyright notice for the CDF library which created the current CDF. Required arguments are...

out: char *copyRight	CDF copyright text. This character string must be large enough hold

CDF_COPYRIGHT_LEN + 1 characters (including the NUL terminator).

`LIB_COPYRIGHT_`

Reads the copyright notice of the CDF library being used. Required arguments are...

`out: char *copyRight`	CDF library copyright text. This character string must be large enough hold CDF_COPYRIGHT_LEN + 1 characters (including the NUL terminator).

`CDF_NUMVARS_`

Inquires the number of variables in the current CDF. Required arguments are...

`out: long *numVars`	Number of variables.

`CDF_NUMATTRS_`

Inquires the number of attributes in the current CDF. Required arguments are...

`out: long *numAttrs`	Number of attributes.

`CDF_MAXREC_`

Inquires the maximum record number in the current CDF Note that this is not the number of records but rather the maximum record number (which is one less than the number of records). A value of negative one (-1) indicates that the CDF contains no records. The maximum record number for an individual variable may be inquired using <GET_,VAR_MAXREC_>. Required arguments are...

`out: long *maxRec`	Maximum record number in the CDF.

`CDF_VERSION_`

Inquires the version number of the CDF library which created the current CDF. Required arguments are...

`out: long *version`	Version number.

`CDF_RELEASE_`

Inquires the release number of the CDF library which created the current CDF. Required arguments are...

`out: long *release`	Release number.

`CDF_INCREMENT_`

Inquires the incremental number of the CDF library which created the current CDF. Required arguments are...

`out: long *increment`	Incremental number.

`LIB_VERSION_`

Inquires the version number of the CDF library being used. Required arguments are...

`out: long *version`	Version number.

`LIB_RELEASE_`

Inquires the release number of the CDF library being used. Required arguments are...

`out: long *release`	Release number.

`LIB_INCREMENT_` Inquires the incremental number of the CDF library being used. Required arguments are...

`out: long *increment`	Incremental number.

`LIB_subINCREMENT_` Inquires the subincremental character of the CDF library being used. Required arguments are...

`out: char *subincrement`	Subincremental character.

`VAR_NAME_` Inquires the name of the current variable (in the current CDF). Required arguments are...

`out: char *varName`	Name of the variable. This character string must be large enough to hold `CDF_VAR_NAME_LEN + 1` characters (including the NUL).

`VAR_DATATYPE_` Inquires the data type of the current variable (in the current CDF). Required arguments are...

`out: long *dataType`	Variable data type. The data types are described in Section 4.5.4.

`VAR_NUMELEMS_` Inquires the number of elements (of the data type) for the current variable (in the current CDF). Required arguments are...

`out: long *numElements`	Number of elements of the data type at each variable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (each variable value consists of the entire string). For all other data types, this will always be one (1) — multiple elements at each variable value are not allowed for non-character string data types.

`VAR_RECVARY_` Inquires the record variance of the current variable (in the current CDF). Required arguments are...

`out: long *recVary`	Record variance. The variances are described in Section 4.5.7.

`VAR_DIMVARYS_` Inquires the dimension variances of the current variable (in the current CDF). For 0-dimensional CDFs this operation is not applicable. Required arguments are...

`out: long dimVarys[]`	Dimension variances. `dimVarys` is a 1-dimensional array containing one element per CDF dimension. Each element of `dimVarys` receives the corresponding dimension variance. The variances are described in Section 4.5.7. For 0-dimensional

		CDFs this argument is ignored (but a place holder is necessary).

VAR_NUMBER_ Gets the number of the named variable (in the current CDF). Required arguments are...

`in: char *varName`	The variable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the NUL).
`out: long *varNum`	The variable number.

VAR_FILLVALUE_ Inquires the fill value of the current variable (in the current CDF). Required arguments are...

`out: void *value`	Fill value. This buffer must be large enough to hold the fill value. The fill value is read from the CDF and placed in memory at address `value`.

VAR_EXTENDRECS_ Inquires the number of extend records for the current variable (in the current CDF). Extend records are described in Section 2.7.4. Required arguments are...

`out: long *NextendRecs`	Number of records to extend.

VAR_MAXREC_ Inquires the maximum record number for the current variable (in the current CDF). This will always be less than or equal to the maximum record number in the CDF. For variables with a record variance of `NOVARY`, this will be at most zero (0). A value of negative one (-1) indicates that no records have been written to the variable. Required arguments are...

`out: long *varMaxRec`	Maximum record number for the variable.

VAR_DATA_ Reads a value from the current variable (in the current CDF). The value is read at the current record number and current indices (for the current CDF). Required arguments are...

`out: void *value`	Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

VAR_HYPERDATA_ Reads one or more values from the current variable (in the current CDF). The values are read based on the current record number, count, and interval and the current dimension indices, counts, and intervals (for the current CDF). Required arguments are...

`out: void *buffer`	Values. This buffer must be large enough to hold the values. The values are read from the CDF and placed into memory starting at address `buffer`.

`VAR_SEQDATA_` Reads one value from the current variable (in the current CDF) at the current sequential position for that variable. After the read, the sequential position is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the sequential position is past the last record for the variable. Required arguments are...

`out: void *value`	Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

`ATTR_SCOPE_` Inquires the scope of the current attribute (in the current CDF). Required arguments are...

`out: long *scope`	Attribute scope. The scopes are described in Section 4.5.8.

`ATTR_NAME_` Inquires the name of the current attribute (in the current CDF). Required arguments are...

`out: char *attrName`	Attribute name. This character string must be large enough to hold `CDF_ATTR_NAME_LEN + 1` characters (including the NUL).

`ATTR_NUMBER_` Gets the number of the named attribute (in the current CDF). Required arguments are...

`in: char *attrName`	Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the NUL).
`out: long *attrNum`	The attribute number.

`ATTR_MAXENTRY_` Inquires the maximum entry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of entries for the attribute. Required arguments are...

`out: long *maxEntry`	The maximum entry number for the attribute.

`ATTR_NUMENTRIES_` Inquires the number of entries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum entry number used. Required arguments are...

`out: long *numEntries`	The number of entries for the attribute.

`ENTRY_DATATYPE_` Inquires the data type for the current entry number for the current attribute (in the current CDF). Required arguments are...

`out: long *dataType`	Entry data type. The data types are described in Section 4.5.4.

ENTRY_NUMELEMS_ Inquires the number of elements (of the data type) for the current entry number for the current attribute (in the current CDF). Required arguments are...

`out: long *numElements` Number of elements of the data type. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (an array of characters). For all other data types, this is the number of elements in an array of that data type.

ENTRY_DATA_ Reads the entry from the current attribute at the current entry number (in the current CDF). Required arguments are...

`out: void *value` Value. This buffer must be large to hold the value. The value is read from the CDF and placed into memory at address `value`.

STATUS_TEXT_ Inquires the explanation text for the current status code. Note that the current status code is NOT the status from the last operation performed. Required arguments are...

`out: char *text` Text explaining the status code. This string must be large enough to hold `CDF_STATUSTEXT_LEN + 1` characters.

More Examples

Several more examples of the use of `CDFlib` follow.

Variable creation In this example a variable will be created with a fill value being specified, initial records will be written, and the number of records to extend the variable when necessary being specified. Note that the fill value was specified before the initial records. This results in the specified fill value being written. Had the fill value not been specified first, the initial records would have been written with the default fill value.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                         /* CDF identifier (handle). */
CDFstatus   status;                     /* Status returned from CDF
                                           library. */
long        dimVarys[2];                /* Dimension variances. */
long        varNum;                     /* Variable number. */
float       fillValue = -999.9;         /* Fill value. */
```

```
.
.
dimVarys[0] = VARY;
dimVarys[1] = VARY;
status = CDFlib (SELECT_, CDF_, id,
                 CREATE_, VAR_, "HUMIDITY", CDF_REAL4, 1, VARY,
                                dimVarys, &varNum,
                 PUT_, VAR_FILLVALUE_, &fillValue,
                       VAR_INITIALRECS_, 500,
                       VAR_EXTENDRECS_, 50,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Variable creation (character string variable) In this example a character string variable will be created with a fill value being specified.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                          /* CDF identifier (handle). */
CDFstatus   status;                      /* Status returned from CDF
                                            library. */
long        dimVarys[1];                 /* Dimension variances. */
long        varNum;                      /* Variable number. */
static char fillValue = "**********";   /* Fill value. */
.
.
dimVarys[0] = VARY;
status = CDFlib (SELECT_, CDF_, id,
                 CREATE_, VAR_, "Station", CDF_CHAR, 10, NOVARY,
                                dimVarys, &varNum,
                 PUT_, VAR_FILLVALUE_, fillValue,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Hyper read with subsampling In this example a variable will be subsampled in a 2-dimensional CDF having dimension sizes `100,200`. The CDF is row major and the data type of the variable is `CDF_UINT2`.

```
.
.
#include "cdf.h"
.
```

```
.
CDFid          id;                   /* CDF identifier (handle). */
CDFstatus      status;               /* Status returned from CDF library. */
unsigned short values[50][100];      /* Buffer to receive values. */
long           recCount = 1;         /* Record count, one record per hyper
                                        get. */
long           recInterval = 1;      /* Record interval, set to one to
                                        indicate contiguous records (really
                                        meaningless since record count is
                                        one). */
static long    indices[2] = {0,0};   /* Dimension indices, start each read
                                        at 0,0 of the array. */
static long    counts[2] = {50,100}; /* Dimension counts, half of the
                                        values along each dimension will
                                        be read. */
static long    intervals[2] = {2,2}; /* Dimension intervals, every other
                                        value along each dimension will be
                                        read. */
long           recNum;               /* Record number. */
long           maxRec;               /* Maximum record in the CDF - this
                                        was determined with a call to
                                        CDFinquire. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                          VAR_NAME_, "BRIGHTNESS",
                          CDF_RECCOUNT_, recCount,
                          CDF_RECINTERVAL_, recInterval,
                          CDF_DIMINDICES_, indices,
                          CDF_DIMCOUNTS_, counts,
                          CDF_DIMINTERVALS_, intervals,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);

for (recNum = 0; recNum <= maxRec; recNum++) {
   status = CDFlib (SELECT_, CDF_, id,
                             CDF_RECNUMBER_, recNum,
                    GET_, VAR_HYPERDATA_, values,
                    NULL_);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
   /* process values */
   .
   .
}
.
.
```

Attribute renaming In this example the attribute named `Tmp` will be renamed `TMP`.

```
.
.
#include "cdf.h"
.
.
CDFid          id;                    /* CDF identifier (handle). */
CDFstatus      status;                /* Status returned from CDF library. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                          ATTR_NAME_, "Tmp",
                 PUT_, ATTR_NAME, "TMP",
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Sequential Access In this example the values for a variable will be averaged. The values will be read using the sequential access method (see Section 2.7.10). Each value in each record will be read and averaged. It is assumed that the data type of the variable has been determined to be `CDF_REAL4`.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                    /* CDF identifier (handle). */
CDFstatus status;                /* Status returned from CDF library. */
long      varNum;                /* Variable number. */
long      recNum = 0;            /* Record number, start at first record. */
long      indices[2] = {0,0};    /* Dimension indices. */
float     value;                 /* Variable value. */
double    sum = 0.0;             /* Sum of all variable values. */
long      count = 0;             /* Number of variable values. */
float     ave;                   /* Average value for variable. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                 GET_, VAR_NUMBER_, "FLUX", &varNum,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (SELECT_, CDF_, id,
                          VAR_, varNum,
                          VAR_SEQPOS_, recNum, indices,
                 GET_, VAR_SEQDATA_, &value,
                 NULL_);

while (status >= CDF_OK) {
```

```
    sum += value;
    count++;
    status = CDFlib (SELECT_, CDF_, id,
                     GET_, VAR_SEQDATA_, &value,
                     NULL_);
  }

  if (status != END_OF_VAR) UserStatusHandler (status);

  ave = sum / count;
  .
  .
```

Attribute entry writes In this example a set of attribute entries for a particular variable will be written.

```
.
.
#include "cdf.h"
.
.
CDFid          id;                       /* CDF identifier (handle). */
CDFstatus      status;                   /* Status returned from CDF
                                            library. */
static float   scale[2] = {-90.0,90.0};  /* Scale, minimum/maximum. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                          VAR_NAME_, "LATITUDE",
                          ATTR_NAME_, "FIELDNAM",
                 PUT_, ENTRY_DATA_, CDF_CHAR, 20, "Latitude            ",
                 SELECT_, ATTR_NAME_, "SCALE",
                 PUT_, ENTRY_DATA_, CDF_REAL4, 2, scale,
                 SELECT_, ATTR_NAME_, "UNITS",
                 PUT_, ENTRY_DATA_, CDF_CHAR, 20, "Degrees north       ",
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

4.7 Interpreting CDF Status Codes

Most CDF functions return a status code of type `CDFstatus`. The symbolic names for these codes are defined in `cdf.h` and should be used in your applications rather than using the true numeric values. Appendix B explains each status code. When testing the status code returned from a CDF function, the following rules apply.

`status > CDF_OK`	Indicates successful completion but some additional information is provided. These are informational codes.
`status = CDF_OK`	Indicates successful completion.
`CDF_WARN < status < CDF_OK`	Indicates that the function completed but probably not as expected. These are warning codes.
`status < CDF_WARN`	Indicates that the function did not complete. These are error codes.

The following example shows how you could check the status code returned from CDF functions.

```
CDFstatus status;
.
.
status = CDFfunction (...);   /* any CDF function returning CDFstatus */
if (status != CDF_OK) {
  UserStatusHandler (status, ...);
  .
  .
}
```

In your own status handler you can take whatever action is appropriate to the application. An example status handler follows. Note that no action is taken in the status handler if the status is `CDF_OK`.

```
#include <stdio.h>

#include "cdf.h"

void UserStatusHandler (status)
CDFstatus status;
{
char message[CDF_ERRTEXT_LEN+1];

if (status < CDF_WARN) {
  printf ("An error has occurred, halting...\n");
  CDFerror (status, message);
  printf ("%s\n", message);
  exit (status);
}
else
  if (status < CDF_OK) {
    printf ("Warning, function may not have completed as expected...\n");
    CDFerror (status, message);
    printf ("%s\n", message);
  }
  else
    if (status > CDF_OK) {
      printf ("Function completed successfully, but be advised that...\n");
```

```
        CDFerror (status, message);
        printf ("%s\n", message);
      }
  return;
}
```

Explanations for all CDF status codes are available to your applications through the function `CDFerror`. `CDFerror` writes to a text string an explanation of a given status code.

4.8 CDF_EPOCH Values

Two functions exist which compute and breakdown `CDF_EPOCH` values. These functions should be used by applications using the `CDF_EPOCH` data type. Section 2.5.4 describes EPOCH values.

4.8.1 computeEPOCH

`computeEPOCH` calculates a `CDF_EPOCH` value given the individual components.

```
double computeEPOCH(     /* out -- CDF_EPOCH value returned. */
long year,               /* in  -- Year (AD, eg. 1991). */
long month,              /* in  -- Month (1-12). */
long day,                /* in  -- Day (1-31). */
long hour,               /* in  -- Hour (0-23). */
long minute,             /* in  -- Minute (0-59). */
long second,             /* in  -- Second (0-59). */
long msec);              /* in  -- Millisecond (0-999). */
```

`computeEPOCH` is a function in the CDF library.

4.8.2 EPOCHbreakdown

`EPOCHbreakdown` breaks down a `CDF_EPOCH` value into the individual components.

```
void EPOCHbreakdown(
double epoch,            /* in  -- The CDF_EPOCH value. */
long   *year,            /* out -- Year (AD, eg. 1991). */
long   *month,           /* out -- Month (1-12). */
long   *day,             /* out -- Day (1-31). */
long   *hour,            /* out -- Hour (0-23). */
long   *minute,          /* out -- Minute (0-59). */
long   *second,          /* out -- Second (0-59). */
long   *msec);           /* out -- Millisecond (0-999). */
```

`EPOCHbreakdown` is a function in the CDF library.

4.9 Custom C Functions

Most of the Standard interface functions callable from C applications are implemented as C macros which call `CDFlib` (Internal interface). For example, the `CDFcreate` function is actually defined as the following C macro...

```
#define CDFcreate(CDFname,numDims,dimSizes,encoding,majority,id) \
CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, id, \
        PUT_, CDF_ENCODING_, encoding, \
              CDF_MAJORITY_, majority, \
        NULL_)
```

These macros are defined in `cdf.h`. Where your application calls `CDFcreate`, the C compiler expands the macro into the corresponding call to `CDFlib`.

The flexibility of `CDFlib` allows you to define your own custom CDF functions using C macros. For instance, a function which inquires the format of a CDF could be defined as follows...

```
#define CDFinquireFormat(id,format) \
CDFlib (SELECT_, CDF_, id, \
        GET_, CDF_FORMAT_, format, \
        NULL_)
```

Your application would call the function as follows...

```
.
.
CDFid     id;             /* CDF identifier. */
CDFstatus status;         /* Returned status code. */
long      format;         /* Format of CDF. */
.
.
status = CDFinquireFormat (id, &format);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Chapter 5

CDF FORTRAN PROGRAMMING REFERENCE

5.1 Introduction

This chapter provides a reference for using the CDF library from FORTRAN applications. A separate reference exists for writing CDF applications in C (Chapter 4).

5.2 Compiling

Each program, subroutine, or function which calls the CDF library or references CDF parameters must include `cdf.inc`. The following line would be included at the top of each...

```
INCLUDE '<inc-path>cdf.inc'
```

where `<inc-path>` is the pathname of the directory containing `cdf.inc`. Check with your system manager to determine where on your system `cdf.inc` has been installed.

On VMS systems, `<inc-path>` may contain a logical name defined by yourself or your system manager (on UNIX systems, `<inc-path>` may NOT contain an environment variable).

On UNIX systems, a symbolic link to `cdf.inc` (using `ln -s`) could be created making `cdf.inc` appear to be in the same directory as the source file to be compiled. In that case, specifying `<inc-path>` would not be necessary.

5.2.1 VMS systems

The command to compile a source file on VMS systems would be as follows...

```
$ FORTRAN <source-file(s)>
```

where <source-file(s)> is the name of the source file(s) being compiled.

5.2.2 UNIX systems

The command to compile a source file on UNIX systems would be as follows...

```
% f77 -c <source-file(s)>.f
```

where <source-file(s)> is the name of the source file(s) being compiled.

The -c option specifies that only an object module(s) is to be produced (the link step is described in Section 4.3).

5.2.3 MS-DOS systems

FORTRAN is not yet supported on the IBM PC.

5.3 Linking

5.3.1 VMS systems

On VMS systems, applications using the Standard interface must be linked with the CDF object library (LIBCDF.OLB) and the Standard interface object module for FORTRAN (CDF_F_IF.OBJ). An example of this would be...

```
$ LINK <object-file(s)>, <lib-path>CDF_F_IF, <lib-path>LIBCDF/LIBRARY
```

where <object-file(s)> is your application's object module(s) and <lib-path> is the pathname of the directory containing LIBCDF.OLB and CDF_F_IF.OBJ. <lib-path> may contain a logical name defined by yourself or your system manager. <lib-path>CDF_F_IF must preceed <lib-path>LIBCDF/LIBRARY in the LINK command.

Applications using only the Internal interface should be linked with only the CDF object library (LIBCDF.OLB). An example of this would be...

```
$ LINK <object-file(s)>, <lib-path>LIBCDF/LIBRARY
```

where <object-file(s)> is your application's object module(s) and <lib-path> is the pathname of the directory containing LIBCDF.OLB. <lib-path> may contain a logical name defined by yourself or your system manager.

In both cases, the name of the executable created will be the name part of the (first) object file with `.EXE` appended.

Check with your system manager to determine where on your system `LIBCDF.OLB` and `CDF_F_IF.OBJ` have been installed.

NOTE: An application can use both interfaces when necessary. If this is the case, both the CDF library and the Standard interface object module for FORTRAN must be linked to the application's object modules. (The Internal interface is built into the CDF library).

5.3.2 UNIX systems

On UNIX systems, your applications must be linked with the CDF object library (`libcdf.a`). An example of this would be...

```
% f77 <object-file(s)>.o <lib-path>libcdf.a
```

where `<object-file(s)>.o` is your application's object module(s) and `<lib-path>` is the pathname of the directory containing `libcdf.a`. `<lib-path>` may contain an environment variable defined by yourself or your system manager.

The name of the executable created will be `a.out` by default. It may also be explicitly specified using the `-o` option.

Combining the compile and link

On UNIX systems the compile and link may be combined into one step as follows...

```
% f77 <source-file(s)>.f <lib-path>libcdf.a
```

Check with your system manager to determine where on your system `libcdf.a` has been installed.

NOTE: On UNIX systems, the Standard interface and Internal interface are both built into the CDF library.

5.3.3 MS-DOS systems

FORTRAN is not yet supported on the IBM PC.

5.4 Item Referencing

For FORTRAN applications, all items are referenced starting at one (1). These include variable, attribute, and attribute entry numbers, record numbers, dimensions, and dimension indices.

5.5 CDF Parameters — Predefined Constants and Types

The following sections describe constants and types defined for use by all CDF application programs written in FORTRAN. These constants and types are defined in `cdf.inc`. `cdf.inc` should be included in all application source files referencing CDF functions/parameters.

5.5.1 Status Code Constants

`CDF_OK`	A status code indicating the normal completion of a CDF function.
`CDF_WARN`	Threshold constant for testing severity of non-normal CDF status codes.

Section 5.7 describes how to use these constants to interpret status codes.

5.5.2 CDF formats

`SINGLE_FILE`	The CDF consists of only one file.
`MULTI_FILE`	The CDF consists of one header file for control and attribute data and one additional file for each variable in the CDF.

5.5.3 CDF Datatypes

One of the following constants must be used when specifying a CDF data type for an attribute entry or variable.

`CDF_BYTE`	1-byte, signed integer.
`CDF_CHAR`	1-byte, signed character.
`CDF_INT1`	1-byte, signed integer.
`CDF_UCHAR`	1-byte, unsigned character.
`CDF_UINT1`	1-byte, unsigned integer.
`CDF_INT2`	2-byte, signed integer.
`CDF_UINT2`	2-byte, unsigned integer.
`CDF_INT4`	4-byte, signed integer.
`CDF_UINT4`	4-byte, unsigned integer.
`CDF_REAL4`	4-byte, floating point.
`CDF_FLOAT`	4-byte, floating point.
`CDF_REAL8`	8-byte, floating point.

`CDF_DOUBLE`	8-byte, floating point.
`CDF_EPOCH`	8-byte, floating point.

`CDF_CHAR` and `CDF_UCHAR` are considered character string data types. These are significant because only variables of these data types may have more than one element per value (where each element is a character).

5.5.4 Data Encodings

A CDF's data encoding affects how its attribute entry and variable data values are stored (on disk). Attribute entry and variable values in memory are always in the host machine's native encoding.

`HOST_ENCODING`	Indicates host machine data representation (native). This encoding will provide the greatest performance but will not allow the CDF to be transported to other machines that have different encodings.
`NETWORK_ENCODING`	Indicates network transportable data representation (XDR). This encoding allows the CDF to be transported to other machines but with (possibly) reduced performance.
`VAX_ENCODING`	Indicates VAX data representation.
`SUN_ENCODING`	Indicates SUN data representation.
`MIPSEB_ENCODING`	Indicates MIPSEB data representation (Silicon Graphics Iris and Power Series).
`MIPSEL_ENCODING`	Indicates MIPSEL data representation (DECstation).
`IBMRS_ENCODING`	Indicates IBMRS data representation (IBM RS/6000 series).
`HP_ENCODING`	Indicates HP data representation (HP 9000 series).
`IBMPC_ENCODING`	Indicates IBM PC data representation.

When creating a CDF, you must specify `NETWORK_ENCODING`, `HOST_ENCODING`, or the specific encoding for the host machine (eg. `VAX_ENCODING`). Specifying the host machine's encoding explicitly has the same effect as specifying `HOST_ENCODING`.

When inquiring the encoding of a CDF, either `NETWORK_ENCODING` or the specific machine encoding will be returned (`HOST_ENCODING` is never returned).

5.5.5 Variable Majorities

A CDF's variable majority determines the order in which variable values (within the variable arrays) are stored in the CDF file(s).

`ROW_MAJOR`	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest.

`COL_MAJOR`	FORTRAN-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

For FORTRAN applications, the compiler defined majority for arrays is column major. The first dimension of multi-dimensional arrays varies the fastest in memory.

Knowing the variable majority of a CDF is necessary when performing hyper reads and writes. During a hyper read (using `CDF_var_hyper_get`), the CDF library will place the variable data values into the memory buffer in the same majority as that of the CDF. The buffer must then be processed according to that majority. Likewise, during a hyper write (using `CDF_var_hyper_put`), the CDF library will expect to find the variable data values in the memory buffer in the same majority as that of the CDF.

5.5.6 Record/Dimension Variances

Record and dimension variances affect how variable data values are physically stored.

`VARY`	True record or dimension variance.
`NOVARY`	False record or dimension variance.

If a variable has a record variance of `VARY`, then each record for that variable is physically stored. If the record variance is `NOVARY`, then only one record is physically stored (all virtual records contain the same values).

If a variable has a dimension variance of `VARY`, then each value along that dimension is physically stored. If the dimension variance is `NOVARY`, then only one value along that dimension is physically stored (all virtual values along that dimension are the same).

5.5.7 Attribute Scopes

Attribute scopes do not affect how the CDF library handles attribute entries. They are simply a way to explicitly declare the intended use of an attribute by user applications.

`GLOBAL_SCOPE`	Indicates that an attribute's scope is global (applies to the CDF as a whole).
`VARIABLE_SCOPE`	Indicates that an attribute's scope is by-variable (each entry corresponds to a variable).
`GLOBAL_SCOPE_ASSUMED`	Indicates that an attribute's scope is assumed global.
`VARIABLE_SCOPE_ASSUMED`	Indicates that an attribute's scope is assumed by-variable.

`GLOBAL_SCOPE` and `VARIABLE_SCOPE` are definite scopes. `GLOBAL_SCOPE_ASSUMED` and `VARIABLE_SCOPE_ASSUMED` are assumed scopes.

Section 2.8.3 explains attribute scopes and how assumed scopes can arise in a CDF.

5.5.8 Operational Limits

These are limits within the CDF library. If you reach one of these limits please contact CDF User Support.

`CDF_MAX_DIMS`	Maximum number of dimensions in a CDF.
`CDF_MAX_VARS`	Maximum number of variables in a CDF.
`CDF_MAX_CDFS`	Maximum number of CDFs that may be open at one time.

The CDF library imposes no limit on the number of attributes or attribute entries that a CDF may have.

5.5.9 Limits of Names and Other Character Strings

`CDF_PATHNAME_LEN`	Maximum length of a CDF pathname (excluding the `.cdf` or `.vnn` appended by the CDF library to construct filenames). A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems). NOTE: On UNIX systems pathnames are case sensitive.
`CDF_VAR_NAME_LEN`	Maximum length of a variable name.
`CDF_ATTR_NAME_LEN`	Maximum length of an attribute name.
`CDF_COPYRIGHT_LEN`	Maximum length of the copyright text returned by `CDF_doc`.
`CDF_STATUSTEXT_LEN`	Maximum length of the explanation text returned by `CDF_error`.

5.6 FORTRAN Function Reference

The following sections describe the CDF library routines callable from FORTRAN applications. Most routines return a status code of type `INTEGER*4` (see Section 5.7). Each routine except `CDF_lib` is part of the Standard interface — `CDF_lib` is part of the Internal interface. An application can use both interfaces when necessary.

5.6.1 CDF_create (Standard interface)

```
SUBROUTINE CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority,
1                      id, status)

CHARACTER CDF_name*(*)       ! in  -- CDF pathname.
INTEGER*4 num_dims           ! in  -- Number of dimensions.
INTEGER*4 dim_sizes(*)       ! in  -- Dimension sizes.
INTEGER*4 encoding           ! in  -- Data encoding.
INTEGER*4 majority           ! in  -- Variable majority.
```

```
INTEGER*4 id                  ! out -- CDF identifier.
INTEGER*4 status              ! out -- Completion status.
```

`CDF_create` creates a CDF as defined by the arguments. A CDF cannot be created if it already exists (the existing CDF will not be overwritten). If you want to recreate an existing CDF, you must first open it with `CDF_open`, delete it with `CDF_delete`, and then recreate it with `CDF_create`. If the existing CDF is corrupted, the call to `CDF_open` will fail (an error code will be returned). In this case you must delete the CDF at the command line. Delete the `.cdf` file and if the CDF has the multi-file format, delete all of the `.v0`,`.v1`,... files.

The arguments to `CDF_create` are defined as follows.

`CDF_name`	The pathname of the CDF to create (do not specify an extension). This may be at most `CDF_PATHNAME_LEN` characters. A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems). NOTE: On UNIX systems pathnames are case sensitive.
`num_dims`	Number of dimensions the CDF is to have. This may be as few as zero (0) and at most `CDF_MAX_DIMS`.
`dim_sizes`	The size of each dimension. `dim_sizes` is a 1-dimensional array containing `num_dims` elements. Each element of `dim_sizes` specifies the corresponding dimension size. Each size must be greater than zero (0). For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
`encoding`	The encoding for variable and attribute entry data. Specify one of the encodings described in Section 5.5.4.
`majority`	The majority for variable data. Specify one of the majorities described in Section 5.5.5.
`id`	The identifier for the created CDF. This identifier must be used in all subsequent operations on the CDF.
`status`	The completion status code. Section 5.7 explains how to interpret status codes.

When a CDF is created, both read and write access are allowed. The default format for a CDF created with `CDF_create` is multi-file (`MULTI_FILE`). The `CDF_lib` function (Internal interface) must be used to change a CDF's format to single-file (`SINGLE_FILE`).

`CDF_close` MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 5.6.5).

Example(s)

The following example will create a 3-dimensional CDF named `test1` with network encoding and row majority.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id              ! CDF identifier.
INTEGER*4 status          ! Returned status code.
INTEGER*4 num_dims        ! Number of dimensions.
INTEGER*4 dim_sizes(3)    ! Dimension sizes.
INTEGER*4 majority        ! Variable majority.

DATA num_dims/3/, dim_sizes/180,360,10/, majority/ROW_MAJOR/
.
.
CALL CDF_create ('test1', num_dims, dim_sizes, NETWORK_ENCODING,
1                majority, id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

ROW_MAJOR and NETWORK_ENCODING are defined in cdf.inc.

5.6.2 CDF_open (Standard interface)

```
SUBROUTINE CDF_open (CDF_name, id, status)

CHARACTER CDF_name*(*)        ! in  -- CDF pathname.
INTEGER*4 id                  ! out -- CDF identifier.
INTEGER*4 status              ! out -- Completion status.
```

CDF_open opens an existing CDF. The CDF is initially opened with only read access. This allows multiple applications to read the same CDF simultaneously. When an attempt to modify the CDF is made, it is automatically closed and reopened with read/write access (the function will fail if the application does not have or can not get write access to the CDF).

The arguments to CDF_open are defined as follows.

CDF_name
: The pathname of the CDF to open (do not specify an extension). This may be at most CDF_PATHNAME_LEN characters. A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

  NOTE: On UNIX systems pathnames are case sensitive.

id
: The identifier for the opened CDF. This identifier must be used in all subsequent operations on the CDF.

status
: The completion status code. Section 5.7 explains how to interpret status codes.

CDF_close MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 5.6.5).

Example(s)

The following example will open a CDF named NOAA1.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
CHARACTER CDF_name*(CDF_PATHNAME_LEN) ! Name of CDF.

DATA CDF_name/'NOAA1'/
.
.
CALL CDF_open (CDF_name, id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.3 CDF_doc (Standard interface)

```
SUBROUTINE CDF_doc (id, version, release, copyright, status)

INTEGER*4 id,                         ! in  -- CDF identifier.
INTEGER*4 version,                    ! out -- Version number.
INTEGER*4 release,                    ! out -- Release number.
CHARACTER copyright*(CDF_DOCUMENT_LEN)  ! out -- Copyright.
```

CDF_doc is used to inquire general documentation about a CDF. The version/release of the CDF library which created the CDF is provided (eg. CDF V2.1 is version 2, release 1) along with the CDF copyright notice.

The arguments to CDF_doc are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
version	The version number of the CDF library which created the CDF.
release	The release number of the CDF library which created the CDF.
copyright	The copyright notice of the CDF library which created the CDF. This character string must be large enough to hold CDF_COPYRIGHT_LEN characters. This string

will contain a linefeed character after each line of the copyright notice (it can be printed without modification but may contain many trailing blanks).

`status` The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will inquire and display the version/release and copyright notice.

```
     .
     .
     INCLUDE '<path>cdf.inc'
     .
     .
     INTEGER*4 id                            ! CDF identifier.
     INTEGER*4 status                        ! Returned status code.
     INTEGER*4 version                       ! CDF version number.
     INTEGER*4 release                       ! CDF release number.
     CHARACTER copyright*(CDF_COPYRIGHT_LEN)  ! Copyright notice.
     .
     .
     CALL CDF_doc (id, version, release, copyright, status)
     IF (status .LT. CDF_OK) THEN               ! INFO status codes ignored
       CALL UserStatusHandler (status)
     ELSE
       WRITE (6,101) version, release
 101   FORMAT (' ','Version: ',I3,' Release:',I3)
       WRITE (6,102) copyright
 102   FORMAT (' ','Copyright:',/,A)
     END IF
     .
     .
```

5.6.4 CDF_inquire (Standard interface)

```
SUBROUTINE CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
1                       max_rec, num_vars, num_attrs, status)

INTEGER*4 id                        ! in  -- CDF identifier.
INTEGER*4 num_dims                  ! out -- Number of dimensions.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)   ! out -- Dimension sizes.
INTEGER*4 encoding                  ! out -- Data encoding.
INTEGER*4 majority                  ! out -- Variable majority.
INTEGER*4 max_rec                   ! out -- Maximum record number in the CDF.
INTEGER*4 num_vars                  ! out -- Number of variables in the CDF.
INTEGER*4 num_attrs                 ! out -- Number of attributes in the CDF.
INTEGER*4 status                    ! out -- Completion status.
```

CDF_inquire inquires the basic characteristics of a CDF. An application needs to know the number of dimensions and their sizes before it can access variable data. Knowing the variable majority can be used to optimize performance and is necessary to properly access the buffer of data values used with the hyper functions — CDF_var_hyper_put and CDF_var_hyper_get.

The arguments to CDF_inquire are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
num_dims	The number of dimensions in the CDF.
dim_sizes	The dimension sizes. dim_sizes is a 1-dimensional array containing one element per CDF dimension. Each element of dim_sizes receives the corresponding dimension size. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
encoding	The encoding of the variable and attribute entry data. The encodings are defined in Section 5.5.4.
majority	The majority of the variable data. The majorities are defined in Section 5.5.5.
max_rec	The maximum record number written to the CDF. Note that the maximum record number written is kept separately for each variable in the CDF. max_rec is the largest of these (the conceptual maximum record written). Some variables may have fewer records actually written. CDF_lib (Internal interface) may be used to inquire the maximum record written for an individual variable (see Section 5.6.24).
num_vars	The number of variables in the CDF.
num_attrs	The number of attributes in the CDF.
status	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will inquire the basic information about a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 num_dims                  ! Number of dimensions.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)   ! Dimension sizes (allocate to allow the
                                    ! maximum number of dimensions).
INTEGER*4 encoding                  ! Data encoding.
INTEGER*4 majority                  ! Variable majority.
INTEGER*4 max_rec                   ! Maximum record number in CDF.
```

```
INTEGER*4 num_vars                ! Number of variables in CDF.
INTEGER*4 num_attrs               ! Number of attributes in CDF.
.
.
CALL CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
.                 max_rec, num_vars, num_attrs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.5 CDF_close (Standard interface)

```
SUBROUTINE CDF_close (id, status)

INTEGER*4 id              ! in  -- CDF identifier.
INTEGER*4 status          ! out -- Completion status.
```

CDF_close closes the specified CDF. The CDF's data buffers are flushed, all of the CDF's open files are closed, and the CDF identifier is made available for reuse.

You MUST close a CDF with CDF_close to guarantee that all modifications you have made will actually be written to the CDF's file(s). If your program exits, normally or otherwise, without a successful call to CDF_close, the CDF's data buffers are left unflushed. If you have only read from a CDF, and not written to it, failure to close it is of no consequence. Of course, it is always good practice to close your CDFs before exiting your application even if they are only read.

The arguments to CDF_close are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
status	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will close an open CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                      ! CDF identifier.
INTEGER*4 status                  ! Returned status code.
.
.
CALL CDF_close (id, status)
```

```
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.6 CDF_delete (Standard interface)

```
SUBROUTINE CDFdelete (id, status)

INTEGER*4 id              ! in  -- CDF identifier.
INTEGER*4 status          ! out -- Completion status.
```

`CDF_delete` deletes the specified CDF. The CDF files deleted include the `.cdf` file and if the CDF consists of multiple files, the `.v0`,`.v1`,... files.

You must open a CDF before you are allowed to deleted it. If you have no privilege to delete the CDF file(s), they will not be deleted. If the CDF is corrupted and cannot be opened, the CDF file(s) must be deleted at the command line.

The arguments to `CDF_delete` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`status`	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will open and then delete an existing CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
.
.
CALL CDF_open ('test2', id, status)
IF (status .LT. CDF_OK) THEN          ! INFO status codes ignored.
  CALL UserStatusHandler (status)
ELSE
  CALL CDF_delete (id, status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END IF
.
.
```

5.6.7 CDF_error (Standard interface)

```
SUBROUTINE CDF_error (status, message)

INTEGER*4 status                        ! in  -- Status code.
CHARACTER message*(CDF_STATUSTEXT_LEN)  ! out -- Explanation text for
                                                 the status code.
```

CDF_error is used to inquire the explanation of a given status code (not just error codes). Section 5.7 explains how to interpret status codes and Appendix B lists all of the possible status codes.

The arguments to CDF_error are defined as follows.

status
: The status code to check.

message
: The explanation of the status code. This character string must be large enough to hold CDF_STATUSTEXT_LEN characters. The CDF library will pad this character string with trailing blanks if necessary.

Example(s)

The following example displays the explanation text if an error code is returned from a call to CDF_open.

```
      .
      .
      INCLUDE '<path>cdf.inc'
      .
      .
      INTEGER*4 id                          ! CDF identifier.
      INTEGER*4 status                      ! Returned status code.
      CHARACTER text*(CDF_STATUSTEXT_LEN)   ! Explanation text.
      .
      .
      CALL CDF_open ('giss_wetl', id, status)
      IF (status .LT. CDF_WARN) THEN          ! INFO and WARNING codes ignored.
        CALL CDF_error (status, text)
        WRITE (6,101) text
 101    FORMAT (' ','ERROR> ',A)
      END IF
      .
      .
```

5.6.8 CDF_attr_create (Standard interface)

```
SUBROUTINE CDF_attr_create (id, attr_name, attr_scope, attr_num, status)
INTEGER*4 id                 ! in  -- CDF identifier.
```

```
CHARACTER attr_name*(*)       ! in  -- Attribute name.
INTEGER*4 attr_scope          ! in  -- Scope of attribute.
INTEGER*4 attr_num            ! out -- Attribute number.
INTEGER*4 status              ! out -- Completion status.
```

`CDF_attr_create` creates an attribute in the specified CDF. An attribute with the same name must not already exist in the CDF.

The arguments to `CDF_attr_create` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`attr_name`	The name of the attribute to create. This may be at most `CDF_ATTR_NAME_LEN` characters. Attribute names are case-sensitive.
`attr_scope`	The scope of the new attribute. Specify one of the scopes described in Section 5.5.7.
`attr_num`	The number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may be determined with the `CDF_attr_num` function.
`status`	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example creates two attributes. The `TITLE` attribute is created with global scope — it applies to the entire CDF (most likely the title of the data set stored in the CDF). The `Units` attribute is created with variable scope — each entry describes some property of the corresponding variable (in this case the units for the data).

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                         ! CDF identifier.
INTEGER*4 status                     ! Returned status code.
CHARACTER UNITS_attr_name*5          ! Name of "Units" attribute.
INTEGER*4 UNITS_attr_num             ! "Units" attribute number.
INTEGER*4 TITLE_attr_num             ! "TITLE" attribute number.
INTEGER*4 TITLE_attr_scope           ! "TITLE" attribute scope.

DATA UNITS_attr_name/'Units'/, TITLE_attr_scope/GLOBAL_SCOPE/
.
.
CALL CDF_attr_create (id, 'TITLE', TITLE_attr_scope, TITLE_attr_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
```

```
CALL CDF_attr_create (id, UNITS_attr_name, VARIABLE_SCOPE, UNITS_attr_num,
1                     status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.9 CDF_attr_num (Standard interface)

```
INTEGER*4 FUNCTION CDF_attr_num (id, attr_name)

INTEGER*4 id                ! in --  CDF id.
CHARACTER attr_name*(*)     ! in -- attribute name.
```

CDF_attr_num is used to determine the attribute number associated with a given attribute name. If the attribute is found, CDF_attr_num returns its number — which will be equal to or greater than one (1). If an error occurs (eg. the attribute name does not exist in the CDF), an error code is returned. Error codes are less than zero (0).

The arguments to CDF_attr_num are defined as follows.

id — The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.

attr_name — The name of the attribute for which to search. This may be at most CDF_ATTR_NAME_LEN characters. Attribute names are case-sensitive.

CDF_attr_num may be used as an embedded function call when an attribute number is needed. CDF_attr_num is declared in cdf.inc (FORTRAN functions must be declared so that the returned value is interpreted correctly).

Example(s)

In the following example the attribute named pressure will be renamed PRESSURE with CDF_attr_num being used as an embedded function call.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                       ! CDF identifier.
INTEGER*4 status                   ! Returned status code.
.
.
CALL CDF_attr_rename (id, CDF_attr_num(id,'pressure'), 'PRESSURE', status)
```

```
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Note that if the attribute pressure did not exist in the CDF, the call to CDF_attr_num would have returned an error code. Passing that error code to CDF_attr_rename as an attribute number would have resulted in CDF_attr_rename also returning an error code. CDF_attr_rename is described in Section 5.6.10.

5.6.10 CDF_attr_rename (Standard interface)

```
SUBROUTINE CDF_attr_rename (id, attr_num, attr_name, status)

INTEGER*4 id             ! in  -- CDF identifier.
INTEGER*4 attr_num       ! in  -- Attribute number.
CHARACTER attr_name*(*)  ! in  -- New attribute name.
INTEGER*4 status         ! out -- Completion status.
```

CDF_attr_rename is used to rename an existing attribute. An attribute with the new name must not already exist in the CDF.

The arguments to CDF_attr_rename are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
attr_num	The number of the attribute to rename. This number may be determined with a call to CDF_attr_num (see Section 5.6.9).
attr_name	The new attribute name. This may be at most CDF_ATTR_NAME_LEN characters. Attribute names are case-sensitive.
status	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

In the following example the attribute named LAT is renamed LATITUDE.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
.
.
CALL CDF_attr_rename (id, CDF_attr_num(id,'LAT'), 'LATITUDE', status)
```

```
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.11 CDF_attr_inquire (Standard interface)

```
SUBROUTINE CDF_attr_inquire (id, attr_num, attr_name, attr_scope, max_entry,
1                             status)
INTEGER*4 id                              ! in  -- CDF identifier.
INTEGER*4 attr_num                        ! in  -- Attribute number.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN)   ! out -- Attribute name.
INTEGER*4 attr_scope                      ! out -- Attribute scope.
INTEGER*4 max_entry                       ! out -- Maximum entry number for
                                          !        attribute.
INTEGER*4 status                          ! out -- Completion status.
```

`CDF_attr_inquire` is used to inquire about the specified attribute. To inquire about a specific attribute entry, use `CDF_attr_entry_inquire` (Section 5.6.12).

The arguments to `CDF_attr_inquire` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`attr_num`	The number of the attribute to inquire. This number may be determined with a call to `CDF_attr_num` (see Section 5.6.9).
`attr_name`	The attribute's name. This character string must be large enough to hold `CDF_ATTR_NAME_LEN` characters.
`attr_scope`	The scope of the attribute. Attribute scopes are defined in Section 5.5.7.
`max_entry`	The maximum entry number for this attribute. This may not correspond with the number of entries (if some entry numbers were not used). The number of entries actually used may be inquired with the `CDF_lib` function (Internal interface, see Section 5.6.24).
`status`	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example displays the name of each attribute in a CDF. The number of attributes in the CDF is first determined using `CDF_inquire`. Note that attribute numbers start at one (1) and are consecutive.

```
.
.
INCLUDE '<path>cdf.inc'
.
```

```
   .
   INTEGER*4 id                          ! CDF identifier.
   INTEGER*4 status                      ! Returned status code.
   INTEGER*4 num_dims                    ! Number of dimensions.
   INTEGER*4 dim_sizes(CDF_MAX_DIMS)     ! Dimension sizes (allocate to
                                         ! allow the maximum number of
                                         ! dimensions).
   INTEGER*4 encoding                    ! Data encoding.
   INTEGER*4 majority                    ! Variable majority.
   INTEGER*4 max_rec                     ! Maximum record number in CDF.
   INTEGER*4 num_vars                    ! Number of variables in CDF.
   INTEGER*4 num_attrs                   ! Number of attributes in CDF.
   INTEGER*4 attr_n                      ! Attribute number.
   CHARACTER attr_name*(CDF_ATTR_NAME_LEN)  ! Attribute name.
   INTEGER*4 attr_scope                  ! Attribute scope.
   INTEGER*4 max_entry                   ! Maximum entry number.
   .
   .
   CALL CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
   1                 max_rec, num_vars, num_attrs, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

   DO attr_n = 1, num_attrs
     CALL CDF_attr_inquire (id, attr_n, attr_name, attr_scope, max_entry,
     1                      status)
     IF (status .LT. CDF_OK) THEN       ! INFO status codes ignored.
       CALL UserStatusHandler (status)
     ELSE
       WRITE (6,10) attr_name
10     FORMAT (' ',A)
     END IF
   END DO
   .
   .
```

5.6.12 CDF_attr_entry_inquire (Standard interface)

```
SUBROUTINE CDF_attr_entry_inquire (id, attr_num, entry_num, data_type,
1                                  num_elements, status)
INTEGER*4 id              ! in  -- CDF identifier.
INTEGER*4 attr_num        ! in  -- Attribute number.
INTEGER*4 entry_num       ! in  -- Entry number.
INTEGER*4 data_type       ! out -- Entry data type.
INTEGER*4 num_elements    ! out -- Number of elements (of the data type).
INTEGER*4 status          ! out -- Completion status.
```

`CDF_attr_entry_inquire` is used to inquire about a specific attribute entry. To inquire about the attribute in general, use `CDF_attr_inquire` (Section 5.6.11). `CDF_attr_entry_inquire` would normally be called before calling `CDF_attr_get` in order to determine the data type and number of elements (of that data type) for an entry. This would be necessary to correctly allocate enough memory to receive the value read by CDF_attr_get.

The arguments to CDF_attr_entry_inquire are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
attr_num	The attribute number in which to inquire an entry. This number may be determined with a call to CDF_attr_num (see Section 5.6.9).
entry_num	The entry number to inquire. If the attribute is global in scope, this is simply the entry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated variable (the variable being described in some way by this entry).
data_type	The data type of the specified entry. The data types are defined in Section 5.5.3.
num_elements	The number of elements of the data type. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. For all other data types, this is the number of elements in an array of that data type.
status	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example inquires each entry for an attribute. Note that entry numbers need not be consecutive — not every entry number between one (1) and the maximum entry number must exist. For this reason, NO_SUCH_ENTRY is an expected error code. Note also that if the attribute is variable in scope, then the entry numbers are actually variable numbers.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                           ! CDF identifier.
INTEGER*4 status                       ! Returned status code.
INTEGER*4 attr_n                       ! Attribute number.
INTEGER*4 entryN                       ! Entry number.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN)  ! Attribute name.
INTEGER*4 attr_scope                   ! Attribute scope.
INTEGER*4 max_entry                    ! Maximum entry number.
INTEGER*4 data_type                    ! Entry data type.
INTEGER*4 num_elems                    ! Number of elements (of the
                                       ! data type).
.
.
attr_n = CDF_attr_num (id, 'TMP')
```

```
   IF (attr_n .LT. 1) CALL UserStatusHandler (attr_n)  ! If less than one (1),
                                                       ! then it must be a
                                                       ! warning/error code.

   CALL CDF_attr_inquire (id, attr_n, attr_name, attr_scope, max_entry, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

   DO entryN = 1, max_entry
     CALL CDF_attr_entry_inquire (id, attr_n, entryN, data_type, num_elems,
     1                            status)
     IF (status .LT. CDF_OK) THEN
       IF (status .NE. NO_SUCH_ENTRY) CALL UserStatusHandler (status)
     ELSE
C      (process entries)
       .
       .
     END IF
   END DO
```

5.6.13 CDF_attr_put (Standard interface)

```
SUBROUTINE CDF_attr_put (id, attr_num, entry_num, data_type, num_elements,
1                        value, status)

INTEGER*4 id,              ! in --  CDF identifier.
INTEGER*4 attr_num         ! in --  Attribute number.
INTEGER*4 entry_num        ! in --  Entry number.
INTEGER*4 data_type        ! in --  Data type of this attribute entry.
INTEGER*4 num_elements     ! in --  Number of elements (of the data type).
<type>    value            ! in --  Value (<type> depends on the data type
                           !        of the entry).
INTEGER*4 status           ! out -- Completion status.
```

CDF_attr_put is used to write an attribute entry to a CDF. The entry may or may not already exist. If it does exist, it is overwritten. The data type and number of elements (of that data type) may be changed when overwriting an existing entry.

The arguments to CDF_attr_put are defined as follows.

id — The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.

attr_num — The attribute number. This number may be determined with a call to CDF_attr_num (see Section 5.6.9).

entry_num — The entry number. If the attribute is global in scope, this is simply the entry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated variable (the variable being described in some way by this entry).

`data_type`	The data type of the specified entry. Specify one of the data types defined in Section 5.5.3.
`num_elements`	The number of elements of the data type. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string. For all other data types, this is the number of elements in an array of that data type.
`value`	The value(s) to write. The entry value is written to the CDF from memory addres `value`.
`status`	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example writes two attribute entries. The first is to entry number one (1) of the global scope attribute `TITLE`. The second is to the variable scope attribute `VALIDs` for the entry which corresponds to variable `TMP`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
PARAMETER TITLE_LEN = 10          ! Length of CDF title.
.
.
INTEGER*4 id                      ! CDF identifier.
INTEGER*4 status                  ! Returned status code.
INTEGER*4 entry_num               ! Entry number.
INTEGER*4 num_elements            ! Number of elements (of data type).
CHARACTER title*(TITLE_LEN)       ! Value of TITLE attribute, entry 1.
INTEGER*2 TMPvalids(2)            ! Value(s) of VALIDs attribute,
                                  ! entry for TMP variable.

DATA title/'CDF title.'/, TMPvalids/15,30/
.
.
entry_num = 1
CALL CDF_attr_put (id, CDF_attr_num(id,'TITLE'), entry_num, CDF_CHAR,
1                  TITLE_LEN, title, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
num_elements = 2
CALL CDF_attr_put (id, CDF_attr_num(id,'VALIDs'), CDF_var_numm(id,'TMP'),
1                  CDF_INT2, num_elements, TMPvalids, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.14 CDF_attr_get (Standard interface)

```
SUBROUTINE CDF_attr_get (id, attr_num, entry_num, value, status)

INTEGER*4 id              ! in  -- CDF identifier.
INTEGER*4 attr_num        ! in  -- Attribute number.
INTEGER*4 entry_num       ! in  -- Entry number.
<type>    value           ! out -- Value (<type> is dependent on the data
                          !        type of the entry).
INTEGER*4 status          ! out -- Completion status.
```

`CDF_attr_get` is used to read an attribute entry from a CDF. In most cases it will be necessary to call `CDF_attr_entry_inquire` before calling `CDF_attr_get` in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to `CDF_attr_get` are defined as follows.

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`attr_num`	The attribute number. This number may be determined with a call to `CDF_attr_num` (see Section 5.6.9).
`entry_num`	The entry number. If the attribute is global in scope, this is simply the entry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated variable (the variable being described in some way by this entry).
`value`	The value read. This buffer must be large enough to hold the entry value. `CDF_attr_entry_inquire` would be used to determine the entry data type and number of elements (of that data type). The entry value is read from the CDF and placed into memory at address `value`.

Example(s)

The following example displays the value of the `UNITS` attribute for the entry corresponding to the `PRES_LVL` variable (but only if the data type is `CDF_CHAR`).

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                    ! CDF identifier.
INTEGER*4 status                ! Returned status code.
INTEGER*4 attr_n                ! Attribute number.
INTEGER*4 entryN                ! Entry number.
INTEGER*4 data_type             ! Data type.
INTEGER*4 num_elems             ! Number of elements (of data type).
```

```
   CHARACTER buffer*100              ! Buffer to receive entry value (in
                                     ! this case it is assumed that 100
                                     ! characters is enough).
   .
   .
   attr_n = CDF_attr_Num (id, 'UNITS')
   IF (attr_n .LT. 0) CALL UserStatusHandler (attr_n)  ! If less than one (1),
                                                       ! then it must be a
                                                       ! warning/error code.

   entryN = CDF_var_num (id, 'PRES_LVL')              ! Variable number is
                                                      ! entry number.
   IF (entryN .LT. 0) CALL UserStatusHandler (status) ! If less than one (1),
                                                      ! then it must be a
                                                      ! warning/error code.

   CALL CDF_attr_entry_inquire (id, attr_n, entryN, data_type, num_elems,
   1                            status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

   IF (data_type .EQ. CDF_CHAR) THEN
     CALL CDF_attr_get (id, attr_n, entryN, buffer, status)
     IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
     WRITE (6,10) buffer(1:num_elems)
10   FORMAT (' ',A)
   END IF
   .
   .
```

5.6.15 CDF_attr_correct_scope (Standard interface)

```
SUBROUTINE CDF_attr_correct_scope (id, attr_num, attr_scope, status)

INTEGER*4 id                  ! in  -- CDF identifier.
INTEGER*4 attr_num            ! in  -- Attribute number.
INTEGER*4 attr_scope          ! in  -- New scope of attribute.
INTEGER*4 status              ! out -- Completion status.
```

CDF_attr_correct_scope is used to correct the scope of an attribute. Section 2.8.3 describes attribute scopes and the need for this function.

The arguments to CDF_attr_correct_scope are defined as follows.

id
: The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.

attr_num
: The attribute number. This number may be determined with a call to CDF_attr_num (see Section 5.6.9).

attr_scope The new scope for the attribute. Specify one of the definite scopes defined in Section 5.5.7.

status The completion status code. Section 5.7 explains how to interpret status codes.

IMPORTANT NOTE

This function is not a permanent one — it may be deleted from a future release of the CDF library. Version 1 CDFs should be converted to Version 2 CDFs as soon as possible. Any assumed attribute scopes should then be corrected by using this function or the CDFscope toolkit program.

Example(s)

The following example will correct the scope of every attribute in a CDF. Assumed scopes will be corrected to the corresponding definite scope. Note that it may not always be the case that an assumed scope should be corrected to the corresponding definite scope (see Section 2.8.3). The toolkit program CDFscope could also be used to correct attribute scopes.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 num_dims                  ! Number of dimensions.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)   ! Dimension sizes (allocate to allow
                                    ! the maximum number of dimensions).
INTEGER*4 encoding                  ! Data encoding.
INTEGER*4 majority                  ! Variable majority.
INTEGER*4 max_rec                   ! Maximum record number in CDF.
INTEGER*4 num_vars                  ! Number of variables in CDF.
INTEGER*4 num_attrs                 ! Number of attributes in CDF.
INTEGER*4 attr_n                    ! Attribute number.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN) ! Attribute name.
INTEGER*4 attr_scope                ! Attribute scope.
INTEGER*4 max_entry                 ! Maximum entry number.
.
.
CALL CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
1                 max_rec, num_vars, num_attrs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

DO attr_n = 1, num_attrs
  CALL CDF_attr_inquire (id, attr_n, attr_name, attr_scope, max_entry,
  1                      status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

  IF (attr_scope .EQ. GLOBAL_SCOPE_ASSUMED) THEN
```

```
      CALL CDF_attr_correct_scope (id, attr_n, GLOBAL_SCOPE, status)
      IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
    ELSE
      IF (attr_scope .EQ. VARIABLE_SCOPE_ASSUMED) THEN
        CALL CDF_attr_correct_scope (id, attr_n, VARIABLE_SCOPE, status)
        IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
      END IF
    END IF
  END DO
  .
  .
```

5.6.16 CDF_var_create (Standard interface)

```
SUBROUTINE CDF_var_create (id, var_name, data_type, num_elements,
1                           rec_variance, dim_variances, var_num, status)

INTEGER*4 id                  ! in --  CDF identifier.
CHARACTER var_name*(*)        ! in --  Variable name.
INTEGER*4 data_type           ! in --  Data type.
INTEGER*4 num_elements        ! in --  Number of elements (of the data type).
INTEGER*4 rec_variance        ! in --  Record variance.
INTEGER*4 dim_variances(*)    ! in --  Dimension variances.
INTEGER*4 var_num             ! out -- Variable number.
INTEGER*4 status              ! out -- Completion status.
```

`CDF_var_create` is used to create a new variable in a CDF. A variable with the same name must not already exist in the CDF.

The arguments to `CDF_var_create` are defined as follows.

`id` — The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.

`var_name` — The name of the variable to create. This may be at most `CDF_VAR_NAME_LEN` characters. Variable names are case-sensitive.

`data_type` — The data type of the new variable. Specify one of the data types defined in Section 5.5.3.

`num_elements` — The number of elements of the data type at each variable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (each variable value consists of the entire string). For all other data types, this must always be one (1) — multiple elements at each variable value are not allowed for non-character string data types.

`rec_variance` — The variable's record variance. Specify one of the variances defined in Section 5.5.6.

`dim_variances` — The variable's dimension variances. `dim_variances` is a 1-dimensional array containing one element per CDF dimension. Each element of `dim_variances` specifies the corresponding dimension variance. For each dimension, specify one of the variances defined in Section 5.5.6. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`var_num` The number assigned to the new variable. This number must be used in subsequent CDF function calls when referring to this variable. An existing variable's number may be determined with the `CDF_var_num` function.

`status` The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will create several variables in a 2-dimensional CDF. In this case, `EPOCH`, `LAT`, and `LON` are independent variables and `TMP` is a dependent variable.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 EPOCH_rec_vary            ! EPOCH record variance.
INTEGER*4 LAT_rec_vary              ! LAT record variance.
INTEGER*4 LON_rec_vary              ! LON record variance.
INTEGER*4 TMP_rec_vary              ! TMP record variance.
INTEGER*4 EPOCH_dim_varys(2)        ! EPOCH dimension variances.
INTEGER*4 LAT_dim_varys(2)          ! LAT dimension variances.
INTEGER*4 LON_dim_varys(2)          ! LON dimension variances.
INTEGER*4 TMP_dim_varys(2)          ! TMP dimension variances.
INTEGER*4 EPOCH_var_num             ! EPOCH variable number.
INTEGER*4 LAT_var_num               ! LAT variable number.
INTEGER*4 LON_var_num               ! LON variable number.
INTEGER*4 TMP_var_num               ! TMP variable number.

DATA EPOCH_rec_vary/VARY/, LAT_rec_vary/NOVARY/,
1    LON_rec_vary/NOVARY/, TMP_rec_vary/VARY/
DATA EPOCH_dim_varys/NOVARY,NOVARY/, LAT_dim_varys/NOVARY,VARY/,
1    LON_dim_varys/VARY,NOVARY/, TMP_dim_varys/VARY,VARY/
.
.
CALL CDF_var_create (id, 'EPOCH', CDF_EPOCH, 1,
1                    EPOCH_rec_vary, EPOCH_dim_varys, EPOCH_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_var_create (id, 'LATITUDE', CDF_INT2, 1,
1                    LAT_rec_vary, LAT_dim_varys, LAT_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_var_create (id, 'LONGITUDE', CDF_INT2, 1,
```

```
1                    LON_rec_vary, LON_dim_varys, LON_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_var_create (id, 'TEMPERATURE', CDF_REAL4, 1,
1                    TMP_rec_vary, TMP_dim_varys, TMP_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.17 CDF_var_num (Standard interface)

```
INTEGER*4 FUNCTION CDF_var_num (id, var_name)

INTEGER*4 id                ! in -- CDF identifier.
CHARACTER var_name*(*)      ! in -- Variable name.
```

`CDF_var_num` is used to determine the variable number associated with a given variable name. If the variable is found, `CDF_var_num` returns its number — which will be equal to or greater than one (1). If an error occurs (eg. the variable name does not exist in the CDF), an error code is returned. Error codes are less than zero (0).

The arguments to `CDF_var_num` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.

`var_name` The name of the variable for which to search. This may be at most `CDF_VAR_NAME_LEN` characters. Variable names are case-sensitive.

`CDF_var_num` may be used as an embedded function call when a variable number is needed. `CDF_var_num` is declared in `cdf.inc` (FORTRAN functions must be declared so that the returned value is interpreted correctly).

Example(s)

In the following example `CDF_var_num` is used as an embedded function call when inquiring about a variable.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                         ! CDF identifier.
INTEGER*4 status                     ! Returned status code.
CHARACTER var_name*(CDF_VAR_NAME_LEN)  ! Variable name.
INTEGER*4 data_type                  ! Data type of the variable.
```

```
INTEGER*4 num_elements                ! Number of elements (of the
                                      ! data type).
INTEGER*4 rec_variances               ! Record variance.
INTEGER*4 dim_variances(CDF_MAX_DIMS) ! Dimension variances.
.
.
CALL CDF_var_inquire (id, CDF_var_num(id,'LATITUDE'), var_name, data_type,
1                     num_elements, rec_variance, dim_variances, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

In this example the variable named LATITUDE was inquired. Note that if the variable LATITUDE did not exist in the CDF, the call to CDF_var_num would have returned an error code. Passing that error code to CDF_var_inquire as a variable number would have resulted in CDF_var_inquire also returning an error code. Also note that the name written into var_name is already known (LATITUDE). In some cases the variable names will be unknown — CDF_var_inquire would be used to determine them. CDF_var_inquire is described in Section 5.6.19.

5.6.18 CDF_var_rename (Standard interface)

```
SUBROUTINE CDF_var_rename (id, var_num, var_name, status)

INTEGER*4 id              ! in  -- CDF identifier.
INTEGER*4 var_num         ! in  -- Variable number.
CHARACTER var_name*(*)    ! in  -- New variable name.
INTEGER*4 status          ! out -- Completion status.
```

CDF_var_rename is used to rename an existing variable. A variable with the new name must not already exist in the CDF.

The arguments to CDF_var_rename are defined as follows.

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
var_num	The number of the variable to rename. This number may be determined with a call to CDF_var_num (see Section 5.6.17).
var_name	The new variable name. This may be at most CDF_VAR_NAME_LEN characters. Variables names are case-sensitive.
status	The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

In the following example the variable named TEMPERATURE is renamed TMP (if it exists). Note that if CDF_var_num returns a value less than one (1), that value is not a variable number but rather a warning/error code.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 var_num                   ! Variable number.
.
.
var_num = CDF_var_num (id, 'TEMPERATURE')
IF (var_num .LT. 1) THEN
  IF (var_num .NE. NO_SUCH_VAR) CALL UserStatusHandler (var_num)
ELSE
  CALL CDF_var_rename (id, var_num, 'TMP', status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END IF
.
.
```

5.6.19 CDF_var_inquire (Standard interface)

```
SUBROUTINE CDF_var_inquire (id, var_num, var_name, data_type, num_elements,
1                           rec_variance, dim_variances, status)

INTEGER*4 id                          ! in  -- CDF identifier.
INTEGER*4 var_num                     ! in  -- Variable number.
CHARACTER var_name*(CDF_VAR_NAME_LEN) ! out -- Variable name.
INTEGER*4 data_type                   ! out -- Data type.
INTEGER*4 num_elements                ! out -- Number of elements (of the
                                      !        data type).
INTEGER*4 rec_variance                ! out -- Record variance.
INTEGER*4 dim_variances(CDF_MAX_DIMS) ! out -- Dimension variances.
INTEGER*4 status                      ! out -- Completion status.
```

`CDF_var_inquire` is used to inquire about the specified variable. This function would normally be used before reading variable values (with `CDF_var_get` or `CDF_var_hyper_get`) to determine the data type and number of elements (of that data type).

The arguments to `CDF_var_inquire` are defined as follows.

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
- **var_num**: The number of the variable to inquire. This number may be determined with a call to `CDF_var_num` (see Section 5.6.17).

var_name The variable's name. This character string must be large enough to hold CDF_VAR_NAME_LEN characters.

data_type The data type of the variable. The data types are defined in Section 5.5.3.

num_elements The number of elements of the data type at each variable value. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string (each variable value consists of the entire string). For all other data types, this will always be one (1) — multiple elements at each variable value are not allowed for non-character string data types.

rec_variance The record variance. The record variances are defined in Section 5.5.6.

dim_variances The dimension variances. dim_variances is a 1-dimensional array containing one element per CDF dimension. Each element of dim_variances receives the corresponding dimension variance. The dimension variances are defined in Section 5.5.6. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

status The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example inquires about a variable named HEAT_FLUX in a CDF. Note that the variable name returned by CDF_var_inquire will be the same as that passed in to CDF_var_num.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
CHARACTER var_name*(CDF_VAR_NAME_LEN) ! Variable name.
INTEGER*4 data_type                   ! Data type.
INTEGER*4 num_elems                   ! Number of elements (of data type).
INTEGER*4 rec_vary                    ! Record variance.
INTEGER*4 dim_varys(CDF_MAX_DIMS)     ! Dimension variances (allocate
                                      ! to allow the maximum number of
                                      ! dimensions).
.
.
CALL CDF_var_inquire (id, CDF_var_num(id,'HEAT_FLUX'), var_name, data_type,
1                     num_elems, rec_vary, dim_varys, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6.20 CDF_var_put (Standard interface)

```
SUBROUTINE CDF_var_put (id, var_num, rec_num, indices, value, status)

INTEGER*4 id            ! in -- CDF identifier.
INTEGER*4 var_num       ! in -- Variable number.
INTEGER*4 rec_num       ! in -- Record number.
INTEGER*4 indices(*)    ! in -- Dimension indices.
<type>    value         ! in -- Value. (<type> depends on the data type
                        !       of the variable).
INTEGER*4 status        ! out -- Completion status.
```

CDF_var_put is used to write a single value to a variable. CDF_var_hyper_put may be used to write more than one variable value with a single call.

The arguments to CDF_var_put are defined as follows.

id — The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.

var_num — The number of the variable to which to write. This number may be determined with a call to CDF_var_num (see Section 5.6.17).

rec_num — The record number at which to write.

indices — The array indices within the specified record at which to write. indices is a 1-dimensional array containing one element per CDF dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

value — The value to write. The value is written to the CDF from memory address value.

status — The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example writes values to the LATITUDE variable of a 2-dimensional CDF having dimension sizes 360,181. For LATITUDE, the record variance is NOVARY, the dimension variances are NOVARY,VARY, and the data type is CDF_INT2.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*2 lat                       ! Latitude value.
```

```
INTEGER*4 var_n                   ! Variable number.
INTEGER*4 rec_num                 ! Record number.
INTEGER*4 indices(2)              ! Dimension indices.

DATA rec_num/1/, indices/1,1/
.
.
var_n = CDF_var_num (id, 'LATITUDE')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then not a variable
                                                  ! number but rather a
                                                  ! warning/error code.

DO lat = -90, 90
   indices(2) = 91 + lat
   CALL CDF_var_put (id, var_n, rec_num, indices, lat, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END DO
.
.
```

Since the record variance is `NOVARY`, the record number (`rec_num`) is set to one (1). Also note that because the dimension variances are `NOVARY,VARY`, only the second dimension is varied as values are written (the values written are the same at each index of the first dimension in the conceptual view of the variable).

5.6.21 CDF_var_get (Standard interface)

```
SUBROUTINE CDF_var_get (id, var_num, rec_num, indices, value, status)

INTEGER*4 id             ! in  -- CDF identifier.
INTEGER*4 var_num        ! in  -- Variable number.
INTEGER*4 rec_num        ! in  -- Record number.
INTEGER*4 indices(*)     ! in  -- Dimension indices.
<type>    value          ! out -- Value (<type> depends on the data type
                         !        of the variable).
INTEGER*4 status         ! out -- Completion status.
```

`CDF_var_get` is used to read a single value from a variable. `CDF_var_hyper_get` may be used to read more than one variable value with a single call.

The arguments to `CDF_var_get` are defined as follows.

`id` The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.

`var_num` The number of the variable from which to read. This number may be determined with a call to `CDF_var_num` (see Section 5.6.17).

`rec_num` The record number at which to read.

indices The array indices within the specified record at which to read. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

value The value read. This buffer must be large enough to hold the variable value. `CDF_var_inquire` would be used to determine the variable's data type and number of elements (of that data type) at each variable value. The value is read from the CDF and placed into memory at address `value`.

status The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will read and hold an entire record of data from a variable. The CDF is 3-dimensional with sizes `180,91,10`. For the variable, the record variance is `VARY`, the dimension variances are `VARY,VARY,VARY`, and the data type is `CDF_REAL4`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
REAL*4    tmp(180,91,10)              ! Temperature values.
INTEGER*4 indices(3)                  ! Dimension indices.
INTEGER*4 var_n                       ! Variable number.
INTEGER*4 rec_num                     ! Record number.
INTEGER*4 d1, d2, d3                  ! Dimension index values.
.
.
var_n = CDF_var_num (id, 'Temperature')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then it is actually a
                                                  ! warning/error code.

rec_num = 13

DO d1 = 1, 180
  indices(1) = d1
  DO d2 = 1, 91
    indices(2) = d2
    DO d3 = 1, 10
      indices(3) = d3
      CALL CDF_var_get (id, var_n, rec_num, indices, tmp(d1,d2,d3), status)
      IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
    END DO
  END DO
END DO
```

.
.

5.6.22 CDF_var_hyper_put (Standard interface)

```
SUBROUTINE CDF_var_hyper_put (id, var_num, rec_start, rec_count, rec_interval,
1                             indices, counts, intervals, buffer, status)

INTEGER*4 id                ! in -- CDF identifier.
INTEGER*4 var_num           ! in -- Variable number.
INTEGER*4 rec_start         ! in -- Starting record number.
INTEGER*4 rec_count         ! in -- Number of records.
INTEGER*4 rec_interval      ! in -- Interval between records.
INTEGER*4 indices(*)        ! in -- Dimension indices of starting value.
INTEGER*4 counts(*)         ! in -- Number of values along each dimension.
INTEGER*4 intervals(*)      ! in -- Interval between values along each
                            !       dimension.
<type>    buffer            ! in -- Buffer of values (<type> depends on the
                            !       data type of the variable.)
INTEGER*4 status            ! out -- Completion status.
```

CDF_var_hyper_put is used to write a buffer of one or more values to a variable. It is important to know the variable majority of the CDF before using CDF_var_hyper_put because the values in the buffer to be written must be in the same majority. CDF_var_inquire is used to determine the variable majority of a CDF. Section 2.7.8 describes the variable majorities.

The arguments to CDF_var_hyper_put are defined as follows.

id The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.

var_num The number of the variable to which to write. This number may be determined with a call to CDF_var_num (see Section 5.6.17).

rec_start The record number at which to start writing.

rec_count The number of records to write.

rec_interval The interval between records for subsampling[1] (eg. an interval of 2 means write to every other record).

indices The indices (within each record) at which to start writing. indices is a 1-dimensional array containing one element per CDF dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

counts The number of values along each dimension to write. counts is a 1-dimensional array containing one element per CDF dimension. Each element of counts specifies the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

[1] "Subsampling" is not the best term to use when writing data but you should know what we mean.

intervals For each dimension, the interval between values for subsampling[2] (eg. an interval of 2 means write to every other value). **intervals** is a 1-dimensional array containing one element per CDF dimension. Each element of **intervals** specifies the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

buffer The buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address **buffer** are written to the CDF.

status The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example writes values to the `LATITUDE` variable of a 2-dimensional CDF having dimension sizes `360,181`. For `LATITUDE`, the record variance is `NOVARY`, the dimension variances are `NOVARY,VARY`, and the data type is `CDF_INT2`. This example is similar to the example in Section 5.6.20 except that is uses a single call to `CDF_var_hyper_put` rather than numerous calls to `CDF_var_put`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*2 lat                       ! Latitude value.
INTEGER*2 lats(181)                 ! Buffer of latitude values.
INTEGER*4 var_n                     ! Variable number.
INTEGER*4 rec_start                 ! Record number.
INTEGER*4 rec_count                 ! Record counts.
INTEGER*4 rec_interval              ! Record interval.
INTEGER*4 indices(2)                ! Dimension indices.
INTEGER*4 counts(2)                 ! Dimension counts.
INTEGER*4 intervals(2)              ! Dimension intervals.

DATA rec_start/1/, rec_count/1/, rec_interval/1/,
1    indices/1,1/, counts/1,181/, intervals/1,1/
.
.
var_n = CDF_var_num (id, 'LATITUDE')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then not a variable
                                                  ! number but rather a
                                                  ! warning/error code.

DO lat = -90, 90
  lats(91+lat) = lat
```

[2] Again, not the best term.

```
   END DO

   CALL CDF_var_hyper_put (id, var_n, rec_start, rec_count, rec_interval,
   1                       indices, counts, intervals, lats, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
   .
   .
```

5.6.23 CDF_var_hyper_get (Standard interface)

```
SUBROUTINE CDF_var_hyper_get (id, var_num, rec_start, rec_count, rec_interval,
1                             indices, counts, intervals, buffer, status)

INTEGER*4 id              ! in -- CDF identifier.
INTEGER*4 var_num         ! in -- Variable number.
INTEGER*4 rec_start       ! in -- Starting record number.
INTEGER*4 rec_count       ! in -- Number of records.
INTEGER*4 rec_interval    ! in -- Subsampling interval between records.
INTEGER*4 indices(*)      ! in -- Dimension indices of starting value.
INTEGER*4 counts(*)       ! in -- Number of values along each dimension.
INTEGER*4 intervals(*)    ! in -- Subsampling intervals along each dimension.
<type>    buffer          ! out -- Buffer of values (<type> depends on the
                          !        data type of the variable).
INTEGER*4 status          ! out -- Completion status.
```

CDF_var_hyper_get is used to read a buffer of one or more values from a variable. It is important to know the variable majority of the CDF before using CDF_var_hyper_get because the values placed into the buffer will be in that majority. CDF_var_inquire is used to determine the variable majority of a CDF. Section 2.7.8 describes the variable majorities.

The arguments to CDF_var_hyper_get are defined as follows.

id The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.

var_num The number of the variable from which to read. This number may be determined with a call to CDF_var_num (see Section 5.6.17).

rec_start The record number at which to start reading.

rec_count The number of records to read.

rec_interval The interval between records for subsampling (eg. an interval of 2 means read every other record).

indices The indices (within each record) at which to start reading. indices is a 1-dimensional array containing one element per CDF dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

counts The number of values along each dimension to read. counts is a 1-dimensional array containing one element per CDF dimension. Each element of `counts` specifies the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`intervals` For each dimension, the interval between values for subsampling (eg. an interval of 2 means read every other value). `intervals` is a 1-dimensional array containing one element per CDF dimension. Each element of `intervals` specifies the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

`buffer` The buffer of values read. The majority of the values in this buffer will be the same as that of the CDF. This buffer must be large enought to hold the values. The values are read from the CDF and placed into memory starting at address `buffer`. `CDF_var_inquire` would be used to determine the variable's data type and number of elements (of that data type) at each value.

`status` The completion status code. Section 5.7 explains how to interpret status codes.

Example(s)

The following example will read an entire record of data from a variable. The CDF is 3-dimensional with sizes `180,91,10`. The CDF's variable majority is `COL_MAJOR`. For the variable, the record variance is `VARY`, the dimension variances are `VARY,VARY,VARY`, and the data type is `CDF_REAL4`. This example is similar to the example in Section 5.6.21 except that it uses a single call to `CDF_var_hyper_get` rather than numerous calls to `CDF_var_get`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                         ! CDF identifier.
INTEGER*4 status                     ! Returned status code.
REAL*4    tmp(180,91,10)             ! Temperature values.
INTEGER*4 var_n                      ! Variable number.
INTEGER*4 rec_start                  ! Record number.
INTEGER*4 rec_count                  ! Record counts.
INTEGER*4 rec_interval               ! Record interval.
INTEGER*4 indices(3)                 ! Dimension indices.
INTEGER*4 counts(3)                  ! Dimension counts.
INTEGER*4 intervals(3)               ! Dimension intervals.

DATA rec_start/13/, rec_count/1/, rec_interval/1/,
1    indices/1,1,1/, counts/180,91,10/, intervals/1,1,1/
.
.
var_n = CDF_var_num (id, 'Temperature')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then it is actually a
                                                  ! warning/error code.
```

```
   CALL CDF_var_hyper_get (id, var_n, rec_start, rec_count, rec_interval,
   1                       indices, counts, intervals, tmp, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
   .
   .
```

Note that if the CDF's variable majority had been ROW_MAJOR, the tmp array would have been declared REAL*4 tmp(10,91,180) to achieve the same results.

5.6.24 CDF_lib (Internal interface)

```
SUBROUTINE CDF_lib (..., status)

INTEGER*4 status                ! out -- Completion status.
```

The Internal interface consists of only one function, CDF_lib. CDF_lib can be used to perform all possible operations on a CDF. (In fact, all of the Standard interface functions are implemented using the Internal interface). CDF_lib must be used to perform operations not possible with the Standard interface functions. These operations would involve CDF features added after the Standard interface functions had been defined (eg. specifying a single-file format for a CDF, specifying a fill value for a variable, or allocating initial records for a variable). Note that CDF_lib can also be used to perform certain operations more efficiently than with the Standard interface functions.

CDF_lib takes a variable number of arguments which specify one or more operations to be performed. The operations are performed according to the order of the arguments. Each operation is performed on an item. An item may be either an object (eg. a CDF, variable, or attribute) or a state (eg. a CDF's format or a variable's data type/number of elements). The possible operations and corresponding items (on which to perform those operations) are described in detail later in this section.

Example(s)

The easiest way to explain how to use CDF_lib would be to start with a few examples. The following example will show how a CDF would be created with the single-file format.

```
   .
   .
   INCLUDE '<path>cdf.inc'
   .
   .
   INTEGER*4 id                    ! CDF identifier.
   INTEGER*4 status                ! Returned status code.
   CHARACTER CDF_name*5            ! Name of the CDF.
   INTEGER*4 num_dims              ! Number of dimensions.
   INTEGER*4 dim_sizes(2)          ! Dimension sizes.
   INTEGER*4 encoding              ! Data encoding.
```

```
INTEGER*4 majority                 ! Variable data majority.
INTEGER*4 format                   ! Format of CDF.

DATA CDF_name/'test1'/, num_dims/2/, dim_sizes/100,200/,
1    encoding/HOST_ENCODING/, majority/ROW_MAJOR/, format/SINGLE_FILE/
.
.
CALL CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority,
1                id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_lib (SELECT_, CDF_, id,
1             PUT_, CDF_FORMAT_, format,
2             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

The call to `CDF_create` created the CDF as expected but the default format is `MULTI_FILE` (a separate file for each variable in the CDF) The call to `CDF_lib` is then used to change the format to `SINGLE_FILE` (which must be done before any variables are created in the CDF).

The arguments to `CDF_lib` in this example are explained as follows.

`SELECT_`	The first operation to be performed. In this case an item is going to be selected. `SELECT_` is defined in `cdf.inc` (as are all CDF constants).
`CDF_`	The item to be selected — the current CDF (the CDF on which subsequent operations are to be performed).
`id`	The identifier of the CDF being selected.
`PUT_`	The next operation to be performed. In this case an item is going to be put to the CDF (a new format). `PUT_` is defined in `cdf.inc` (as are all CDF constants).
`CDF_FORMAT_`	The item to be put. In this case it is the CDF's format. One required argument for this operation follows.
`format`	The actual format for the CDF. Depending on the item being put, one or more arguments would have been necessary. In this case only one argument is necessary. `format` contains the value `SINGLE_FILE`.
`NULL_`	This argument could have been one of two things. It could have been another item to put (followed by the arguments required for that item) or it could have been a new operation to perform. In this case it is a new operation to perform — the `NULL_` operation. `NULL_` indicates the end of the call to `CDF_lib`. Specifying `NULL_` at the end of the argument list is required because not all FORTRAN compilers/operating systems provide the ability for a called function to determine how many arguments were passed in by the calling function.
`status`	The completion status code. Section 5.7 explains how to interpret status codes.

The next example shows how the same CDF could have been created using only one call to `CDF_lib` (the declarations would be the same).

```
.
.
CALL CDF_lib (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id,
1             PUT_, CDF_ENCODING_, encoding,
2                   CDF_MAJORITY_, majority,
3                   CDF_FORMAT_, format,
4             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

The purpose of each argument is as follows.

`CREATE_`	The first operation to be performed. In this case something will be created.
`CDF_`	The item to be created — a CDF in this case. There are four required argument which must follow. When a CDF is created (with `CDF_lib`), the format defaults to multi-file (`MULTI_FILE`), the data encoding defaults to host (`HOST_ENCODING`), and the variable majority defaults to row major (`ROW_MAJOR`).
`CDF_name`	The name of the CDF.
`num_dims`	The number of dimensions in the CDF.
`dim_sizes`	The dimension sizes.
`id`	The identifier to be used when referencing the created CDF in subsequent operations.
`PUT_`	This argument could have been one of two things. Another item to create or a new operation to perform. In this case it is another operation to perform — something will be put.
`CDF_ENCODING_`	The item to be put — in this case the CDF's encoding. Note that the CDF did not have to be selected. It was automatically selected as the current CDF when it was created. One required argument follows.
`encoding`	The encoding to be put to the CDF.
`CDF_MAJORITY_`	This argument could have been one of two things. Another item to put or a new operation to perform. In this case it is another item to put — the CDF's majority. One required argument follows.
`majority`	The majority to be put to the CDF.
`CDF_FORMAT_`	Once again this argument could have been either another item to put or a new operation to perform. It is another item to put — the CDF's format. One required argument follows.
`format`	The format to be put to the CDF.
`NULL_`	This argument could have been either another item to put or a new operation

to perform. Here it is another operation to perform — the `NULL_` operation which ends the call to `CDF_lib`.

`status` The completion status code. Section 5.7 explains how to interpret status codes.

Note that the operations are performed in the order that they appear in the argument list. The CDF had to be created before the encoding, majority, and format could be specified (put).

Current Objects/States (Items)

The use of `CDF_lib` requires that an application be aware of the current objects/states maintained by the CDF library. The following objects/states are used by the CDF library when performing operations.

CDF — A CDF operation is always performed on the current CDF. The current CDF is implicitly selected whenever a CDF is opened or created. The current CDF may be explicitly selected using the `<SELECT_,CDF_>`[3] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.

NOTE

The current CDF is not maintained by the CDF library from one call to the next of `CDF_lib`. A current CDF (or CDFs) must be selected in each call to `CDF_lib` (if an operation is to be performed on a CDF).

variable — A variable operation is always performed on the current variable in the current CDF. For each open CDF a current variable is maintained. This current variable is implicitly selected when a variable is created (in the current CDF) or it may be explicitly selected with the `<SELECT_,VAR_>` or `<SELECT_,VAR_NAME_>` operations. There is no current variable in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

attribute — An attribute operation is always performed on the current attribute in the current CDF. For each open CDF a current attribute is maintained. This current attribute is implicitly selected when an attribute is created (in the current CDF) or it may be explicitly selected with the `<SELECT_,ATTR_>` or `<SELECT_,ATTR_NAME_>` operations. There is no current attribute in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

entry number — An attribute entry operation is always performed on the current entry number in the current CDF for the current attribute in that CDF. For each open CDF a current entry number is maintained. This current entry number must be explicitly selected with the `<SELECT_,ENTRY_>` operation (there is no implicit or default selection of the current entry number for a CDF). Note that the current

[3] This notation is used to specify an operation to be performed on an item. The syntax is: `<operation,item>`.

entry number is maintained for a CDF (not each attribute) — it applies to all of the attributes in that CDF.

record number

A variable read or write operation is always performed at (for regular reads and writes) or starting at (for hyper reads and writes) the current record number for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current record number is initialized to one (1). It may be explicitly selected using the `<SELECT_,CDF_RECNUMBER_>` operation. Note that the current record number is maintained for a CDF (not each variable) — it applies to all of the variables in that CDF.

record count

A variable hyper read or write operation is always performed using the current record count for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current record count is initialized to one (1). It may be explicitly selected using the `<SELECT_,CDF_RECCOUNT_>` operation. Note that the current record count is maintained for a CDF (not each variable) — it applies to all of the variables in that CDF.

record interval

A variable hyper read or write operation is always performed using the current record interval for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current record interval is initialized to one (1). It may be explicitly selected using the `<SELECT_,CDF_RECINTERVAL_>` operation. Note that the current record interval is maintained for a CDF (not each variable) — it applies to all of the variables in that CDF.

dimension indices

A variable read or write operation is always performed at (for regular reads and writes) or starting at (for hyper reads and writes) the current dimension indices for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current dimension indices are initialized to ones (1,1,...). They may be explicitly selected using the `<SELECT_,CDF_DIMINDICES_>` operation. Note that the current dimension indices are maintained for a CDF (not each variable) — they apply to all of the variables in that CDF. For 0-dimensional CDFs the current dimension indices are not applicable.

dimension counts

A variable hyper read or write operation is always performed using the current dimension counts for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current dimension counts are initialized to the CDF's dimension sizes (which specifies the entire record array). They may be explicitly selected using the `<SELECT_,CDF_DIMCOUNTS_>` operation. Note that the current dimension counts are maintained for a CDF (not each variable) — they apply to all of the variables in that CDF. For 0-dimensional CDFs the current dimension counts are not applicable.

dimension intervals

A variable hyper read or write operation is always performed using the current dimension intervals for the current CDF and on the current variable in that CDF. When a CDF is opened or created its current dimension intervals are initialized to ones (1,1,...). They may be explicitly selected using the <SELECT_,CDF_DIMINTERVALS_> operation. Note that the current dimension intervals are maintained for a CDF (not each variable) — they apply to all of the variables in that CDF. For 0-dimensional CDFs the current dimension intervals are not applicable.

status code
: When inquiring the explanation of a CDF status code, the text returned is always for the current status code. One current status code is maintained for the entire CDF library (regardless of the number of open CDFs). The current status code may be selected using the <SELECT_,CDF_STATUS_> operation. There is no default current status code. The current status code is NOT the status code from the last operation performed.

  NOTE

  The current status code is not maintained by the CDF library from one call to the next of CDF_lib.

Returned Status

CDF_lib returns a status code of type INTEGER*4. Since more than one operation may be performed with a single call to CDF_lib, the following rules apply.

1. The first error detected aborts the call to CDF_lib and the corresponding status code is returned.
2. In the absence of any errors, the status code for the last warning detected is returned.
3. In the absence of any errors or warnings, the status code for the last informational condition is returned.
4. In the absence of any errors, warnings, or informational conditions, CDF_OK is returned.

Section 5.7 explains how to interpret status codes and Appendix B lists the possible status codes.

Indentation/Style

Indentation should be used to make calls to CDF_lib readable. The following example shows a call to CDF_lib using proper indentation.

```
CALL CDF_lib (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id,
1             PUT_, CDF_FORMAT_, format,
2                   CDF_MAJORITY_, majority,
3             CREATE_, ATTR_, attr_name, scope, attr_num,
4                      VAR_, var_name, data_type, num_elements,
5                            rec_vary, dim_varys, var_num,
6             NULL_, status)
```

Note that the operations (CREATE_, PUT_, and NULL_) are indented the same and that the items (CDF_, CDF_FORMAT_, CDF_MAJORITY_, ATTR_, and VAR_) are indented the same under their corresponding operations.

The following example shows the same call to CDF_lib without the proper indentation.

```
   CALL CDF_lib (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id, PUT_,
   1             CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
   2             ATTR_, attr_name, scope, attr_num, VAR_, var_name, data_type,
   3             num_elements, rec_vary, dim_varys, var_num, NULL_, status)
```

The need for proper indentation to ensure the readability of your applications should be obvious.[4]

Syntax

CDF_lib has a variable number of arguments. There must always be at least one argument (the first operation to perform). The maximum number of arguments is not limited by CDF but rather the FORTRAN compiler and operating system being used. Under normal circumstances that limit would never be reached (or even approached). Note that a call to CDF_lib with a large number of arguments can always be broken up into two or more calls to CDF_lib with fewer arguments.

Operations/items/arguments...

The syntax for CDF_lib is as follows...

```
   CALL CDF_lib (op1, item1, arg1, arg2, ...argN,
  +                   item2, arg1, arg2, ...argN,
                        .
                        .
  +                   itemN, arg1, arg2, ...argN,
  +              op2, item1, arg1, arg2, ...argN,
  +                   item2, arg1, arg2, ...argN,
                        .
                        .
  +                   itemN, arg1, arg2, ...argN,
                  .
                  .
  +              opN, item1, arg1, arg2, ...argN,
  +                   item2, arg1, arg2, ...argN,
                        .
                        .
  +                   itemN, arg1, arg2, ...argN,
  +              NULL_, status)
```

[4] If it isn't, call us, we can help you.

where `op`x is an operation to perform, `item`x is the item on which to perform the operation, and `arg`x is a required argument for the operation. The `NULL_` operation must be used to end the call to `CDF_lib`. The completion status from `CDF_lib` is placed into `status`.

The supported operations are as follows.

`CREATE_`	Used to create an item.
`OPEN_`	Used to open an item.
`DELETE_`	Used to delete an item.
`CLOSE_`	Used to close an item.
`SELECT_`	Used to select the value of an item.
`CONFIRM_`	Used to confirm the value of an item.
`PUT_`	Used to put (write) something to an item.
`GET_`	Used to get (read) something from an item.

For each operation, the supported items and required arguments are as follows.

`CREATE_` The following items may be created...

`CDF_` A new CDF will be created. It is illegal to create a CDF that already exists. The created CDF implicitly becomes the current CDF. Required arguments are...

in: `CHARACTER CDF_name*(*)`	Pathname of the CDF to be created (do not specify an extension). This can be at most `CDF_PATHNAME_LEN` characters. A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems). NOTE: On UNIX systems pathnames are case sensitive. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `CDF_name` MUST be of type `CHARACTER`.
in: `INTEGER*4 num_dims`	Number of dimensions. This can be as few as zero (0) and at most `CDF_MAX_DIMS`.
in: `INTEGER*4 dim_sizes(*)`	Dimension sizes. `dim_sizes` is a 1-dimensional array containing `num_dims` elements. Each element of `dim_sizes` specifies the corresponding dimension size. Each size must be greater than zero

	(0). For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
out: INTEGER*4 id	CDF identifier to be used in subsequent operations on the CDF.

A CDF is created with a default format of multi-file (MULTI_FILE), a default data encoding of host (HOST_ENCODING), and a default variable majority of row major (ROW_MAJOR). These defaults can be changed with the corresponding PUT_ operations (of CDF_FORMAT_, CDF_ENCODING_, and CDF_MAJORITY_ respectively).

<CLOSE_,CDF_> MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk.

VAR_

A new variable will be created in the current CDF. A variable with the same name must not already exist in the CDF. The created variable implicitly becomes the current variable (in the current CDF). Required arguments are...

in: CHARACTER var_name*(*)	Name of the variable to be created. This can be at most CDF_VAR_NAME_LEN characters. Variable names are case-sensitive. NOTE: On UNIX systems, for the proper operation of CDF_lib, var_name MUST be of type CHARACTER.
in: INTEGER*4 data_type	Data type of the new variable. Specify one of the data types described in Section 5.5.3.
in: INTEGER*4 num_elements	Number of elements of the data type at each variable value. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in each string. A string exists at each variable value. For the non-character string data types this must be one (1) — multiple elements are not allowed for non-character string data types.
in: INTEGER*4 rec_vary	Record variance. Specify one of the variances described in Section 5.5.6.
in: INTEGER*4 dim_varys(*)	Dimension variances. dim_varys is a 1-dimensional array containing one element per CDF dimension. Each element of dim_varys specifies the corresponding dimension variance. For each dimen-

	sion specify one of the variances described in Section 5.5.6. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).
`out: INTEGER*4 var_num`	Number assigned to the new variable. This number must be used in subsequent CDF function calls when referring to this variable. An existing variable's number may be determined with the `<GET_,VAR_NUMBER_>` operation.

`ATTR_` A new attribute will be created in the current CDF. An attribute with the same name must not already exist in the CDF. The created attribute implicitly becomes the current attribute (in the current CDF). Required arguments are...

`in: CHARACTER attr_name*(*)`	Name of the attribute to be created. This can be at most `CDF_ATTR_NAME_LEN` characters. Attribute names are case-sensitive. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `attr_name` MUST be of type `CHARACTER`.
`in: INTEGER*4 scope`	Scope of the new attribute. Specify one of the scopes described in Section 5.5.7.
`out: INTEGER*4 attr_num`	Number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may be determined with the `<GET_,ATTR_NUMBER_>` operation.

`OPEN_` The follow items may be opened...

`CDF_` Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are...

`in: CHARACTER CDF_name*(*)`	Pathname of the CDF to be opened. This can be at most `CDF_PATHNAME_LEN` characters. A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and

		environment variables on UNIX systems).
		NOTE: On UNIX systems pathnames are case sensitive.
		NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `CDF_name` MUST be of type `CHARACTER`.
	`out: INTEGER*4 id`	CDF identifier to be used in subsequent operations of the CDF.

`DELETE_` The following items may be deleted...

`CDF_` Deletes the current CDF. A CDF must be opened before it can be deleted. When a CDF is deleted, there is no longer a current CDF. There are no arguments.

`CLOSE_` The following items may be closed...

`CDF_` Closes the current CDF. When a CDF is closed, there is no longer a current CDF. A CDF must be closed to ensure that it will be properly written to disk. There are no arguments.

`VAR_` Closes the current variable (in the current CDF). There are no arguments.

`SELECT_` The following items may be selected...

`CDF_` Explicitly selects the current CDF. Required arguments are...

`in: INTEGER*4 id`	Identifier of the CDF. This identifier must have been initialized by a successful `<CREATE_,CDF_>` or `<OPEN_,CDF_>` operation.

`CDF_STATUS_` Selects the current status code. Required arguments are...

`in: INTEGER*4 status`	CDF status code.

`VAR_` Explicitly selects the current variable (in the current CDF) by number. Required arguments are...

`in: INTEGER*4 var_num`	Variable number.

`VAR_NAME_` Explicitly selects the current variable (in the current CDF) by name. Required arguments are...

`in: CHARACTER var_name*(*)`	Variable name. This may be at most `CDF_VAR_NAME_LEN` characters.
	NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `var_name` MUST be of type `CHARACTER`.

CDF_RECNUMBER_ Selects the current record number for all variables in the current CDF. Required arguments are...

in:	`INTEGER*4 rec_num`	Record number.

CDF_RECCOUNT_ Selects the current record count for all variables in the current CDF. Required arguments are...

in:	`INTEGER*4 rec_count`	Record count.

CDF_RECINTERVAL_ Selects the current record interval for all variables in the current CDF. Required arguments are...

in:	`INTEGER*4 rec_interval`	Record interval.

CDF_DIMINDICES_ Selects the current dimension indices for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	`INTEGER*4 indices(*)`	Dimension indices. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_DIMCOUNTS_ Selects the current dimension counts for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	`INTEGER*4 counts(*)`	Dimension counts. `counts` is a 1-dimensional array containing one element per CDF dimension. Each element of `counts` specifies the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_DIMINTERVALS_ Selects the current dimension intervals for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	`INTEGER*4 intervals(*)`	Dimension intervals. `intervals` is a 1-dimensional array containing one element per CDF dimension. Each element of `intervals` specifies the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

VAR_SEQPOS_ Selects the current position for sequential access for the current variable (in the current CDF). Note that the current position is selected for each variable individually. Required arguments are...

in: INTEGER*4 rec_num	Record number.
in: INTEGER*4 indices(*)	Dimension indices. indices is a 1-dimensional array containing one element per CDF dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

ATTR_ Explicitly selects the current attribute (in the current CDF). Required arguments are...

in: INTEGER*4 attr_num	Attribute number.

ATTR_NAME_ Explicitly selects the current attribute (in the current CDF) by name. Required arguments are...

in: CHARACTER attr_name*(*)	Attribute name. This may be at most CDF_ATTR_NAME_LEN characters. NOTE: On UNIX systems, for the proper operation of CDF_lib, attr_name MUST be of type CHARACTER.

ENTRY_ Selects the current entry number for all attributes in the current CDF. Required arguments are...

in: INTEGER*4 entry_num	Entry number.

CONFIRM_ The following items may be confirmed...

CDF_NAME_ Confirms the pathname of the current CDF. Required arguments are...

out: CHARACTER CDF_name*(*)	Pathname of the CDF. This character string must be large enough to hold CDF_PATHNAME_LEN characters. NOTE: On UNIX systems, for the proper operation of CDF_lib, CDF_name MUST be of type CHARACTER.

VAR_ Confirms the current variable (in the current CDF). Required arguments are...

out: INTEGER*4 var_num	Variable number.

CDF_RECNUMBER_ Confirms the current record number for all variables in the current CDF. Required arguments are...

`out: INTEGER*4 rec_num`	Record number.

CDF_RECCOUNT_ Confirms the current record count for all variables in the current CDF. Required arguments are...

`out: INTEGER*4 rec_count`	Record count.

CDF_RECINTERVAL_ Confirms the current record interval for all variables in the current CDF. Required arguments are...

`out: INTEGER*4 rec_interval`	Record interval.

CDF_DIMINDICES_ Confirms the current dimension indices for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

`out: INTEGER*4 indices(*)`	Dimension indices. `indices` is a 1-dimensional array containing one element per CDF dimension. Each element of `indices` receives the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_DIMCOUNTS_ Confirms the current dimension counts for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

`out: INTEGER*4 counts(*)`	Dimension counts. `counts` is a 1-dimensional array containing one element per CDF dimension. Each element of `counts` receives the corresponding dimension count. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_DIMINTERVALS_ Confirms the current dimension intervals for all variables in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

`out: INTEGER*4 intervals(*)`	Dimension intervals. `intervals` is a 1-dimensional array containing one element per CDF dimension. Each element of `intervals` receives the corresponding dimension interval. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

VAR_SEQPOS_ Confirms the current position for sequential access for the current variable (in the current CDF). Required arguments are...

out: INTEGER*4 rec_num	Record number.
out: INTEGER*4 indices(*)	Dimension indices. indices is a 1-dimensional array containing one element per CDF dimension. Each element of indices receives the corresponding dimension index. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

ATTR_ Confirms the current attribute (in the current CDF). Required arguments are...

out: INTEGER*4 attr_num	Attribute number.

ENTRY_ Confirms the current entry number for all attributes in the current CDF. Required arguments are...

out: INTEGER*4 entry_num	Entry number.

PUT_ The following items may be written (put)...

CDF_ARRAY_ Respecifies the dimensionality of the current CDF. A CDF's dimensionality may not be changed after any variables have been created. Required arguments are...

in: INTEGER*4 num_dims	Number of dimensions. The number of dimensions must be in the range from zero (0) to CDF_MAX_DIMS (inclusive).
in: INTEGER*4 dim_sizes(*)	Dimension sizes. dim_sizes is a 1-dimensional array containing num_dims elements. Each element of dim_sizes specifies the corresponding dimension size. Each size must be greater than zero (0). For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_ENCODING_ Respecifies the data encoding of the current CDF. A CDF's data encoding may not be changed after any variable values (including the fill value) or attribute entries have been written. Required arguments are...

in: INTEGER*4 encoding	Data encoding. Specify one of the encodings described in Section 5.5.4.

`CDF_MAJORITY_` Respecifies the variable majority of the current CDF. A CDF's variable majority may not be changed after any variable values have been written. Required arguments are...

- in: `INTEGER*4 majority` — Variable majority. Specify one of the majorities described in Section 5.5.5.

`CDF_FORMAT_` Respecifies the format of the current CDF. A CDF's format may not be changed after any variables have been created. Required arguments are...

- in: `INTEGER*4 format` — CDF format. Specify one of the formats described in Section 5.5.2.

`VAR_NAME_` Renames the current variable (in the current CDF). A variable with the same name must not already exist in the CDF. Required arguments are...

- in: `CHARACTER var_name*(*)` — Name of the variable. This may consist of at most `CDF_VAR_NAME_LEN` characters.

  NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `var_name` MUST be of type `CHARACTER`.

`VAR_DATASPEC_` Respecifies the data specification (data type and number of elements) of the current variable (in the current CDF). A variable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any variable values (including the fill value) have been written. Data specifications are considered equivalent if the data types are equivalent (see Section 2.5.5) and the number of elements are the same. Required arguments are...

- in: `INTEGER*4 data_type` — Variable data type. Specify one of the data types described in Section 5.5.3.
- in: `INTEGER*4 num_elements` — Number of elements of the data type at each variable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in each string (a string exists at each variable value). For non-character string data types this must be one (1) — multiple elements are not allowed for non-character string data types.

`VAR_RECVARY_` Respecifies the record variance of the current variable (in the current CDF). A variable's record variance may not be changed if any values have been written to that variable (except for an explicit fill value — it may have been written). Required arguments are...

in:	INTEGER*4 rec_vary	Record variance. Specify one of the variances described in Section 5.5.6.

VAR_DIMVARYS_

Respecifies the dimension variances of the current variable (in the current CDF). A variable's dimension variances may not be changed if any values have been written to that variable (except for an explicit fill value — it may have been written). For 0-dimensional CDFs this operation is not applicable. Required arguments are...

in:	INTEGER*4 dim_varys(*)	Dimension variances. dim_varys is a 1-dimensional array containing one element per CDF dimension. Each element of dim_varys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 5.5.6. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

VAR_FILLVALUE_

Specifies the fill value for the current variable (in the current CDF). A variable's fill value may be specified (or respecified) at any time without affecting already written values (including where fill values were used). Section 2.7.11 describes variable fill values. Required arguments are...

in:	<type> value	Fill value. <type> depends on the data type of the variable. The fill value is written to the CDF from memory address value. NOTE: On UNIX systems, if the variable has a character string data type, then <type> MUST be CHARACTER to ensure the proper operation of CDF_lib.

VAR_INITIALRECS_

Specifies the number of records to initially write to the current variable (in the current CDF). This may be specified only once per variable and before any other records have been written to that variable. If a fill value has not yet been specified, the default is used (see Section 2.7.11). If a fill value has been specified, that value is written to the records. Section 2.7.4 describes initial records. Required arguments are...

in:	INTEGER*4 n_initial_recs	Number of records to write.

VAR_EXTENDRECS_

Specifies the number of physical records to extend the current variable (in the current CDF) whenever the variable needs to be extended. Section 2.7.4 describes extend records. Required arguments are...

in:	INTEGER*4 n_extend_recs	Number of records to extend.

VAR_DATA_

Writes one value to the current variable (in the current CDF). The value is written at the current record number and current indices (for the current CDF). Required arguments are...

in: `<type> value` — Value. `<type>` depends on the data type of the variable. The value is written to the CDF from memory address `value`.

NOTE: On UNIX systems, if the variable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

VAR_HYPERDATA_

Writes one or more values to the current variable (in the current CDF). The values are written based on the current record number, count, and interval and the current dimension indices, counts, and intervals (for the current CDF). Required arguments are...

in: `<type> buffer` — Values. `<type>` depends on the data type of the variable. The values starting at memory address `buffer` are written to the CDF.

NOTE: On UNIX systems, if the variable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

VAR_SEQDATA_

Writes one value to the current variable (in the current CDF) at the current sequential position for that variable. After the write, the sequential position is automatically incremented to the next value (crossing a record boundary if necessary). If the sequential position is past the last record for the variable, the variable is extended as necessary. Required arguments are...

in: `<type> value` — Value. `<type>` depends on the data type of the variable. The value is written to the CDF from memory address `value`.

NOTE: On UNIX systems, if the variable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

ATTR_SCOPE_

Respecifies the scope for the current attribute (in the current CDF). Required arguments are...

in: `INTEGER*4 scope` — Attribute scope. Specify one of the scopes described in Section

5.5.7.

ATTR_NAME_

Renames the current attribute (in the current CDF). An attribute with the same name must not already exist in the CDF. Required arguments are...

in:	`CHARACTER attr_name*(*)`	Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `attr_name` MUST be of type `CHARACTER`.

ENTRY_DATA_

Writes an entry to the current attribute at the current entry number (in the current CDF). An existing entry may be overwritten with a new entry having the same data specification (data type and number of elements) or a different data specification. Required arguments are...

in:	`INTEGER*4 data_type`	Entry data type. Specify one of the data types described in Section 5.5.3.
in:	`INTEGER*4 num_elements`	Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string. For all other data types, this is the number of elements in an array of that data type.
in:	`<type> value`	Value(s). `<type>` depends on the data type of the entry. The entry value is written to the CDF from memory address `value`. NOTE: On UNIX systems, if the entry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

GET_ The following items may be read (gotten)...

CDF_NUMDIMS_

Inquires the number of dimensions in the current CDF. Required arguments are...

out:	`INTEGER*4 num_dims`	Number of dimensions.

CDF_DIMSIZES_

Inquires the size of each dimension in the current CDF. For 0-dimensional CDFs this operation is not applicable. Required arguments are...

	out: INTEGER*4 dim_sizes(*)	Dimension sizes. dim_sizes is a 1-dimensional array containing one element per CDF dimension. Declaring dim_sizes as INTEGER*4 dim_sizes(CDF_MAX_DIMS) would work in all cases. Each element of dim_sizes receives the corresponding dimension size. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

CDF_ENCODING_ Inquires the data encoding of the current CDF. Required arguments are...

out: INTEGER*4 encoding	Data encoding. The encodings are described in Section 5.5.4.

CDF_MAJORITY_ Inquires the variable majority of the current CDF. Required arguments are...

out: INTEGER*4 majority	Variable majority. The majorities are described in Section 5.5.5.

CDF_FORMAT_ Inquires the format of the current CDF. Required arguments are...

out: INTEGER*4 format	CDF format. The formats are described in Section 5.5.2.

CDF_COPYRIGHT_ Reads the copyright notice for the CDF library which created the current CDF. Required arguments are...

out: CHARACTER copy_right*(*)	CDF copyright text. This character string must be large enough hold CDF_COPYRIGHT_LEN characters. NOTE: On UNIX systems, for the proper operation of CDF_lib, copy_right MUST be of type CHARACTER.

LIB_COPYRIGHT_ Reads the copyright notice for the CDF library being used. Required arguments are...

out: CHARACTER copy_right*(*)	CDF library copyright text. This character string must be large enough hold CDF_COPYRIGHT_LEN characters. NOTE: On UNIX systems, for the proper operation of CDF_lib, copy_right MUST be of type CHARACTER.

CDF_NUMVARS_ Inquires the number of variables in the current CDF. Required arguments are...

out: INTEGER*4 num_vars	Number of variables.

CDF_NUMATTRS_ Inquires the number of attributes in the current CDF. Required arguments are...

out: INTEGER*4 num_attrs	Number of attributes.

CDF_MAXREC_ Inquires the maximum record number in the current CDF. A value of zero (0) indicates that no records have been written to the CDF. The maximum record number for an individual variable may be inquired using <GET_,VAR_MAXREC_>. Required arguments are...

out: INTEGER*4 max_rec	Maximum record number in the CDF.

CDF_VERSION_ Inquires the version number of the CDF library which created the current CDF. Required arguments are...

out: INTEGER*4 version	Version number.

CDF_RELEASE_ Inquires the release number of the CDF library which created the current CDF. Required arguments are...

out: INTEGER*4 release	Release number.

CDF_INCREMENT_ Inquires the incremental number of the CDF library which created the current CDF. Required arguments are...

out: INTEGER*4 increment	Incremental number.

LIB_VERSION_ Inquires the version number of the CDF library being used. Required arguments are...

out: INTEGER*4 version	Version number.

LIB_RELEASE_ Inquires the release number of the CDF library being used. Required arguments are...

out: INTEGER*4 release	Release number.

LIB_INCREMENT_ Inquires the incremental number of the CDF library being used. Required arguments are...

out: INTEGER*4 increment	Incremental number.

LIB_subINCREMENT_ Inquires the subincremental character of the CDF library being used. Required arguments are...

out: CHARACTER*1 subincrement	Subincremental character. NOTE: On UNIX systems, for the proper operation of CDF_lib, subincrement MUST be of type CHARACTER.

VAR_NAME_ Inquires the name of the current variable (in the current CDF). Required arguments are...

out: CHARACTER var_name*(*)	Name of the variable. This character string must be large enough to hold CDF_VAR_NAME_LEN characters. NOTE: On UNIX systems, for the proper operation of CDF_lib, var_name MUST be of type CHARACTER.

VAR_DATATYPE_ Inquires the data type of the current variable (in the current CDF). Required arguments are...

out: INTEGER*4 data_type	Variable data type. The data types are described in Section 5.5.3.

VAR_NUMELEMS_ Inquires the number of elements (of the data type) for the current variable (in the current CDF). Required arguments are...

out: INTEGER*4 num_elements	Number of elements of the data type at each variable value. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string (each variable value consists of the entire string). For all other data types, this will always be one (1) — multiple elements at each variable value are not allowed for non-character string data types.

VAR_RECVARY_ Inquires the record variance of the current variable (in the current CDF). Required arguments are...

out: INTEGER*4 rec_vary	Record variance. The variances are described in Section 5.5.6.

VAR_DIMVARYS_ Inquires the dimension variances of the current variable (in the current CDF). For 0-dimensional CDFs this operation is not applicable. Required arguments are...

out: INTEGER*4 dim_varys(*)	Dimension variances. dim_varys is a 1-dimensional array containing one element per CDF dimension. Each element of dim_varys receives the corresponding dimension variance. The variances are described in Section 5.5.6. For 0-dimensional CDFs this argument is ignored (but a place holder is necessary).

VAR_NUMBER_ Gets the number of the named variable (in the current CDF). Required arguments are...

in: `CHARACTER var_name*(*)`	The variable name. This may be at most `CDF_VAR_NAME_LEN` characters. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `var_name` MUST be of type `CHARACTER`.
out: `INTEGER*4 var_num`	The variable number.

VAR_FILLVALUE_ Inquires the fill value of the current variable (in the current CDF). Required arguments are...

out: `<type> value`	Fill value. `<type>` depends on the data type of the variable. The fill value is read from the CDF and placed into memory at address `value`. This buffer must be large enough to hold the fill value. NOTE: On UNIX systems, if the variable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

VAR_EXTENDRECS_ Inquires the number of extend records for the current variable (in the current CDF). Extend records are described in Section 2.7.4. Required arguments are...

out: `INTEGER*4 n_extend_recs`	Number of records to extend.

VAR_MAXREC_ Inquires the maximum record number for the current variable (in the current CDF). This will always be less than or equal to the maximum record number in the CDF. For variables with a record variance of `NOVARY`, this will be at most one (1). A value of zero (0) indicates that no records have been written to the variable. Required arguments are...

out: `INTEGER*4 var_max_rec`	Maximum record number for the variable.

VAR_DATA_ Reads a value from the current variable (in the current CDF). The value is read at the current record number and current indices (for the current CDF). Required arguments are...

out: `<type> value`	Value. `<type>` depends on the data type of the variable. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

NOTE: On UNIX systems, if the variable has a character string data type, then <type> MUST be CHARACTER to ensure the proper operation of CDF_lib.

VAR_HYPERDATA_ Reads one or more values from the current variable (in the current CDF). The values are read based on the current record number, count, and interval and the current dimension indices, counts, and intervals (for the current CDF). Required arguments are...

`out: <type> buffer` Value(s). <type> depends on the data type of the variable. The values are read from the CDF and placed into memory starting at address buffer. This buffer must be large enough to hold the values.

NOTE: On UNIX systems, if the variable has a character string data type, then <type> MUST be CHARACTER to ensure the proper operation of CDF_lib.

VAR_SEQDATA_ Reads one value from the current variable (in the current CDF) at the current sequential position for that variable. After the read, the sequential position is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the sequential position is past the last record for the variable. Required arguments are...

`out: <type> value` Value. <type> depends on the data type of the variable. The value is read from the CDF and placed into memory at address value. This buffer must be large enough to hold the value.

NOTE: On UNIX systems, if the variable has a character string data type, then <type> MUST be CHARACTER to ensure the proper operation of CDF_lib.

ATTR_SCOPE_ Inquires the scope of the current attribute (in the current CDF). Required arguments are...

`out: INTEGER*4 scope` Attribute scope. The scopes are described in Section 5.5.7.

ATTR_NAME_ Inquires the name of the current attribute (in the current CDF). Required arguments are...

`out: CHARACTER attr_name*(*)`	Attribute name. This character string must be large enough to hold `CDF_ATTR_NAME_LEN` characters. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `attr_name` MUST be of type `CHARACTER`.

`ATTR_NUMBER_` Gets the number of the named attribute (in the current CDF). Required arguments are...

`in: CHARACTER attr_name*(*)`	Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `attr_name` MUST be of type `CHARACTER`.
`out: INTEGER*4 attr_num`	The attribute number.

`ATTR_MAXENTRY_` Inquires the maximum entry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of entries for the attribute. Required arguments are...

`out: INTEGER*4 max_entry`	The maximum entry number for the attribute.

`ATTR_NUMENTRIES_` Inquires the number of entries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum entry number used. Required arguments are...

`out: INTEGER*4 num_entries`	The number of entries for the attribute.

`ENTRY_DATATYPE_` Inquires the data type for the current entry number for the current attribute (in the current CDF). Required arguments are...

`out: INTEGER*4 data_type`	Entry data type. The data types are described in Section 5.5.3.

`ENTRY_NUMELEMS_` Inquires the number of elements (of the data type) for the current entry number for the current attribute (in the current CDF). Required arguments are...

`out: INTEGER*4 num_elements`	Number of elements of the data type. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string. For all other data types, this is the number of elements in an array of that data type.

ENTRY_DATA_

Reads the entry from the current attribute at the current entry number (in the current CDF). Required arguments are...

`out: <type> value`	Value. `<type>` depends on the data type of the variable. The value is read from the CDF and placed into memory at address `value`. This buffer must be large enought to hold the value. NOTE: On UNIX systems, if the entry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

STATUS_TEXT_

Inquires the explanation text for the current status code. Note that the current status code is NOT the status code from the last operation performed. Required arguments are...

`out: CHARACTER text*(*)`	Text explaining the status code. This string must be large enough to hold `CDF_STATUSTEXT_LEN` characters. NOTE: On UNIX systems, for the proper operation of `CDF_lib`, `text` MUST be of type `CHARACTER`.

More Examples

Several more examples of the use of `CDF_lib` follow.

Variable creation In this example a variable will be created with a fill value being specified, initial records will be written, and the number of records to extend the variable when necessary being specified. Note that the fill value was specified before the initial records. This results in the specified fill value being written. Had the fill value not been specified first, the initial records would have been written with the default fill value.

```
.
.
INCLUDE '<path>cdf.h'
.
.
INTEGER*4 id                        ! CDF identifier (handle).
INTEGER*4 status                    ! Status returned from CDF library.
INTEGER*4 dim_varys(2)              ! Dimension variances.
INTEGER*4 var_num                   ! Variable number.
REAL*4    fill_value                ! Fill value.
```

```
DATA fill_value/-999.9/
.
.
dim_varys(1) = VARY
dim_varys(2) = VARY
CALL CDF_lib (SELECT_, CDF_, id,
0             CREATE_, VAR_, 'HUMIDITY', CDF_REAL4, 1, VARY,
1                             dim_varys, var_num,
2             PUT_, VAR_FILLVALUE_, fill_value,
3                   VAR_INITIALRECS_, 500,
4                   VAR_EXTENDRECS_, 50,
5             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Variable creation (character string variable) In this example a character string variable will be created with a fill value being specified.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 id                          ! CDF identifier (handle).
INTEGER*4 status                      ! Status returned from CDF library.
INTEGER*4 dim_varys(1)                ! Dimension variances.
INTEGER*4 var_num                     ! Variable number.
CHARACTER fill_value                  ! Fill value.

DATA fill_value/'**********'/
.
.
dim_varys(0) = VARY
CALL CDF_lib (SELECT_, CDF_, id,
0             CREATE_, VAR_, 'Station', CDF_CHAR, 10, NOVARY,
1                             dim_varys, var_num,
2             PUT_, VAR_FILLVALUE_, fill_value,
3             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Hyper read with subsampling In this example a variable will be subsampled in a 2-dimensional CDF having dimension sizes `100,200`. The CDF is column major and the data type of the variable is `CDF_INT2`.

```
.
.
```

```
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 id                   ! CDF identifier (handle).
INTEGER*4 status               ! Status returned from CDF library.
INTEGER*2 values(50,100)       ! Buffer to receive values.
INTEGER*4 rec_count            ! Record count, one record per hyper get.
INTEGER*4 rec_interval         ! Record interval, set to one to indicate
                               ! contiguous records (really meaningless
                               ! since record count is one).
INTEGER*4 indices(2)           ! Dimension indices, start each read
                               ! at 1,1 of the array.
INTEGER*4 counts(2)            ! Dimension counts, half of the values along
                               ! each dimension will be read.
INTEGER*4 intervals(2)         ! Dimension intervals, every other value
                               ! along each dimension will be read.
INTEGER*4 rec_num              ! Record number.
INTEGER*4 max_rec              ! Maximum record in the CDF - this was
                               ! determined with a call to CDF_inquire.

DATA rec_count/1/, rec_interval/1/, indices/1,1/, counts/50,100/,
1    intervals/2,2/
.
.
CALL CDF_lib (SELECT_, CDF_, id,
1                      VAR_NAME_, 'BRIGHTNESS',
2                      CDF_RECCOUNT_, rec_count,
3                      CDF_RECINTERVAL_, rec_interval,
4                      CDF_DIMINDICES_, indices,
5                      CDF_DIMCOUNTS_, counts,
6                      CDF_DIMINTERVALS_, intervals,
7             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

DO rec_num = 1, max_rec
  CALL CDF_lib (SELECT_, CDF_, id,
  0                      CDF_RECNUMBER_, rec_num,
  1             GET_, VAR_HYPERDATA_, values,
  2             NULL_, status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
  .
  .
  ! process values
  .
  .
END DO
.
.
```

Attribute renaming In this example the attribute named `Tmp` will be renamed `TMP`.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 id                    ! CDF identifier (handle).
INTEGER*4 status                ! Status returned from CDF library.
.
.
CALL CDF_lib (SELECT_, CDF_, id,
0             ATTR_NAME_, 'Tmp',
1             PUT_, ATTR_NAME, 'TMP',
2             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Sequential Access In this example the values for a variable will be averaged. The values will be read using the sequential access method (see Section 2.7.10). Each value in each record will be read and averaged. It is assumed that the data type of the variable has been determined to be `CDF_REAL4`.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 id                    ! CDF identifier (handle).
INTEGER*4 status                ! Status returned from CDF library.
INTEGER*4 var_num               ! Variable number.
INTEGER*4 rec_num               ! Record number, start at first record.
INTEGER*4 indices(2)            ! Dimension indices.
REAL*4    value                 ! Variable value.
REAL*8    sum                   ! Sum of all variable values.
INTEGER*4 count                 ! Number of variable values.
REAL*4    ave                   ! Average value for variable.

DATA indices/1,1/, sum/0.0/, count/0/, rec_num/1/
.
.
CALL CDF_lib (SELECT_, CDF_, id,
0             GET_, VAR_NUMBER_, 'FLUX', var_num,
1             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_lib (SELECT_, CDF_, id,
0                      VAR_, var_num,
1                      VAR_SEQPOS_, rec_num, indices,
2             GET_, VAR_SEQDATA_, value,
3             NULL_, status)
```

```
DO WHILE (status .GE. CDF_OK)
  sum = sum + value
  count = count + 1
  CALL CDF_lib (SELECT_, CDF_, id,
  0             GET_, VAR_SEQDATA_, value,
  1             NULL_, status)
END DO

IF (status .NE. END_OF_VAR) CALL UserStatusHandler (status)

ave = sum / count
.
.
```

Attribute entry writes In this example a set of attribute entries for a particular variable will be written.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 id                        ! CDF identifier (handle)
INTEGER*4 status                    ! Status returned from CDF library.
REAL*4    scale(2)                  ! Scale, minimum/maximum.

DATA scale/-90.0,90.0/
.
.
CALL CDF_lib (SELECT_, CDF_, id,
0                      VAR_NAME_, 'LATITUDE',
1                      ATTR_NAME_, 'FIELDNAM',
2             PUT_, ENTRY_DATA_, CDF_CHAR, 20, 'Latitude            ',
3             SELECT_, ATTR_NAME_, 'SCALE',
4             PUT_, ENTRY_DATA_, CDF_REAL4, 2, scale,
5             SELECT_, ATTR_NAME_, 'UNITS',
6             PUT_, ENTRY_DATA_, CDF_CHAR, 20, 'Degrees north       ',
7             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.7 Interpreting CDF Status Codes

Most CDF functions return a status code of type `INTEGER*4`. The symbolic names for these codes are defined in `cdf.inc` and should be used in your applications rather than using the true numeric values. Appendix B explains each status code. When testing the status code returned from a CDF function, the following rules apply.

`status > CDF_OK`	Indicates successful completion but some additional information is provided. These are informational codes.
`status = CDF_OK`	Indicates successful completion.
`CDF_WARN < status < CDF_OK`	Indicates that the function completed but probably not as expected. These are warning codes.
`status < CDF_WARN`	Indicates that the function did not complete. These are error codes.

The following example shows how you could check the status code returned from CDF functions.

```
INTEGER*4 status
.
.
CALL CDF_function (..., status)   ! any CDF function returning status
IF (status .NE. CDF_OK) THEN
  CALL UserStatusHandler (status, ...)
  .
  .
END IF
```

In your own status handler you can take whatever action is appropriate to the application. An example status handler follows. Note that no action is taken in the status handler if the status is `CDF_OK`.

```
   INCLUDE '<path>cdf.inc'

   SUBROUTINE UserStatusHandler (status)
   INTEGER*4 status

   CHARACTER message*(CDF_ERRTEXT_LEN)

   IF (status .LT. CDF_WARN) THEN
     WRITE (6,10)
10   FORMAT (' ','An error has occurred, halting...')
     CALL CDF_error (status, message)
     WRITE (6,11) message
11   FORMAT (' ',A)
     STOP
   ELSE
     IF (status .LT. CDF_OK) THEN
       WRITE (6,12)
12     FORMAT (' ','Warning, function may not have completed as expected...')
       CALL CDF_error (status, message)
       WRITE (6,13) message
```

```
13    FORMAT (' ',A)
    ELSE
      IF (status .GT. CDF_OK) THEN
        WRITE (6,14)
14      FORMAT (' ','Function completed successfully, but be advised that...')
        CALL CDF_error (status, message)
        WRITE (6,15) message
15      FORMAT (' ',A)
      END IF
    END IF
  END IF

  RETURN
  END
```

Explanations for all CDF status codes are available to your applications through the function `CDF_error`. `CDF_error` writes to a text string an explanation of a given status code.

5.8 CDF_EPOCH Values

Two subroutines exist which compute and breakdown `CDF_EPOCH` values. These subroutines should be used by applications using the `CDF_EPOCH` data type. Section 2.5.4 describes EPOCH values.

5.8.1 compute_EPOCH

`compute_EPOCH` calculates a `CDF_EPOCH` value given the individual components.

```
SUBROUTINE compute_EPOCH (year, month, day, hour, minute, second, msec, EPOCH)

INTEGER*4 year              ! in  -- Year (AD, eg. 1991).
INTEGER*4 month             ! in  -- Month (1-12).
INTEGER*4 day               ! in  -- Day (1-31).
INTEGER*4 hour              ! in  -- Hour (0-23).
INTEGER*4 minute            ! in  -- Minute (0-59).
INTEGER*4 second            ! in  -- Second (0-59).
INTEGER*4 msec              ! in  -- Millisecond (0-999).
REAL*8    EPOCH             ! out -- CDF_EPOCH value.
```

`compute_EPOCH` is a subroutine in the CDF library.

5.8.2 EPOCH_breakdown

`EPOCH_breakdown` breaks down a `CDF_EPOCH` value into the individual components.

```
SUBROUTINE EPOCH_breakdown (EPOCH, year, month, day, hour, minute, second,
1                            msec)

REAL*8    EPOCH            ! in  -- The CDF_EPOCH value.
INTEGER*4 year             ! out -- Year (AD, eg. 1991).
INTEGER*4 month            ! out -- Month (1-12).
INTEGER*4 day              ! out -- Day (1-31).
INTEGER*4 hour             ! out -- Hour (0-23).
INTEGER*4 minute           ! out -- Minute (0-59).
INTEGER*4 second           ! out -- Second (0-59).
INTEGER*4 msec             ! out -- Millisecond (0-999).
```

`EPOCH_breakdown` is a subroutine in the CDF library.

Chapter 6

CDF Release Notes

6.1 Introduction

This chapter should provide you with all of the information you need to know to begin using CDF V2.2 assuming that you are already familar with CDF V2.1. For additional information contact CDF User Support.

6.2 Supported Systems

CDF V2.2 is currently supported on the following machines/operating systems.

1. VAX (VMS)
2. Sun (SunOS)
3. DECstation (ULTRIX)
4. Silicon Graphics Iris & Power Series (IRIX)
5. IBM RS/6000 series (AIX)
6. HP 9000 series (HP-UX)
7. IBM PC (MS-DOS)

Additional ports are planned for the future. They include: Cray (UNICOS) and Macintosh (MacOS).

6.3 Changes for CDF V2.2

Several changes have been made to the CDF distribution for V2.2. CDF V2.2 differs from CDF V2.1 in the following ways.

1. The CDF library was modified to prevent fragmentation within a single-file CDF when possible. A highly fragmented single-file CDF would have slower access times, would required more disk space, and would require more dynamic memory when opened. Single-file CDFs having a large number of records are more likely to be fragmented than those having a small number of records.

   Converting a single-file CDF created with the CDF V2.1 library (or toolkit programs) to a V2.2 CDF with `CDFconvert` will defragment the CDF.

   NOTE: Only CDFs created prior to CDF V2.1.5 are effected. The CDF V2.1.5 library was also modified to prevent fragmentation when possible.

2. The CDF User's Guide is no longer available in separate versions for the operating systems supported by CDF. One version exists with differences between operating systems noted as appropriate.

3. The distribution has been ported to the HP 9000 series (HP-UX) and the IBM PC (MS-DOS). FORTRAN support is not yet available on the IBM PC.

4. The routines to compute and breakdown `CDF_EPOCH` values have been added to the CDF library. The following object modules no longer exist in the same directory as the CDF library but have been included in the CDF library.

   VMS:

Object Module	Routine	Callable from...
`CM_EPOCH.OBJ`	`compute_EPOCH`	FORTRAN
`EPOCH_BK.OBJ`	`EPOCH_breakdown`	FORTRAN
`CMEPOCH.OBJ`	`computeEPOCH`	C
`EPOCHBK.OBJ`	`EPOCHbreakdown`	C

   UNIX:

Object Module	Routine	Callable from...
`cm_epoch.o`	`compute_EPOCH`	FORTRAN
`epoch_bk.o`	`EPOCH_breakdown`	FORTRAN
`cmepoch.o`	`computeEPOCH`	C
`epochbk.o`	`EPOCHbreakdown`	C

5. A message logging option has been added to the `CDFconvert` toolkit program.

6. `CDFcompare` has been modified to allow multiple CDFs to be compared.

7. An option has been added to `CDFstats` to allow the filtering out of fill values when collecting statistics.

8. The `/OUTPUT`,`-output` qualifier of `CDFstats`, `CDFinquire`, and `CDFdump` has been modified to allow a file extension to be specified. If a file extension is not specified, a default file extension is appended.

9. This change affects only the terminology used when discussing the available CDF programming interfaces. What were previously referred to as the C and FORTRAN interfaces are now known as the Standard interface. The Standard interface is callable from both C and FORTRAN applications. The Internal interface still exists and is also callable from both C and FORTRAN applications. So sorry.

10. The CDF library no longer maintains a "current" CDF identifier and "current" CDF status code from one subroutine/function call to the next. They must be selected in each subroutine/function call where they are needed. This affects only the Internal interface.

Appendix A

Skeleton Table & NRV File Formats

A.1 Skeleton Table

A.1.1 Introduction

Skeleton tables are both created by and read by CDF utility programs. `SkeletonTable` creates a skeleton table by reading a CDF. `CDFskeleton` creates a CDF by reading a skeleton table. In almost all cases the format of the skeleton tables read and written will be the same. Any differences are minor and will be described where appropriate.

The skeleton table has a free format (except where noted) — you need not be concerned with any column alignments, spaces between fields, or spaces between successive lines. However, certain syntax rules do apply to skeleton tables.

1. Key words for the header section, global scope attributes section, variable scope attributes section, variables section, and end of file must always be specified (in that order).

2. An exclamation point (`!`) at any point signifies a comment until the end of the line. Any characters encountered after the exclamation point will be ignored. An exclamation point may begin a line (making the entire line a comment). Exclamation points inside delimited character strings are part of the string and do not cause the start of a comment.

3. Attribute and variable names must be delimited. Any character not in the name may be used as the delimiter (except for the exclamation point — an exclamation point may, however, be in the name).

4. All items are referenced from one (1). These include attribute entry numbers and index values.

In the descriptions that follow, optional fields are shown in brackets (`[...]`).

A.1.2 Header Section

The header section contains general information about the CDF. The format of the header section is as follows.

```
#header

                    CDF NAME: <cdf-name>
               DATA ENCODING: <data-encoding>
                    MAJORITY: <variable-majority>
                      FORMAT: <cdf-format>

! Variables G. Attributes V.Attributes  Records    Dims       Sizes
! --------- ------------- ------------  -------    ----    -----------
   <n-vars>  <n-G-attrs>   <n-V-attrs> <n-recs>  <n-dims>  <dim-sizes>
```

The fields are defined as follows.

`<cdf-name>`	The name of the CDF. When `SkeletonTable` creates a skeleton table, this will be the name of the corresponding CDF (not the full pathname specified). When `CDFskeleton` reads a skeleton table, this will be the name of the CDF created unless a CDF pathname is specified on the command line. If the CDF name in the skeleton table is to be used, a full pathname must be specified (if desired) or else the CDF will be created in the default/current directory.
`<data-encoding>`	The data encoding of the CDF. When specifying a data encoding to `CDFskeleton`, the following are valid: `HOST`, `NETWORK`, `VAX`, `SUN`, `MIPSEB`, `MIPSEL`, `IBMRS`, `HP`, and `IBMPC`. When a skeleton table is created by `SkeletonTable`, all of the above encodings with the exception of `HOST` are possible. Data encoding is described in Section 2.4.
`<variable-majority>`	The variable majority of the CDF. This may be either `ROW` or `COLUMN`. Variable majority is described in Section 2.7.8.
`<cdf-format>`	The format of the CDF. This may be either `SINGLE` or `MULTI`. CDF formats are described in Section 2.3. Note that this line is optional. Skeleton tables created by `SkeletonTable` in CDF V2.0 did not have this line because the single-file option did not exist. To allow `CDFskeleton` to read skeleton tables created with `SkeletonTable` in CDF V2.0, this line was made optional. If omitted, `CDFskeleton` will created a multi-file CDF by default. `SkeletonTable` (in CDF V2.1) always generates this line regardless of the version of the CDF being read.
`<n-vars>`	The number of variables in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored. The number of variables created is determined by the number of variable definitions in the variable definitions section.
`<n-G-attrs>`	The number of global scope attributes in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads

	a skeleton table, this value is ignored. The number of global scope attributes created is determined by the number of definitions in the global scope attributes section.
`<n-V-attrs>`	The number of variable scope attributes in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored. The number of variable scope attributes created is determined by the number of definitions in the variable scope attributes section.
`<n-recs>`	The number of records in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored. The number records written to the CDF depends on whether or not any values are specified for NRV (non-record-variant) variables. NRV variables are described in Section 2.7.6.
`<n-dims>`	The number of dimensions in the CDF.
`<dim-sizes>`	The dimension sizes for the CDF — one value per dimension. If the CDF has zero dimensions, this field would be left blank.

An example header section for a 2-dimensional CDF follows.

```
#header

                  CDF NAME: sample2
             DATA ENCODING: NETWORK
                  MAJORITY: ROW
                    FORMAT: SINGLE

! Variables G.Attributes V.Attributes Records  Dims   Sizes
! --------- ------------ ------------ -------  ----  -------
     14          18            4         1       2   180 360
```

If the CDF had zero dimensions, the header section would be as follows.

```
#header

                  CDF NAME: sample0
             DATA ENCODING: NETWORK
                  MAJORITY: ROW
                    FORMAT: SINGLE

! Variables G.Attributes V.Attributes Records  Dims  Sizes
! --------- ------------ ------------ -------  ----  -----
     14          18            4         1       0   ! n/a
```

A.1.3 Global Scope Attributes Section

The global scope attributes section contains the definition of each global scope attribute as well as any entries for those attributes. The format of the global scope attributes section is as follows.

```
#GLOBALattributes

[<global-scope-attribute-definition>
 <global-scope-attribute-definition>
 <global-scope-attribute-definition>
                  .
                  .
                  .
 <global-scope-attribute-definition>]
```

Where `<global-scope-attribute-definition>`, needless to say, is a global scope attribute definition.

Zero or more global scope attribute definitions are allowed (there is no limit on the number of attributes that a CDF may have). The format of each global scope attribute definition is as follows.

```
! Attribute        Entry       Data
! Name             Number      Type           Value
! ---------        ------      ----           -----

  <attr-name>   [<entry-n>:    <data-type>    <value>
                 <entry-n>:   [<data-type>]   <value>
                 <entry-n>:   [<data-type>]   <value>
                     .              .             .
                     .              .             .
                     .              .             .
                 <entry-n>:   [<data-type>]   <value>] .         ! note the "."
```

The fields are defined as follows.

`<attr-name>` — The name of the global scope attribute. The name must be delimited with a character not appearing in the name itself (eg. `"TITLE"` or `'History'`). The delimiting characters are not part of the attribute name in the CDF.

`<entry-n>` — The entry number. More than one entry may be specified for an attribute and there are no restrictions on the entry numbers which may be used (except that they must be greater than zero).

`<data-type>` — The data type for the entry. The data type must be one of the following: `CDF_BYTE`, `CDF_INT1`, `CDF_UINT1`, `CDF_INT2`, `CDF_UINT2`, `CDF_INT4`, `CDF_UINT4`, `CDF_REAL4`, `CDF_FLOAT`, `CDF_REAL8`, `CDF_DOUBLE`, `CDF_EPOCH`, `CDF_CHAR`, or `CDF_UCHAR`. The `<data-type>` field is optional for all but the first entry specified. If omitted, the data type of the previous entry is assumed.

`<value>` The value(s) for the entry. A period (.) follows the value(s) of the last entry for an attribute.

An attribute entry can have more than one element of the specified data type. For character string data types (CDF_CHAR and CDF_UCHAR), each character is an element. The character string must be delimited with a character not appearing in the string itself and the entire delimited string must be enclosed in braces (eg. { "The CDF title." }). If the string will not fit on one line, it may be continued on additional lines. The substrings are each delimited with a unique character and a dash (-) is placed at the end (after the terminating delimiter) of each line except the last one. For example,

```
{ "This is a longer " -
  "CDF title that will" -
  " not fit on one line." }
```

For non-character string data types, the elements are enclosed in braces and separated by commas (eg. { 1, 2, 3 }). If the elements will not all fit on one line, they may be continued on additional lines. For example,

```
{ 1.0, 2.0, 3.0, 4.0, 5.0,
  6.0, 7.0, 8.0, 9.0, 10.0 }
```

Note that an individual element value may not be split across lines.

The format of a value for the CDF_EPOCH data type (which is also considered a non-character string data type) is defined in Section 2.5.4. A CDF_EPOCH value may not be split across two lines.

Example Global Scope Attribute Definitions

Several example global scope attribute definitions follow.

```
#GLOBALattributes

! Attribute         Entry       Data
! Name              Number      Type        Value
! ---------         ------      ----        -----

  "TITLEa"              1:    CDF_CHAR    { "CDAW-9A; SABRE" }.

  .TITLEb.              1:    CDF_CHAR    { "CDAW-9A; SABRE " -
                                            "Backscatter Radar, 20s." }.

  "History"             1:    CDF_CHAR    { "CDF created 02-Jan-1961" }
                        2:                { "CDF modified 23-Oct-1964" }.

  "TIMES"               4:    CDF_EPOCH   { 04-Jul-1976 12:00:00.000,
```

```
                                          31-Oct-1976 00:00:00.000 }
                     6:                 { 25-Dec-1976 01:10:00.000,
                                          01-Jan-1977 01:10:30.000 }.

  -Factors-          1:    CDF_REAL4    { 12.5 }
                     2:                 { 17.4 }
                     3:                 {  8.5 }
                     4:    CDF_INT2     { 7 }
                     5:                 { 12 }.
```

A.1.4 Variable Scope Attributes Section

The variable scope attributes section contains the names of the variable scope attributes in the CDF. Any entries for these attributes are defined in the variables section (following the definition of the corresponding variable). The format of the variable scope attributes section is as follows.

```
#VARIABLEattributes

[<attribute-name>
 <attribute-name>
 <attribute-name>
        .
        .
        .
 <attribute-name>]
```

Where `<attribute-name>` is an attribute name delimited with a character not appearing in the name itself (eg. `"VALIDMIN"` or `'Units'`). The delimiting characters are not part of the attribute name in the CDF. There may be zero or more attribute names (there is no limit on the number of attributes that a CDF may have).

An example variable scope attributes section follows.

```
#VARIABLEattributes

  "FIELDNAM"
  "VALIDMIN"
  "Units"
```

A.1.5 Variables Section

The variables section contains the definition of each variable in the CDF, the definitions for any variable scope attribute entries associated with each variable, and data values for those variables that are non-record-variant (NRV). The format of the variables section is as follows.

```
#variables
```

```
[<variable-definition>
 <variable-definition>
 <variable-definition>
           .
           .
           .
 <variable-definition>]
```

Where <variable-definition> is a variable definition. From zero to CDF_MAX_VARS variable definitions are allowed (CDF_MAX_VARS is defined in Section 2.6). The format of each variable definition is as follows.

```
! Variable            Data         Number    Record     Dimension
! Name                Type        Elements  Variance    Variances
! --------            ----        --------  --------    ---------

  <var-name>     <var-data-type>  <n-elems> <rec-vary>  <dim-varys>

  !  Attribute           Data
  !  Name                Type          Value
  !  ---------           ----          -----

     [<attr-name>  <entry-data-type>    <entry-value>
      <attr-name> [<entry-data-type>]   <entry-value>
      <attr-name> [<entry-data-type>]   <entry-value>
        .               .                 .
        .               .                 .
        .               .                 .
   <attr-name> [<entry-data-type>]   <entry-value>].          ! Note the "."

   [<indices> = <NRV-value>
    <indices> = <NRV-value>
    <indices> = <NRV-value>
        .             .
        .             .
        .             .
    <indices> = <NRV-value>]
```

Each field is defined as follows.

<var-name>	The name of the variable. The name must be delimited with a character not appearing in the name itself (eg. "EPOCH" or 'Temperature'). The delimiting characters are not part of the variable name in the CDF.
<var-data-type>	The data type for the variable. The data type must be one of the following: CDF_BYTE, CDF_INT1, CDF_UINT1, CDF_INT2, CDF_UINT2, CDF_INT4, CDF_UINT4, CDF_REAL4, CDF_FLOAT, CDF_REAL8, CDF_DOUBLE, CDF_EPOCH, CDF_CHAR, or CDF_UCHAR.

`<n-elems>` The number of elements of the data type. For character string data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in each string. For non-character string data types, this value must be one (1).

`<rec-vary>` The record variance of the variable. This must be either T (the values vary from record to record) or F (the values do not vary from record to record).

`<dim-varys>` The dimension variances of the variable. For each dimension in the CDF, there must be either a T (the values vary along that dimension) or F (the values do not vary along that dimension). Each dimension variance must be separated by at least one space. If the CDF has zero dimensions, this field would be left blank.

`<attr-name>` The name of the variable scope attribute for which to specify an entry for this variable. The attribute must have been specified in the variable scope attributes section. The name must be delimited with a character not appearing in the name itself (eg. "SCALEMAX" or 'range'). The delimiting characters are not part of the attribute name in the CDF.

`<entry-data-type>` The data type for the entry. The data type must be one of the following: CDF_BYTE, CDF_INT1, CDF_UINT1, CDF_INT2, CDF_UINT2, CDF_INT4, CDF_UINT4, CDF_REAL4, CDF_FLOAT, CDF_REAL8, CDF_DOUBLE, CDF_EPOCH, CDF_CHAR, or CDF_UCHAR. The `<entry-data-type>` field is optional for all but the first entry specified. If omitted, the data type of the previous entry is assumed.

`<entry-value>` The value(s) for the entry. The format of attribute entry values is described in Section A.1.3.

The last attribute entry MUST be followed by a period (.). If no entries are specified for a variable, the period must still be present (on a separate line).

`<indices>` The indices of an NRV value. The indices are enclosed in brackets and separated by commas (eg. [23, 1] or [1, 80]). If the CDF has zero dimensions, [ ] would be specified (the brackets are still required).

`<NRV-value>` The value at the given indices. For character string data types (CDF_CHAR or CDF_UCHAR), the string must be delimited with a unique character and enclosed in braces ({...}) in the same manner as for an attribute entry for a character string data type. For non-character string data types, the value is not enclosed in braces (the braces are not necessary because there can only be one element). The format for CDF_EPOCH values is described in Section 2.5.4.

NRV values may only be specified for NRV variables (described in Section 2.7.6).

The attribute entries are optional. If omitted, the terminating period is still required (on a separate line).

The NRV values are optional (NRV variables are described in Section 2.7.6). NRV values may also be in a separate file called an NRV file. Section A.2 describes the format of the NRV file.

Several sample variable definition sections for a 2-dimensional CDF follow.

```
! Variable           Data       Number     Record    Dimension
! Name               Type      Elements   Variance   Variances
! --------           ----      --------   --------   ---------

  "Latitude"       CDF_REAL4       1          F          F T

  ! Attribute        Data
  ! Name             Type       Value
  ! --------         ----       -----

    "VALIDMIN"     CDF_REAL4    { -90.0 }
    "VALIDMAX"     CDF_REAL4    { 90.0 }
    "scale"        CDF_REAL4    { -60.0, 60.0 }.

    [ 1, 1 ] = -60.0
    [ 1, 2 ] = -30.0
    [ 1, 3 ] = 0.0
    [ 1, 4 ] = 30.0
    [ 1, 5 ] = 60.0


! Variable           Data       Number     Record    Dimension
! Name               Type      Elements   Variance   Variances
! --------           ----      --------   --------   ---------

  "EPOCH"          CDF_EPOCH       1          T          F F

  ! Attribute        Data
  ! Name             Type       Value
  ! --------         ----       -----

    "scale"        CDF_EPOCH    { 10-Oct-1991 00:00:00.000,
                                  20-Oct-1991 23:59:59.999 }.


! Variable           Data       Number     Record    Dimension
! Name               Type      Elements   Variance   Variances
! --------           ----      --------   --------   ---------

  'Tmp'            CDF_INT2        1          T          T T

  ! Attribute        Data
  ! Name             Type       Value
  ! --------         ----       -----

    'Fieldname'    CDF_CHAR     { "Temperature (C)" }.
```

```
! Variable            Data       Number    Record    Dimension
! Name                Type      Elements  Variance  Variances
! --------            ----      --------  --------  ---------

  "Pres_lvl"        CDF_REAL4      1          F         F F

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    .   ! no attribute entries

    [ 1, 1 ] = 1013.0
```

Several sample variable definition sections for a 0-dimensional CDF follow.

```
! Variable            Data       Number    Record    Dimension
! Name                Type      Elements  Variance  Variances
! --------            ----      --------  --------  ---------

  "Latitude"        CDF_REAL4      1          F ! n/a

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "VALIDMIN"      CDF_REAL4    { -90.0 }
    "VALIDMAX"      CDF_REAL4    { 90.0 }.

    [ ] = -12.3
```

A.1.6 End Section

This section simply consists of the keyword `#end`. This section is required.

A.1.7 Examples

2-Dimensional Skeleton Table

An example 2-dimensional skeleton table follows...

```
! Skeleton table for the "example2" CDF.
! Generated: Fri 31 Jan 1992 14:44:16

#header
```

```
                  CDF NAME: example2
             DATA ENCODING: NETWORK
                  MAJORITY: ROW
                    FORMAT: SINGLE

! Variables G.Attributes V.Attributes Records  Dims  Sizes
! --------- ------------ ------------ -------  ----  -----
      4           1             7         1       2   11 7


#GLOBALattributes

! Attribute         Entry       Data
! Name              Number      Type       Value
! ---------         ------      ----       -----

  "TITLE"               1:    CDF_CHAR     { "Title for example2 CDF." } .


#VARIABLEattributes

  "FIELDNAM"
  "VALIDMIN"
  "VALIDMAX"
  "SCALEMIN"
  "SCALEMAX"
  "UNITS"
  "FORMAT"


#variables

! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "EPOCH"           CDF_EPOCH      1          T         F F

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "FIELDNAM"      CDF_CHAR     { "Time since 0 A.D.   " }
    "VALIDMIN"      CDF_EPOCH    { 01-Jan-0000 00:00:00.000 }
    "VALIDMAX"      CDF_EPOCH    { 01-Jan-2089 00:00:00.000 }
    "SCALEMIN"      CDF_EPOCH    { 01-Apr-1986 07:00:00.000 }
    "SCALEMAX"      CDF_EPOCH    { 01-Apr-1986 23:00:00.000 }
    "UNITS"         CDF_CHAR     { "milliseconds (UT)   " }
    "FORMAT"        CDF_CHAR     { "E14.0   " } .
```

```
! Variable           Data       Number     Record    Dimension
! Name               Type      Elements   Variance   Variances
! --------           ----      --------   --------   ---------

  "LONGITUD"       CDF_REAL4      1          F         T F

  ! Attribute        Data
  ! Name             Type       Value
  ! --------         ----       -----

    "FIELDNAM"     CDF_CHAR     { "Longitude variable  " }
    "VALIDMIN"     CDF_REAL4    { 0.0 }
    "VALIDMAX"     CDF_REAL4    { 180.0 }
    "SCALEMIN"     CDF_REAL4    { -50.0 }
    "SCALEMAX"     CDF_REAL4    { 50.0 }
    "UNITS"        CDF_CHAR     { "Degrees             " }
    "FORMAT"       CDF_CHAR     { "F8.3    " } .

    [ 1, 1 ] = -50.0
    [ 2, 1 ] = -40.0
    [ 3, 1 ] = -30.0
    [ 4, 1 ] = -20.0
    [ 5, 1 ] = -10.0
    [ 6, 1 ] = 0.0
    [ 7, 1 ] = 10.0
    [ 8, 1 ] = 20.0
    [ 9, 1 ] = 30.0
    [ 10, 1 ] = 40.0
    [ 11, 1 ] = 50.0

! Variable           Data       Number     Record    Dimension
! Name               Type      Elements   Variance   Variances
! --------           ----      --------   --------   ---------

  "LATITUDE"       CDF_REAL4      1          F         F T

  ! Attribute        Data
  ! Name             Type       Value
  ! --------         ----       -----

    "FIELDNAM"     CDF_CHAR     { "Latitude variable   " }
    "VALIDMIN"     CDF_REAL4    { 0.0 }
    "VALIDMAX"     CDF_REAL4    { 90.0 }
    "SCALEMIN"     CDF_REAL4    { -30.0 }
    "SCALEMAX"     CDF_REAL4    { 30.0 }
    "UNITS"        CDF_CHAR     { "Degrees             " }
    "FORMAT"       CDF_CHAR     { "F8.3    " } .

    [ 1, 1 ] = -30.0
    [ 1, 2 ] = -20.0
    [ 1, 3 ] = -10.0
```

```
    [ 1, 4 ] = 0.0
    [ 1, 5 ] = 10.0
    [ 1, 6 ] = 20.0
    [ 1, 7 ] = 30.0

! Variable           Data      Number     Record    Dimension
! Name               Type      Elements   Variance  Variances
! --------           ----      --------   --------  ---------

  "TEMPERATURE"    CDF_INT4       1          T         T T

  ! Attribute        Data
  ! Name             Type       Value
  ! --------         ----       -----

    "FIELDNAM"     CDF_CHAR     { "Temperature         " }
    "VALIDMIN"     CDF_INT4     { 0 }
    "VALIDMAX"     CDF_INT4     { 50 }
    "SCALEMIN"     CDF_INT4     { 0 }
    "SCALEMAX"     CDF_INT4     { 10 }
    "UNITS"        CDF_CHAR     { "Deg C               " }
    "FORMAT"       CDF_CHAR     { "I2      " } .


#end
```

0-Dimensional Skeleton Table

An example of a 0-dimensional skeleton table follows...

```
! Skeleton table for the "example0" CDF.
! Generated: Fri 31 Jan 1992 14:45:46

#header

                  CDF NAME: example0
             DATA ENCODING: NETWORK
                  MAJORITY: ROW
                    FORMAT: SINGLE

! Variables G.Attributes V.Attributes Records  Dims  Sizes
! --------- ------------ ------------ -------  ----  -----
      2           1            7         1       0   ! n/a


#GLOBALattributes

! Attribute          Entry       Data
! Name               Number      Type       Value
```

```
! ---------          ------      ----       -----

  "TITLE"               1:    CDF_CHAR     { "Title for example0 CDF." } .


#VARIABLEattributes

  "FIELDNAM"
  "VALIDMIN"
  "VALIDMAX"
  "SCALEMIN"
  "SCALEMAX"
  "UNITS"
  "FORMAT"


#variables

! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "EPOCH"           CDF_EPOCH      1          T         ! n/a

   ! Attribute        Data
   ! Name             Type       Value
   ! --------         ----       -----

    "FIELDNAM"      CDF_CHAR     { "Time since 0 A.D.   " }
    "VALIDMIN"      CDF_EPOCH    { 01-Jan-0000 00:00:00.000 }
    "VALIDMAX"      CDF_EPOCH    { 01-Jan-2089 00:00:00.000 }
    "SCALEMIN"      CDF_EPOCH    { 01-Apr-1986 07:00:00.000 }
    "SCALEMAX"      CDF_EPOCH    { 01-Apr-1986 23:00:00.000 }
    "UNITS"         CDF_CHAR     { "milliseconds (UT)   " }
    "FORMAT"        CDF_CHAR     { "E14.0   " } .

! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "TEMPERATURE"     CDF_INT4       1          T         ! n/a

   ! Attribute        Data
   ! Name             Type       Value
   ! --------         ----       -----

    "FIELDNAM"      CDF_CHAR     { "Temperature         " }
    "VALIDMIN"      CDF_INT4     { 0 }
    "VALIDMAX"      CDF_INT4     { 50 }
    "SCALEMIN"      CDF_INT4     { 0 }
    "SCALEMAX"      CDF_INT4     { 10 }
```

```
    "UNITS"         CDF_CHAR     { "Deg C               " }
    "FORMAT"        CDF_CHAR     { "I2      " } .


#end
```

A.2 NRV File

A.2.1 Introduction

NRV data may be placed either in a CDF's skeleton table or in its NRV file (or neither). When `CDFskeleton` reads a skeleton table and finds no NRV data values for an NRV variable, it checks to see if an NRV file exists. If the NRV file exists, `CDFskeleton` reads it to see if NRV values have been specified for the particular variable. Otherwise, no values are written to the variable.

An NRV file always has a corresponding skeleton table (in the same directory) The name of the NRV file is the same as that of the skeleton table with only the extensions being different. The variables specified in the NRV file must exist in the corresponding skeleton table.

The NRV file has a free format (except where noted) — you need not be concerned with any column alignments, spaces between fields, or spaces between successive lines. However, certain syntax rules do apply to NRV files.

1. An exclamation point (`!`) at any point signifies a comment until the end of the line. Any characters encountered after the exclamation point will be ignored. An exclamation point may begin a line (making the entire line a comment). Exclamation points inside delimited character strings are part of the string and do not cause the start of a comment.
2. Variable names must be delimited. Any character not in the name may be used as the delimiter (except for the exclamation point — an exclamation point may, however, be in the name).
3. Index values are referenced from one (1).

The format of the NRV file is as follows.

```
 <NRV-variable-values-definition>
[<NRV-variable-values-definition>
 <NRV-variable-values-definition>
                 .
                 .
                 .
 <NRV-variable-values-definition>]
```

Where `<NRV-variable-values-definition>` is an NRV variable values definition for the corresponding variable in the corresponding skeleton table. The format of an NRV variable values definition is as follows.

```
<var-name>

  <indices> = <NRV-value>
 [<indices> = <NRV-value>
  <indices> = <NRV-value>
      .             .
      .             .
      .             .
  <indices> = <NRV-value>]
```

Each field is defined as follows.

`<var-name>`	The name of the variable. The name must be delimited with a character not appearing in the name itself (eg. `"EPOCH"` or `'Temperature'`). The delimiting characters are not part of the variable name in the CDF. This variable must exist in the corresponding skeleton table.
`<indices>`	The indices of an NRV value. The indices are enclosed in brackets and separated by commas (eg. `[ 23, 1 ]` or `[ 1, 80 ]`). If the CDF has zero dimensions, a set of empty brackets would be specified (eg. `[ ]`).
`<NRV-value>`	The value at the given indices. For character string data types (`CDF_CHAR` or `CDF_UCHAR`), the string must be delimited with a unique character and enclosed in braces (`{...}`) in the same manner as for an attribute entry for a character string data type. For non-character string data types, the value is not enclosed in braces (the braces are not necessary because there can only be one element). The format for `CDF_EPOCH` values is defined in Section 2.5.4.

A sample NRV file for a 2-dimensional CDF follows.

```
"Latitude"

   [ 1, 1 ] = -90.0
   [ 1, 2 ] = -45.0
   [ 1, 3 ] = 0.0
   [ 1, 4 ] = 45.0
   [ 1, 5 ] = 90.0


'Longitude'

   [ 1, 1 ] = -180.0
   [ 2, 1 ] = -135.0
   [ 3, 1 ] = -90.0
   [ 4, 1 ] = -45.0
   [ 5, 1 ] = 0.0
   [ 6, 1 ] = 45.0
   [ 7, 1 ] = 90.0
```

```
    [ 8, 1 ] = 135.0
    [ 9, 1 ] = 180.0
```

A sample NRV file for a 0-dimensional CDF follows.

```
"DETECTOR"

    [ ] = 8


'ANGLE'

    [ ] = 55.3
```

Appendix B

CDF Status Codes

B.1 Introduction

A status code is returned from most CDF functions. The `cdf.h` (for C) and `CDF.INC` (for FORTRAN) include files contain the numerical values for each of the status codes (and for any other constants referred to in the explanations). The CDF library functions `CDFerror` (for C) and `CDF_error` (for FORTRAN) can be used within a program to inquire the explanation text for a given status code.

B.2 Success Code and Message

`CDF_OK`	Function completed successfully.

B.3 Informational Codes and Messages

Informational codes indicate success but provide some additional information.

`READ_ONLY_ACCESS`	Read-only access is allowed on the CDF (no writing or deleting). Your application does not have write access to the CDF file(s).
`VERSION_1_CDF`	This is a Version 1 CDF, some restrictions may apply. The restrictions are described in Section 2.2.2.
`VIRTUAL_RECORD_DATA`	One or more of the records are virtual (never PUT to the CDF). Virtual records do not physically exist in the CDF file(s) but are part of the conceptual view of the data provided by the CDF library. Virtual records are described in Section 2.7.4.

B.4 Warning Codes and Messages

Warning codes are considered non-normal and indicate a potential problem that may lead to errors in program operation or errors of data interpretation.

ATTR_NAME_TRUNC	Attribute name truncated to CDF_ATTR_NAME_LEN characters. The attribute was created but with a truncated name.
CDF_INCOMPATIBLE	Certain operations on Version 2 CDFs are incompatible with the Obsolete interface (VMS only). These deal mainly with attributes having different data types and/or number of elements per entry. The incompatibilities are described in Section 2.1.3.
CDF_NAME_TRUNC	CDF path name truncated to CDF_PATHNAME_LEN characters. The CDF was created but with a truncated name.
NO_FILLVALUE_SPECIFIED	A fill value has not yet been specified. The default fill value is currently being used for the variable.
NO_VARS_IN_CDF	This CDF contains no variables. The operation performed is not applicable to a CDF with no variables.
SINGLE_FILE_FORMAT	The specified operation is not applicable to CDFs with the single-file format. CDF formats are described in Section 2.3.
VAR_ALREADY_CLOSED	The specified variable is already closed.
VAR_NAME_TRUNC	Variable name truncated to CDF_VAR_NAME_LEN characters. The variable was created but with a truncated name.

B.5 Error Codes and Messages

These codes are errors. The application should take corrective action or halt. Note that some errors may not be fatal to an application.

ATTR_EXISTS	Named attribute already exists - cannot create it. Each attribute in a CDF must have a unique name. Note that trailing blanks are ignored by the CDF library when comparing attribute names.
BAD_ATTR_NAME	Illegal attribute name specified. Attribute names must contain at least one character and each character must be printable.
BAD_ATTR_NUM	Illegal attribute number specified. Attribute numbers must be zero or greater for C applications and one or greater for FORTRAN applications.
BAD_CDF_ID	CDF identifier is unknown or invalid. The CDF identifier specified is not for a currently open CDF.
BAD_CDF_NAME	Illegal CDF name specified. CDF names must contain at least one character and each character must be printable.

`BAD_CDFSTATUS`	Unknown CDF status value received. The status code specified is not used by the CDF library.
`BAD_DATA_TYPE`	Unknown data type specified.
`BAD_DIM_COUNT`	Illegal dimension count specified.
`BAD_DIM_INDEX`	One or more dimension index is out of range. A valid value must be specified regardless of the dimension variance. Note also that the combination of dimension index, count, and interval must not specify an element beyond the end of the dimension.
`BAD_DIM_INTERVAL`	Illegal dimension interval specified.
`BAD_DIM_SIZE`	One or more dimension size specified as zero or less.
`BAD_ENCODING`	Unknown or unsupported data encoding.
`BAD_ENTRY_NUM`	Illegal attribute entry number specified.
`BAD_EXTEND_RECS`	An illegal number of extend records has been specified. The number of extend records must be at least one (1).
`BAD_FNC_OR_ITEM`	The specified function or item is illegal. Check that the proper number of arguments are specified for each operation being performed. Also make sure that `NULL_` is specified as the last operation.
`BAD_FORMAT`	Unknown CDF format specified.
`BAD_INITIAL_RECS`	An illegal number of records to initially write has been specified. The number of initial records must be at least one (1).
`BAD_MAJORITY`	Unknown majority specified.
`BAD_MALLOC`	Unable to allocate dynamic memory - system limit reached. Contact CDF User Support.
`BAD_NUM_DIMS`	The number of dimensions specified is out of the allowed range. Zero (0) through `CDF_MAX_DIMS` dimensions are allowed. If more are needed, contact CDF User Support.
`BAD_NUM_ELEMS`	The number of elements of the data type is illegal. The number of elements must be at least one (1). For non-character string variables, the number of elements must always be one (1).
`BAD_REC_COUNT`	Illegal record count specified.
`BAD_REC_INTERVAL`	Illegal record interval specified.
`BAD_REC_NUM`	Record number is out of range. When reading a variable, the record number must not extend beyond the number of records in the entire CDF. For hyper reads, the combination of record number, count, and interval must not extend beyond the last record in the CDF. Note that a valid value must be specified regardless of the record variance.

`BAD_SCOPE`	Unrecognized scope for attribute creation.
`BAD_VAR_NAME`	Illegal variable name specified. Variable names must contain at least one character and each character must be printable.
`BAD_VAR_NUM`	Illegal variable number specified.
`CANNOT_CHANGE`	Because of dependencies on the value, it cannot be changed.
`CDF_CLOSE_ERROR`	Error detected while trying to close CDF.
`CDF_CREATE_ERROR`	Cannot create CDF specified — error from file system. Make sure sure that sufficient privilege exists to create the CDF file(s) in the disk/directory location specified.
`CDF_EXISTS`	The CDF named already exists - cannot create it. The CDF library will not overwrite an existing CDF.
`CDF_OPEN_ERROR`	Cannot open CDF specified — error from file system. Check that the CDF file(s) are not corrupted and that sufficient privilege exists to open the CDF file(s).
`CDH_OPEN_ERROR`	An error occurred while opening the `.cdh` file (V1 CDF). Check that all of the CDF file(s) exist in the disk/directory specified and that sufficient privilege exists to access them.
`CDH_CLOSE_ERROR`	An error occurred while closing the `.cdh` file (V1 CDF).
`CORRUPTED_V1_CDF`	This Version 1 CDF is corrupted. An internal error has been detected in the CDF's control information. If the CDF file(s) are known to be uncorrupted, please contact CDF User Support.
`CORRUPTED_V2_CDF`	This Version 2 CDF is corrupted. An internal error has been detected in the CDF's control information. If the CDF file(s) are known to be uncorrupted, please contact CDF User Support.
`END_OF_VAR`	The value offset (for sequential access) is at the end of the variable. Reading beyond the end of the last physical record for a variable is not allowed.
`HP_ENCODING_UNSUPPORTED`	HP encoding is not supported on the host machine. The CDF may only be read on an HP 9000 series computer.
`IBM_PC_OVERFLOW`	An operation involving a buffer greater than 64k bytes in size has been specified.
`IBMPC_ENCODING_UNSUPPORTED`	IBMPC encoding is not supported on the host machine. The CDF may only be read on an IBM PC.
`IBMRS_ENCODING_UNSUPPORTED`	IBMRS encoding is not supported on the host machine. The CDF may only be read on an IBM RS/6000 series computer.
`ILLEGAL_ON_V1_CDF`	The specified operation is not allowed on Version 1 CDFs. Section 2.2.2 describes the restrictions when accessing Version 1 CDFs.

`MIPSEB_ENCODING_UNSUPPORTED`	MIPSEB encoding is not supported on the host machine. The CDF may only be read on a Silicon Graphics Iris or Power Series computer.
`MIPSEL_ENCODING_UNSUPPORTED`	MIPSEL encoding is not supported on the host machine. The CDF may only be read on a DECstation computer.
`NO_ATTR_SELECTED`	An attribute has not yet been selected. First specify the attribute on which to perform the operation.
`NO_CDF_SELECTED`	A CDF has not yet been selected. First specify the CDF on which to perform the operation.
`NO_ENTRY_SELECTED`	An attribute entry has not yet been selected. First specify the entry number on which to perform the operation.
`NO_DELETE_ACCESS`	Deleting is not allowed (read-only access). Make sure that delete access is allowed on the CDF file(s).
`NO_RECORDS_EXIST`	No records exist in the CDF.
`NO_STATUS_SELECTED`	A CDF status code has not yet been selected. First specify the status code on which to perform the operation.
`NO_SUCH_ATTR`	Attribute not found in this CDF.
`NO_SUCH_CDF`	The specified CDF does not exist.
`NO_SUCH_ENTRY`	No such entry for specified attribute.
`NO_SUCH_VAR`	Variable not found in this CDF.
`NO_VAR_SELECTED`	A variable has not yet been selected. First specify the variable on which to perform the operation.
`NO_WRITE_ACCESS`	Writing is not allowed (read-only access). Make sure that write access is allowed on the CDF file(s).
`NOT_A_CDF`	Named CDF is corrupted or not actually a CDF. The identifying field in the `.cdf` file is not known to the CDF library. Check to see if the CDF is corrupted. On VMS systems, also make sure that the file attributes indicate fixed-length, 512-byte records.
`SUN_ENCODING_UNSUPPORTED`	Sun encoding is not supported on the host machine. The CDF may only be read on a Sun computer.
`TOO_MANY_CDFS`	A limited number of CDFs may be opened at once. If this limit is reached, contact CDF User Support.
`TOO_MANY_VARS`	A CDF may contain only a limited number of variables. If this limit is reached, contact CDF User Support.
`VAR_CLOSE_ERROR`	Error detected while trying to close variable file.
`VAR_CREATE_ERROR`	An error occurred while creating a variable file (multi-file CDF).
`VAR_EXISTS`	Named variable already exists - cannot create it. Each variable in a CDF must have a unique name. Note that trailing

	blanks are ignored by the CDF library when comparing variable names.
VAR_OPEN_ERROR	An error occurred while opening variable file. Check that sufficient privilege exists to open or create the variable file. For multi-file CDFs, also make sure that the associated variable files exist.
VAR_READ_ERROR	Failed to read variable as requested (file read error).
VAR_WRITE_ERROR	Failed to write variable as requested (file write error).
VAX_ENCODING_UNSUPPORTED	VAX encoding is not supported on the host machine. The CDF may only be read on a VAX computer.
V1_CDF_UNSUPPORTED	Version 1 CDFs are not supported on the host machine. Version 1 CDFs may only be read on VAX computers.

Appendix C

C Programming Summary

C.1 Introduction

Below is a summary of the CDF library functions callable from C applications. Each is written as a function prototype in the ANSI style.

C.2 Standard interface

```
CDFstatus CDFcreate(
char  *CDFname,                                                    /* in  */
long  numDims,                                                     /* in  */
long  dimSizes[],                                                  /* in  */
long  encoding,                                                    /* in  */
long  majority,                                                    /* in  */
CDFid *id);                                                        /* out */

CDFstatus CDFopen(
char  *CDFname,                                                    /* in  */
CDFid *id);                                                        /* out */

CDFstatus CDFdoc(
CDFid id,                                                          /* in  */
long  *version,                                                    /* out */
long  *release,                                                    /* out */
char  text[CDF_DOCUMENT_LEN+1]);                                   /* out */

CDFstatus CDFinquire(
CDFid id,                                                          /* in  */
long  *numDims,                                                    /* out */
long  dimSizes[CDF_MAX_DIMS],                                      /* out */
long  *encoding,                                                   /* out */
```

```
long  *majority,                                                    /* out */
long  *maxRec,                                                      /* out */
long  *numVars,                                                     /* out */
long  *numAttrs);                                                   /* out */

CDFstatus CDFclose(
CDFid id);                                                          /* in  */

CDFstatus CDFdelete(
CDFid id);                                                          /* in  */

CDFstatus CDFerror(
CDFstatus status,                                                   /* in  */
char message[CDF_STATUSTEXT_LEN+1]);                                /* out */

CDFstatus CDFattrCreate(
CDFid id,                                                           /* in  */
char  *attrName,                                                    /* in  */
long  attrScope,                                                    /* in  */
long  *attrNum);                                                    /* out */

long CDFattrNum(
CDFid id,                                                           /* in  */
char  *attrName);                                                   /* in  */

CDFstatus CDFattrRename(
CDFid id,                                                           /* in  */
long  attrNum,                                                      /* in  */
char  *attrName);                                                   /* in  */

CDFstatus CDFattrInquire(
CDFid id,                                                           /* in  */
long  attrNum,                                                      /* in  */
char  *attrName,                                                    /* out */
long  *attrScope,                                                   /* out */
long  *maxEntry);                                                   /* out */

CDFstatus CDFattrEntryInquire(
CDFid id,                                                           /* in  */
long  attrNum,                                                      /* in  */
long  entryNum,                                                     /* in  */
long  *dataType,                                                    /* out */
long  *numElements);                                                /* out */

CDFstatus CDFattrPut(
CDFid id,                                                           /* in  */
long  attrNum,                                                      /* in  */
long  entryNum,                                                     /* in  */
long  dataType,                                                     /* in  */
long  numElements,                                                  /* in  */
void  *value);                                                      /* in  */
```

```
CDFstatus CDFattrGet(
CDFid id,                                                   /* in  */
long  attrNum,                                              /* in  */
long  entryNum,                                             /* in  */
void  *value);                                              /* out */

CDFstatus CDFattrCorrectScope(
CDFid id,                                                   /* in  */
long  attrNum,                                              /* in  */
long  attrScope)                                            /* in  */

CDFstatus CDFvarCreate(
CDFid id,                                                   /* in  */
char  *varName,                                             /* in  */
long  dataType,                                             /* in  */
long  numElements,                                          /* in  */
long  recVariance,                                          /* in  */
long  dimVariances[],                                       /* in  */
long  *varNum);                                             /* out */

long CDFvarNum(
CDFid id,                                                   /* in  */
char  *varName);                                            /* in  */

CDFstatus CDFvarRename(
CDFid id,                                                   /* in  */
long  varNum,                                               /* in  */
char  *varName);                                            /* in  */

CDFstatus CDFvarInquire(
CDFid id,                                                   /* in  */
long  varNum,                                               /* in  */
char  *varName,                                             /* out */
long  *dataType,                                            /* out */
long  *numElements,                                         /* out */
long  *recVariance,                                         /* out */
long  dimVariances[CDF_MAX_DIMS]);                          /* out */

CDFstatus CDFvarPut(
CDFid id,                                                   /* in  */
long  varNum,                                               /* in  */
long  recNum,                                               /* in  */
long  indices[],                                            /* in  */
void  *value);                                              /* in  */

CDFstatus CDFvarGet(
CDFid id,                                                   /* in  */
long  varNum,                                               /* in  */
long  recNum,                                               /* in  */
long  indices[],                                            /* in  */
```

```
void  *value);                                                  /* out */

CDFstatus CDFvarHyperPut(
CDFid id,                                                       /* in  */
long  varNum,                                                   /* in  */
long  recStart,                                                 /* in  */
long  recCount,                                                 /* in  */
long  recInterval,                                              /* in  */
long  indices[],                                                /* in  */
long  counts[],                                                 /* in  */
long  intervals[],                                              /* in  */
void  *buffer);                                                 /* in  */

CDFstatus CDFvarHyperGet(
CDFid id,                                                       /* in  */
long  varNum,                                                   /* in  */
long  recStart,                                                 /* in  */
long  recCount,                                                 /* in  */
long  recInterval,                                              /* in  */
long  indices[],                                                /* in  */
long  counts[],                                                 /* in  */
long  intervals[],                                              /* in  */
void  *buffer);                                                 /* out */
```

C.3 Internal interface

```
CDFstatus CDFlib(...);

   CREATE_    CDF_              char *CDFname                   /* in */
                                long numDims                    /* in  */
                                long dimSizes[]                 /* in  */
                                CDFid *id                       /* out */
              VAR_              char *varName                   /* in  */
                                long dataType                   /* in  */
                                long numElements                /* in  */
                                long recVary                    /* in  */
                                long dimVarys                   /* in  */
                                long *varNum                    /* out */
              ATTR_             char *attrName                  /* in  */
                                long scope                      /* in  */
                                long *attrNum                   /* out */

   OPEN_      CDF_              char *CDFname                   /* in  */
                                CDFid *id                       /* out */

   DELETE_    CDF_

   CLOSE_     CDF_
              VAR_
```

```
SELECT_    CDF_                CDFid id                               /* in  */
           CDF_STATUS_         CDFstatus status                       /* in  */
           VAR_                long varNum                            /* in  */
           VAR_NAME_           char *varName                          /* in  */
           CDF_RECNUMBER_      long recNum                            /* in  */
           CDF_RECCOUNT_       long recCount                          /* in  */
           CDF_RECINTERVAL_    long recInterval                       /* in  */
           CDF_DIMINDICES_     long indices[]                         /* in  */
           CDF_RECCOUNTS_      long counts[]                          /* in  */
           CDF_RECINTERVALS_   long intervals[]                       /* in  */
           VAR_SEQPOS_         long recNum                            /* in  */
                               long indices[]                         /* in  */
           ATTR_               long attrNum                           /* in  */
           ATTR_NAME_          char *attrName                         /* in  */
           ENTRY_              long entryNum                          /* in  */

CONFIRM_   CDF_                CDFid *id                              /* out */
           CDF_STATUS_         CDFstatus *status                      /* out */
           CDF_NAME_           char *CDFname                          /* out */
           VAR_                long *varNum                           /* out */
           CDF_RECNUMBER_      long *recNum                           /* out */
           CDF_RECCOUNT_       long *recCount                         /* out */
           CDF_RECINTERVAL_    long *recInterval                      /* out */
           CDF_DIMINDICES_     long indices[]                         /* out */
           CDF_DIMCOUNTS_      long counts[]                          /* out */
           CDF_DIMINTERVALS_   long intervals[]                       /* out */
           VAR_SEQPOS_         long *recNum                           /* out */
                               long indices[]                         /* out */
           ATTR_               long *attrNum                          /* out */
           ENTRY_              long *entryNum                         /* out */

PUT_       CDF_ARRAY_          long numDims                           /* in  */
                               long dimSizes[]                        /* in  */
           CDF_ENCODING_       long encoding                          /* in  */
           CDF_MAJORITY_       long majority                          /* in  */
           CDF_FORMAT_         long format                            /* in  */
           VAR_NAME_           char *varName                          /* in  */
           VAR_DATASPEC_       long dataType                          /* in  */
                               long numElements                       /* in  */
           VAR_RECVARY_        long recVary                           /* in  */
           VAR_DIMVARYS_       long dimVarys[]                        /* in  */
           VAR_FILLVALUE_      void *value                            /* in  */
           VAR_INITIALRECS_    long NinitialRecs                      /* in  */
           VAR_EXTENDRECS_     long NextendRecs                       /* in  */
           VAR_DATA_           void *value                            /* in  */
           VAR_HYPERDATA_      void *buffer                           /* in  */
           VAR_SEQDATA_        void *value                            /* in  */
           ATTR_SCOPE_         long scope                             /* in  */
           ATTR_NAME_          char *attrName                         /* in  */
           ENTRY_DATA_         long dataType                          /* in  */
```

```
                         long numElements                   /* in  */
                         void *value                        /* in  */

GET_      CDF_NUMDIMS_        long *numDims                 /* out */
          CDF_DIMSIZES_       long dimSizes[]               /* out */
          CDF_ENCODING_       long *encoding                /* out */
          CDF_MAJORITY_       long *majority                /* out */
          CDF_FORMAT_         long *format                  /* out */
          CDF_COPYRIGHT_      char *copyRight               /* out */
          LIB_COPYRIGHT_      char *copyRight               /* out */
          CDF_NUMVARS_        long *numVars                 /* out */
          CDF_NUMATTRS_       long *numAttrs                /* out */
          CDF_MAXREC_         long *maxRec                  /* out */
          CDF_VERSION_        long *version                 /* out */
          CDF_RELEASE_        long *release                 /* out */
          CDF_INCREMENT_      long *increment               /* out */
          LIB_VERSION_        long *version                 /* out */
          LIB_RELEASE_        long *release                 /* out */
          LIB_INCREMENT_      long *increment               /* out */
          LIB_subINCREMENT_   char *subincrement            /* out */
          VAR_NAME_           char *varName                 /* out */
          VAR_DATATYPE_       long *dataType                /* out */
          VAR_NUMELEMS_       long *numElements             /* out */
          VAR_RECVARY_        long *recVary                 /* out */
          VAR_DIMVARYS_       long dimVarys[]               /* out */
          VAR_NUMBER_         char *varName                 /* in  */
                              long *varNum                  /* out */
          VAR_FILLVALUE_      void *value                   /* out */
          VAR_EXTENDRECS_     long *NextendRecs             /* out */
          VAR_MAXREC_         long *maxRec                  /* out */
          VAR_DATA_           void *value                   /* out */
          VAR_HYPERDATA_      void *buffer                  /* out */
          VAR_SEQDATA_        void *value                   /* out */
          ATTR_SCOPE_         long *scope                   /* out */
          ATTR_NAME_          char *attrName                /* out */
          ATTR_NUMBER_        char *attrName                /* in  */
                              long *attrNum                 /* out */
          ATTR_MAXENTRY_      long *maxEntry                /* out */
          ATTR_NUMENTRIES_    long *numEntries              /* out */
          ENTRY_DATATYPE_     long *dataType                /* out */
          ENTRY_NUMELEMS_     long *numElements             /* out */
          ENTRY_DATA_         void *value                   /* out */
          STATUS_TEXT_        char *text                    /* out */
```

Appendix D

FORTRAN Programming Summary

D.1 Introduction

Below is a summary of the CDF library routines callable from FORTRAN applications. Each is written as a function or subroutine declaration.

D.2 Standard interface

```
SUBROUTINE CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority,
1                      id, status)
CHARACTER CDF_name                                                   ! in
INTEGER*4 num_dims                                                   ! in
INTEGER*4 dim_sizes(*)                                               ! in
INTEGER*4 encoding                                                   ! in
INTEGER*4 majority                                                   ! in
INTEGER*4 id                                                         ! out
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_open (CDF_name, id, status)
CHARACTER CDF_name*(*)                                               ! in
INTEGER*4 id                                                         ! out
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_doc (id, version, release, text, status)
INTEGER*4 id                                                         ! in
INTEGER*4 version                                                    ! out
INTEGER*4 release                                                    ! out
CHARACTER text*(CDF_DOCUMENT_LEN)                                    ! out
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
```

```
1                        max_rec, num_vars, num_attrs, status)
INTEGER*4 id                                                       ! in
INTEGER*4 num_dims                                                 ! out
INTEGER*4 dim_sizes(CDF_MAX_DIMS)                                  ! out
INTEGER*4 encoding                                                 ! out
INTEGER*4 majority                                                 ! out
INTEGER*4 max_rec                                                  ! out
INTEGER*4 num_vars                                                 ! out
INTEGER*4 num_attrs                                                ! out
INTEGER*4 status                                                   ! out

SUBROUTINE CDF_close (id, status)
INTEGER*4 id                                                       ! in
INTEGER*4 status                                                   ! out

SUBROUTINE CDF_delete (id, status)
INTEGER*4 id                                                       ! in
INTEGER*4 status                                                   ! out

SUBROUTINE CDF_error (status, text)
INTEGER*4 status                                                   ! in
CHARACTER text*(CDF_STATUSTEXT_LEN)                                ! out

SUBROUTINE CDF_attr_create (id, attr_name, attr_scope, attr_num, status)
INTEGER*4 id                                                       ! in
CHARACTER attr_name*(*)                                            ! in
INTEGER*4 attr_scope                                               ! in
INTEGER*4 attr_num                                                 ! out
INTEGER*4 status                                                   ! out

INTEGER*4 CDF_attr_num (id, attr_name)
INTEGER*4 id                                                       ! in
CHARACTER attr_name*(*)                                            ! in

SUBROUTINE CDF_attr_rename (id, attr_num, attr_name, status)
INTEGER*4 id                                                       ! in
INTEGER*4 attr_num                                                 ! in
CHARACTER attr_name*(*)                                            ! in
INTEGER*4 status                                                   ! out

SUBROUTINE CDF_attr_inquire (id, attr_num, attr_name, attr_scope, max_entry,
1                             status)
INTEGER*4 id                                                       ! in
INTEGER*4 attr_num                                                 ! in
CHARACTER attr_name*(CDF_ATTR_NAME_LEN)                            ! out
INTEGER*4 attr_scope                                               ! out
INTEGER*4 max_entry                                                ! out
INTEGER*4 status                                                   ! out

SUBROUTINE CDF_attr_entry_inquire (id, attr_num, entry_num, data_type,
1                                   num_elements, status)
```

```
INTEGER*4 id                                                        ! in
INTEGER*4 attr_num                                                  ! in
INTEGER*4 entry_num                                                 ! in
INTEGER*4 data_type                                                 ! out
INTEGER*4 num_elements                                              ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_attr_put (id, attr_num, entry_num, data_type, num_elements,
1                         value, status)
INTEGER*4 id                                                        ! in
INTEGER*4 attr_num                                                  ! in
INTEGER*4 entry_num                                                 ! in
INTEGER*4 data_type                                                 ! in
INTEGER*4 num_elements                                              ! in
<type>    value                                                     ! in
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_attr_get (id, attr_num, entry_num, value, status)
INTEGER*4 id                                                        ! in
INTEGER*4 attr_num                                                  ! in
INTEGER*4 entry_num                                                 ! in
<type>    value                                                     ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_attr_correct_scope (id, attr_num, attr_scope, status)
INTEGER*4 id                                                        ! in
INTEGER*4 attr_num                                                  ! in
INTEGER*4 attr_scope                                                ! in
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_var_create (id, var_name, data_type, num_elements,
1                           rec_variance, dim_variances, var_num, status)
INTEGER*4 id                                                        ! in
CHARACTER var_name                                                  ! in
INTEGER*4 data_type                                                 ! in
INTEGER*4 num_elements                                              ! in
INTEGER*4 rec_variance                                              ! in
INTEGER*4 dim_variances(*)                                          ! in
INTEGER*4 var_num                                                   ! out
INTEGER*4 status                                                    ! out

INTEGER*4 FUNCTION CDF_var_num (id, var_name)
INTEGER*4 id                                                        ! in
CHARACTER var_name*(*)                                              ! in

SUBROUTINE CDF_var_rename (id, var_num, var_name, status)
INTEGER*4 id                                                        ! in
INTEGER*4 var_num                                                   ! in
CHARACTER var_name*(*)                                              ! in
INTEGER*4 status                                                    ! out
```

```
SUBROUTINE CDF_var_inquire (id, var_num, var_name, data_type, num_elements,
1                          rec_variance, dim_variances, status)
INTEGER*4 id                                                      ! in
INTEGER*4 var_num                                                 ! in
CHARACTER var_name*(CDF_VAR_NAME_LEN)                             ! out
INTEGER*4 data_type                                               ! out
INTEGER*4 num_elements                                            ! out
INTEGER*4 rec_variance                                            ! out
INTEGER*4 dim_variances(CDF_MAX_DIMS)                             ! out
INTEGER*4 status                                                  ! out

SUBROUTINE CDF_var_put (id, var_num, rec_num, indices, value, status)
INTEGER*4 id                                                      ! in
INTEGER*4 var_num                                                 ! in
INTEGER*4 rec_num                                                 ! in
INTEGER*4 indices(*)                                              ! in
<type>    value                                                   ! in
INTEGER*4 status                                                  ! out

SUBROUTINE CDF_var_get (id, var_num, rec_num, indices, value, status)
INTEGER*4 id                                                      ! in
INTEGER*4 var_num                                                 ! in
INTEGER*4 rec_num                                                 ! in
INTEGER*4 indices(*)                                              ! in
<type>    value                                                   ! out
INTEGER*4 status                                                  ! out

SUBROUTINE CDF_var_hyper_put (id, var_num, rec_start, rec_count, rec_interval,
1                            indices, counts, intervals, buffer, status)
INTEGER*4 id                                                      ! in
INTEGER*4 var_num                                                 ! in
INTEGER*4 rec_start                                               ! in
INTEGER*4 rec_count                                               ! in
INTEGER*4 rec_interval                                            ! in
INTEGER*4 indices(*)                                              ! in
INTEGER*4 counts(*)                                               ! in
INTEGER*4 intervals(*)                                            ! in
<type>    buffer                                                  ! in
INTEGER*4 status                                                  ! out

SUBROUTINE CDF_var_hyper_get (id, var_num, rec_start, rec_count, rec_interval,
1                            indices, counts, intervals, buffer, status)
INTEGER*4 id                                                      ! in
INTEGER*4 var_num                                                 ! in
INTEGER*4 rec_start                                               ! in
INTEGER*4 rec_count                                               ! in
INTEGER*4 rec_interval                                            ! in
INTEGER*4 indices(*)                                              ! in
INTEGER*4 counts(*)                                               ! in
INTEGER*4 intervals(*)                                            ! in
<type>    buffer                                                  ! out
```

```
INTEGER*4 status                                                  ! out
```

D.3 Internal interface

```
INTEGER*4 FUNCTION CDF_lib (..., status)
INTEGER*4 status                                                  ! out

  CREATE_   CDF_               CHARACTER CDF_name*(*)             ! in
                               INTEGER*4 num_dims                 ! in
                               INTEGER*4 dim_sizes(*)             ! in
                               INTEGER*4 id                       ! out
            VAR_               CHARACTER var_name*(*)             ! in
                               INTEGER*4 data_type                ! in
                               INTEGER*4 num_elements             ! in
                               INTEGER*4 rec_vary                 ! in
                               INTEGER*4 dim_varys                ! in
                               INTEGER*4 var_num                  ! out
            ATTR_              CHARACTER attr_name*(*)            ! in
                               INTEGER*4 scope                    ! in
                               INTEGER*4 attr_num                 ! out

  OPEN_     CDF_               CHARACTER CDF_name*(*)             ! in
                               INTEGER*4 id                       ! out

  DELETE_   CDF_

  CLOSE_    CDF_
            VAR_

  SELECT_   CDF_               INTEGER*4 id                       ! in
            CDF_STATUS_        INTEGER*4 status                   ! in
            VAR_               INTEGER*4 var_num                  ! in
            VAR_NAME_          CHARACTER var_name*(*)             ! in
            CDF_RECNUMBER_     INTEGER*4 rec_num                  ! in
            CDF_RECCOUNT_      INTEGER*4 rec_count                ! in
            CDF_RECINTERVAL_   INTEGER*4 rec_interval             ! in
            CDF_DIMINDICES_    INTEGER*4 indices(*)               ! in
            CDF_RECCOUNTS_     INTEGER*4 counts(*)                ! in
            CDF_RECINTERVALS_  INTEGER*4 intervals(*)             ! in
            VAR_SEQPOS_        INTEGER*4 rec_num                  ! in
                               INTEGER*4 indices(*)               ! in
            ATTR_              INTEGER*4 attr_num                 ! in
            ATTR_NAME_         CHARACTER attr_name*(*)            ! in
            ENTRY_             INTEGER*4 entry_num                ! in

  CONFIRM_  CDF_               INTEGER*4 id                       ! out
            CDF_STATUS_        INTEGER*4 status                   ! out
            CDF_NAME_          CHARACTER CDF_name*(*)             ! out
            VAR_               INTEGER*4 var_num                  ! out
```

```
          CDF_RECNUMBER_      INTEGER*4 rec_num                         ! out
          CDF_RECCOUNT_       INTEGER*4 rec_count                       ! out
          CDF_RECINTERVAL_    INTEGER*4 rec_interval                    ! out
          CDF_DIMINDICES_     INTEGER*4 indices(*)                      ! out
          CDF_DIMCOUNTS_      INTEGER*4 counts(*)                       ! out
          CDF_DIMINTERVALS_   INTEGER*4 intervals(*)                    ! out
          VAR_SEQPOS_         INTEGER*4 rec_num                         ! out
                              INTEGER*4 indices(*)                      ! out
          ATTR_               INTEGER*4 attr_num                        ! out
          ENTRY_              INTEGER*4 entry_num                       ! out

PUT_      CDF_ARRAY_          INTEGER*4 num_dims                        ! in
                              INTEGER*4 dim_sizes(*)                    ! in
          CDF_ENCODING_       INTEGER*4 encoding                        ! in
          CDF_MAJORITY_       INTEGER*4 majority                        ! in
          CDF_FORMAT_         INTEGER*4 format                          ! in
          VAR_NAME_           CHARACTER var_name*(*)                    ! in
          VAR_DATASPEC_       INTEGER*4 data_type                       ! in
                              INTEGER*4 num_elements                    ! in
          VAR_RECVARY_        INTEGER*4 rec_vary                        ! in
          VAR_DIMVARYS_       INTEGER*4 dim_varys(*)                    ! in
          VAR_FILLVALUE_      <type> value                              ! in
          VAR_INITIALRECS_    INTEGER*4 n_initial_recs                  ! in
          VAR_EXTENDRECS_     INTEGER*4 _nextend_recs                   ! in
          VAR_DATA_           <type> value                              ! in
          VAR_HYPERDATA_      <type> buffer                             ! in
          VAR_SEQDATA_        <type> value                              ! in
          ATTR_SCOPE_         INTEGER*4 scope                           ! in
          ATTR_NAME_          CHARACTER attr_name*(*)                   ! in
          ENTRY_DATA_         INTEGER*4 data_type                       ! in
                              INTEGER*4 num_elements                    ! in
                              <type> value                              ! in

GET_      CDF_NUMDIMS_        INTEGER*4 num_dims                        ! out
          CDF_DIMSIZES_       INTEGER*4 dim_sizes(*)                    ! out
          CDF_ENCODING_       INTEGER*4 encoding                        ! out
          CDF_MAJORITY_       INTEGER*4 majority                        ! out
          CDF_FORMAT_         INTEGER*4 format                          ! out
          CDF_COPYRIGHT_      CHARACTER copy_right*(*)                  ! out
          LIB_COPYRIGHT_      CHARACTER copy_right*(*)                  ! out
          CDF_NUMVARS_        INTEGER*4 num_vars                        ! out
          CDF_NUMATTRS_       INTEGER*4 num_attrs                       ! out
          CDF_MAXREC_         INTEGER*4 max_rec                         ! out
          CDF_VERSION_        INTEGER*4 version                         ! out
          CDF_RELEASE_        INTEGER*4 release                         ! out
          CDF_INCREMENT_      INTEGER*4 increment                       ! out
          LIB_VERSION_        INTEGER*4 version                         ! out
          LIB_RELEASE_        INTEGER*4 release                         ! out
          LIB_INCREMENT_      INTEGER*4 increment                       ! out
          LIB_subINCREMENT_   CHARACTER subincrement*1                  ! out
          VAR_NAME_           CHARACTER var_name*(*)                    ! out
```

```
VAR_DATATYPE_       INTEGER*4 data_type                    ! out
VAR_NUMELEMS_       INTEGER*4 num_elements                 ! out
VAR_RECVARY_        INTEGER*4 rec_vary                     ! out
VAR_DIMVARYS_       INTEGER*4 dim_varys(*)                 ! out
VAR_NUMBER_         CHARACTER var_name*(*)                 ! in
                    INTEGER*4 var_num                      ! out
VAR_FILLVALUE_      <type> value                           ! out
VAR_EXTENDRECS_     INTEGER*4 n_extend_recs                ! out
VAR_MAXREC_         INTEGER*4 max_rec                      ! out
VAR_DATA_           <type> value                           ! out
VAR_HYPERDATA_      <type> buffer                          ! out
VAR_SEQDATA_        <type> value                           ! out
ATTR_SCOPE_         INTEGER*4 scope                        ! out
ATTR_NAME_          CHARACTER attr_name*(*)                ! out
ATTR_NUMBER_        CHARACTER attr_name*(*)                ! in
                    INTEGER*4 attr_num                     ! out
ATTR_MAXENTRY_      INTEGER*4 max_entry                    ! out
ATTR_NUMENTRIES_    INTEGER*4 num_entries                  ! out
ENTRY_DATATYPE_     INTEGER*4 data_type                    ! out
ENTRY_NUMELEMS_     INTEGER*4 num_elements                 ! out
ENTRY_DATA_         <type> value                           ! out
STATUS_TEXT_        CHARACTER text*(*)                     ! out
```