# CDF
## User's Guide

Version 2.4, February 25, 1994

National Space Science Data Center

Copyright © 1990 — 1994 NASA/GSFC/NSSDC

National Space Science Data Center
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 (U.S.A.)

NSI/DECnet — NSSDCA::CDFSUPPORT
Internet — cdfsupport@nssdca.gsfc.nasa.gov

Permission is granted to make and distribute verbatim copies of this document provided this copyright and permission notice are preserved on all copies.

# Contents

**Preface** **xvii**

**1 CDF Primer** **1**

1.1 Introduction . . . . . 1

1.2 Why Use CDF? . . . . . 1

1.3 Conceptual Organization . . . . . 1

1.4 Features of the CDF Library . . . . . 2

1.4.1 File Format Options . . . . . 2

1.4.2 Data Encoding Options . . . . . 3

1.4.3 Variable Data Access Options . . . . . 6

1.5 Organizing Your Data in a CDF . . . . . 7

1.5.1 Variables . . . . . 7

1.6 Attributes . . . . . 11

1.7 CDF Toolkit . . . . . 12

1.8 Library Interface Routines . . . . . 12

1.8.1 Standard Interface . . . . . 13

1.8.2 Internal Interface . . . . . 13

1.9 Examples . . . . . 14

1.9.1 Creating a CDF, the Hard Way (But Not That Hard) . . . . . 14

1.9.2 Creating a CDF, an Easier Way . . . . . 19

**2 General CDF Concepts** **25**

2.1 CDF Library . . . 25

2.1.1 Interfaces . . . 25

2.1.2 CDF Modes . . . 27

2.1.3 Limits . . . 29

2.1.4 Caching Scheme . . . 29

2.2 CDFs . . . 30

2.2.1 Accessing . . . 30

2.2.2 Creating . . . 31

2.2.3 Opening . . . 31

2.2.4 Closing . . . 31

2.2.5 Naming . . . 31

2.2.6 Format . . . 32

2.2.7 Encoding . . . 33

2.2.8 Decoding . . . 36

2.2.9 Data Types . . . 37

2.2.10 Limits . . . 39

2.3 Variables . . . 40

2.3.1 Accessing . . . 40

2.3.2 Opening . . . 40

2.3.3 Closing . . . 40

2.3.4 Naming . . . 41

2.3.5 Numbering . . . 41

2.3.6 Dimensionality . . . 41

2.3.7 Records . . . 41

2.3.8 Data Specification . . . 44

2.3.9 Record Variance . . . 45

2.3.10 Dimension Variance . . . 45
2.3.11 Majority . . . 46
2.3.12 Single Value Access . . . 47
2.3.13 Hyper Access . . . 48
2.3.14 Sequential Access . . . 50
2.3.15 Multiple Variable Access . . . 51
2.3.16 Variable Pad Values . . . 53
2.4 Attributes . . . 53
2.4.1 Naming . . . 53
2.4.2 Numbering . . . 54
2.4.3 Attribute Scopes . . . 54
2.4.4 Attribute Entries . . . 55
2.5 NSSDC Standard . . . 56
2.5.1 Standard Attributes . . . 56
2.5.2 Standard Variables . . . 57

**3 CDF Toolkit Reference 59**

3.1 Introduction . . . 59
3.2 CDFdir . . . 60
3.2.1 Introduction . . . 60
3.2.2 Executing the CDFdir Program . . . 60
3.2.3 Output from the CDFdir Program . . . 61
3.3 CDFcompare . . . 61
3.3.1 Introduction . . . 61
3.3.2 Executing the CDFcompare Program . . . 61
3.3.3 Output from the CDFcompare Program . . . 64
3.4 CDFconvert . . . 64
3.4.1 Introduction . . . 64

3.4.2 Executing the CDFconvert Program . . . . . 64
3.4.3 Output from the CDFconvert Program . . . . . 67
3.5 CDFscope . . . . . 68
3.5.1 Introduction . . . . . 68
3.5.2 Executing the CDFscope Program . . . . . 68
3.5.3 Output from the CDFscope Program . . . . . 69
3.6 CDFstats . . . . . 69
3.6.1 Introduction . . . . . 69
3.6.2 Executing the CDFstats Program . . . . . 70
3.6.3 Output from the CDFstats Program . . . . . 72
3.7 CDFinquire . . . . . 74
3.7.1 Introduction . . . . . 74
3.7.2 Executing the CDFinquire Program . . . . . 75
3.7.3 Output from the CDFinquire Program . . . . . 78
3.8 SkeletonTable . . . . . 82
3.8.1 Introduction . . . . . 82
3.8.2 Executing the SkeletonTable Program . . . . . 82
3.8.3 Output from the SkeletonTable Program . . . . . 84
3.9 CDFskeleton . . . . . 84
3.9.1 Introduction . . . . . 84
3.9.2 Executing the CDFskeleton Program . . . . . 85
3.9.3 Creating the Skeleton Table . . . . . 86
3.10 CDFbrowse . . . . . 87
3.11 CDFlist . . . . . 87
3.11.1 Introduction . . . . . 87
3.11.2 Executing the CDFlist Program . . . . . 87
3.11.3 Interaction with CDFlist . . . . . 89

3.12 CDFwalk . . . 92
3.12.1 Introduction . . . 92
3.12.2 Executing the CDFwalk Program . . . 92
3.12.3 Interaction with CDFwalk . . . 94
3.13 CDFedit . . . 96
3.13.1 Introduction . . . 96
3.13.2 Executing the CDFedit Program . . . 96
3.13.3 Interaction with CDFedit . . . 98

**4 CDF C Programming Reference 101**

4.1 Introduction . . . 101
4.2 Compiling . . . 101
4.2.1 Specifying `cdf.h` Location in the Compile Command . . . 101
4.2.2 Specifying `cdf.h` Location in the Source File . . . 103
4.3 Linking . . . 103
4.3.1 VMS Systems . . . 103
4.3.2 OpenVMS Systems . . . 104
4.3.3 UNIX Systems . . . 104
4.3.4 MS-DOS Systems . . . 105
4.4 Linking, Shared CDF Library . . . 106
4.4.1 VAX/VMS . . . 106
4.4.2 DEC Alpha/OpenVMS . . . 107
4.4.3 Sun/SunOS . . . 107
4.4.4 Sun/SOLARIS . . . 107
4.4.5 HP 9000/HP-UX . . . 107
4.4.6 IBM RS6000/AIX . . . 107
4.5 Item Referencing . . . 108
4.6 CDF Parameters — Predefined Constants and Types . . . 108

4.6.1 Defined Types . . . 108
4.6.2 CDFstatus Constants . . . 108
4.6.3 CDF Formats . . . 109
4.6.4 CDF Data Types . . . 109
4.6.5 Data Encodings . . . 110
4.6.6 Data Decodings . . . 110
4.6.7 Variable Majorities . . . 111
4.6.8 Record/Dimension Variances . . . 112
4.6.9 Attribute Scopes . . . 112
4.6.10 Read-Only Modes . . . 113
4.6.11 zModes . . . 113
4.6.12 `-0.0` to `0.0` Modes . . . 113
4.6.13 Operational Limits . . . 113
4.6.14 Limits of Names and Other Character Strings . . . 113
4.7 C Function Reference . . . 114
4.7.1 CDFcreate (Standard Interface) . . . 114
4.7.2 CDFopen (Standard Interface) . . . 116
4.7.3 CDFdoc (Standard Interface) . . . 117
4.7.4 CDFinquire (Standard Interface) . . . 118
4.7.5 CDFclose (Standard Interface) . . . 119
4.7.6 CDFdelete (Standard Interface) . . . 120
4.7.7 CDFerror (Standard Interface) . . . 121
4.7.8 CDFattrCreate (Standard Interface) . . . 122
4.7.9 CDFattrNum (Standard Interface) . . . 123
4.7.10 CDFattrRename (Standard Interface) . . . 124
4.7.11 CDFattrInquire (Standard Interface) . . . 125
4.7.12 CDFattrEntryInquire (Standard Interface) . . . 126

4.7.13 CDFattrPut (Standard Interface) . . . 128
4.7.14 CDFattrGet (Standard Interface) . . . 130
4.7.15 CDFattrCorrectScope (Standard Interface) . . . 131
4.7.16 CDFvarCreate (Standard Interface) . . . 133
4.7.17 CDFvarNum (Standard Interface) . . . 135
4.7.18 CDFvarRename (Standard Interface) . . . 136
4.7.19 CDFvarInquire (Standard Interface) . . . 137
4.7.20 CDFvarPut (Standard Interface) . . . 138
4.7.21 CDFvarGet (Standard Interface) . . . 140
4.7.22 CDFvarHyperPut (Standard Interface) . . . 141
4.7.23 CDFvarHyperGet (Standard Interface) . . . 143
4.7.24 CDFvarClose (Standard Interface) . . . 145
4.7.25 CDFlib (Internal Interface) . . . 146
4.8 Interpreting CDF Status Codes . . . 204
4.9 `CDF_EPOCH` Values . . . 206
4.9.1 `computeEPOCH` . . . 206
4.9.2 `EPOCHbreakdown` . . . 206
4.9.3 `parseEPOCH` . . . 207
4.9.4 `encodeEPOCH` . . . 207
4.9.5 `encodeEPOCH1` . . . 207
4.9.6 `encodeEPOCH2` . . . 207
4.10 Custom C Functions . . . 208

**5 CDF Fortran Programming Reference 209**

5.1 Introduction . . . 209
5.2 Compiling . . . 209
5.2.1 VMS Systems . . . 210
5.2.2 UNIX Systems . . . 210

5.2.3 MS-DOS Systems . . . 210
5.3 Linking . . . 211
5.3.1 VMS Systems . . . 212
5.3.2 OpenVMS Systems . . . 212
5.3.3 UNIX Systems . . . 212
5.3.4 MS-DOS Systems . . . 213
5.4 Linking, Shared CDF Library . . . 213
5.4.1 VAX/VMS . . . 214
5.4.2 DEC Alpha/OpenVMS . . . 214
5.4.3 Sun/SunOS . . . 214
5.4.4 Sun/SOLARIS . . . 214
5.4.5 HP 9000/HP-UX . . . 215
5.4.6 IBM RS6000/AIX . . . 215
5.5 Argument Passing . . . 215
5.6 Item Referencing . . . 216
5.7 CDF Parameters — Predefined Constants and Types . . . 216
5.7.1 Status Code Constants . . . 216
5.7.2 CDF Formats . . . 216
5.7.3 CDF Data Types . . . 216
5.7.4 Data Encodings . . . 217
5.7.5 Data Decodings . . . 218
5.7.6 Variable Majorities . . . 219
5.7.7 Record/Dimension Variances . . . 219
5.7.8 Attribute Scopes . . . 220
5.7.9 Read-Only Modes . . . 220
5.7.10 zModes . . . 220
5.7.11 `-0.0` to `0.0` Modes . . . 220

5.7.12 Operational Limits . . . 221
5.7.13 Limits of Names and Other Character Strings . . . 221
5.8 Fortran Function Reference . . . 221
5.8.1 CDF_create (Standard Interface) . . . 221
5.8.2 CDF_open (Standard Interface) . . . 223
5.8.3 CDF_doc (Standard Interface) . . . 224
5.8.4 CDF_inquire (Standard Interface) . . . 226
5.8.5 CDF_close (Standard Interface) . . . 227
5.8.6 CDF_delete (Standard Interface) . . . 228
5.8.7 CDF_error (Standard Interface) . . . 229
5.8.8 CDF_attr_create (Standard Interface) . . . 230
5.8.9 CDF_attr_num (Standard Interface) . . . 231
5.8.10 CDF_attr_rename (Standard Interface) . . . 232
5.8.11 CDF_attr_inquire (Standard Interface) . . . 233
5.8.12 CDF_attr_entry_inquire (Standard Interface) . . . 235
5.8.13 CDF_attr_put (Standard Interface) . . . 237
5.8.14 CDF_attr_get (Standard Interface) . . . 238
5.8.15 CDF_attr_correct_scope (Standard Interface) . . . 240
5.8.16 CDF_var_create (Standard Interface) . . . 241
5.8.17 CDF_var_num (Standard Interface) . . . 243
5.8.18 CDF_var_rename (Standard Interface) . . . 245
5.8.19 CDF_var_inquire (Standard Interface) . . . 246
5.8.20 CDF_var_put (Standard Interface) . . . 247
5.8.21 CDF_var_get (Standard Interface) . . . 249
5.8.22 CDF_var_hyper_put (Standard Interface) . . . 250
5.8.23 CDF_var_hyper_get (Standard Interface) . . . 252
5.8.24 CDF_var_close (Standard Interface) . . . 254

5.8.25 CDF_lib (Internal Interface) . . . 255
5.9 Interpreting CDF Status Codes . . . 315
5.10 CDF_EPOCH Values . . . 317
5.10.1 compute_EPOCH . . . 317
5.10.2 EPOCH_breakdown . . . 317
5.10.3 parse_EPOCH . . . 318
5.10.4 encode_EPOCH . . . 318
5.10.5 encode_EPOCH1 . . . 318
5.10.6 encode_EPOCH2 . . . 318

**A Skeleton Table Format 321**

A.1 Introduction . . . 321
A.2 Header Section . . . 322
A.3 gAttributes Section . . . 324
A.4 vAttributes Section . . . 326
A.5 rVariables Section . . . 327
A.6 zVariables Section . . . 330
A.7 End Section . . . 333
A.8 Example Skeleton Table . . . 333

**B CDF's IDL Support 339**

B.1 Introduction . . . 339
B.2 Using CDF's IDL Interface . . . 339
B.3 CDF "Include" Files . . . 340
B.4 On-Line Help . . . 341
B.5 Available Functions . . . 341
B.6 Example IDL Session . . . 343
B.7 Example IDL Function . . . 344

**C CDF Status Codes** **351**

C.1 Introduction . . . . . 351

C.2 Status Codes and Messages . . . . . 351

**D C Programming Summary** **359**

D.1 Standard Interface . . . . . 359

D.2 Internal Interface . . . . . 362

D.3 EPOCH Utility Routines . . . . . 367

**E Fortran Programming Summary** **369**

E.1 Standard Interface . . . . . 369

E.2 Internal Interface . . . . . 373

E.3 EPOCH Utility Routines . . . . . 377

**F CDF Release Notes** **379**

F.1 Introduction . . . . . 379

F.2 Supported Systems . . . . . 379

F.3 Changes for CDF V2.4 . . . . . 380

**G Glossary** **383**

**Index** **389**

# List of Figures


1.1 Conceptual View of a CDF, 0-Dimensional rVariables . . . . . 3

1.2 Conceptual View of a CDF, 2-Dimensional rVariables . . . . . 4

1.3 Conceptual View of a CDF, zVariables . . . . . 5

1.4 Multi-File Format . . . . . 6

1.5 Single-File Format . . . . . 7

2.1 Physical vs. Virtual Records . . . . . 43

2.2 Physical vs. Virtual Dimensions . . . . . 46

3.1 `Action Menu`, `CDFlist` . . . . . 89

3.2 Option Selection List, `CDFlist` . . . . . 91

3.3 `Action Menu`, `CDFwalk` . . . . . 94

3.4 Option Selection List, `CDFwalk` . . . . . 96

3.5 Window Sections, `CDFedit` . . . . . 99

# List of Tables


1.1 Example Data Set — "Flat" Representation (0-Dimensional) . . . . . 8

1.2 Example CDF — 2-Dimensional Representation (Conceptual) . . . . . 9

1.3 Example CDF — Specification for 2-Dimensional Representation . . . . . 9

1.4 Example CDF — 2-Dimensional Representation (Physical) . . . . . 10

1.5 vAttribute rEntries for the `Temperature` rVariable . . . . . 11

2.1 Standard Interface Routines . . . . . 26

2.2 Internal Interface Routines . . . . . 26

2.3 Equivalent Byte Orderings . . . . . 35

2.4 Equivalent Single-Precision Floating-Point . . . . . 35

2.5 Equivalent Double-Precision Floating-Point . . . . . 36

2.6 NSSDC Standard CDF Attributes . . . . . 56

3.1 Example rVariables, `CDFstats` Monotonicity Checking . . . . . 70

# 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 designing 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 multidimensional 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 toward 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 categories. The first is a series of records comprising a collection of variables consisting of scalers, vectors, and n-dimensional arrays. The second is a set of attribute entries (metadata) describing the CDF in global terms or specifically for a single variable. This dual function of CDF is what provides its "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 that indicate the subset of CDF dimensions that are applicable.

### 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 its 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 (e.g., 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 designing CDFs 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 that permit users to slice data across multidimensional subspaces, access entire structures of data, perform subsampling 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 & SOLARIS), DECstation (ULTRIX), DEC Alpha (OSF/1 & OpenVMS), Silicon Graphics Iris and Power Series (IRIX), IBM RS6000 series (AIX), HP 9000 series (HP-UX), NeXT (MaCH), 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 people first hear the term CDF they intuitively think of data formats in the traditional sense of the word (i.e., 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 that the CDF library generates.

## 1.3 Conceptual Organization

An important feature of CDF is that it can handle data sets that are inherently multidimensional in addition to data sets that 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 2-dimensional. Similarly, the array for volume data would be 3-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, one 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 with repeated observations at different times, altitudes, or pressure levels.

Two types of variables may exist in a CDF: rVariables[1] and zVariables.[2] Every rVariable in a CDF must have the same number of dimensions and dimension sizes. In the scalar data example the CDF's rVariables would be 0-dimensional, whereas for the image data example the CDF's rVariables would be 2-dimensional. Figures 1.1 and 1.2 illustrate 0-dimensional and 2-dimensional rVariables, respectively. zVariables may have a different number of dimensions and/or dimension sizes than that of the rVariables in a CDF. Figure 1.3 illustrates several zVariables. Note that a CDF may contain both rVariables and zVariables. The term "variable" is used when describing a property that applies to both rVariables and zVariables.

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 values are 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 values are organized in whatever way makes sense to the user.

## 1.4 Features of the CDF Library

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

### 1.4.1 File Format Options

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 1.4 (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 1.5, 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 also done with optimum efficiency.

[1] The "r" stands for "regular." rVariables are the type of variables that CDF has always supported. Perhaps "traditional" would have been a better term. We also considered "related" and "correlated" but those just didn't seem right.

[2] The "z" doesn't stand for anything special. We just like the letter "z."

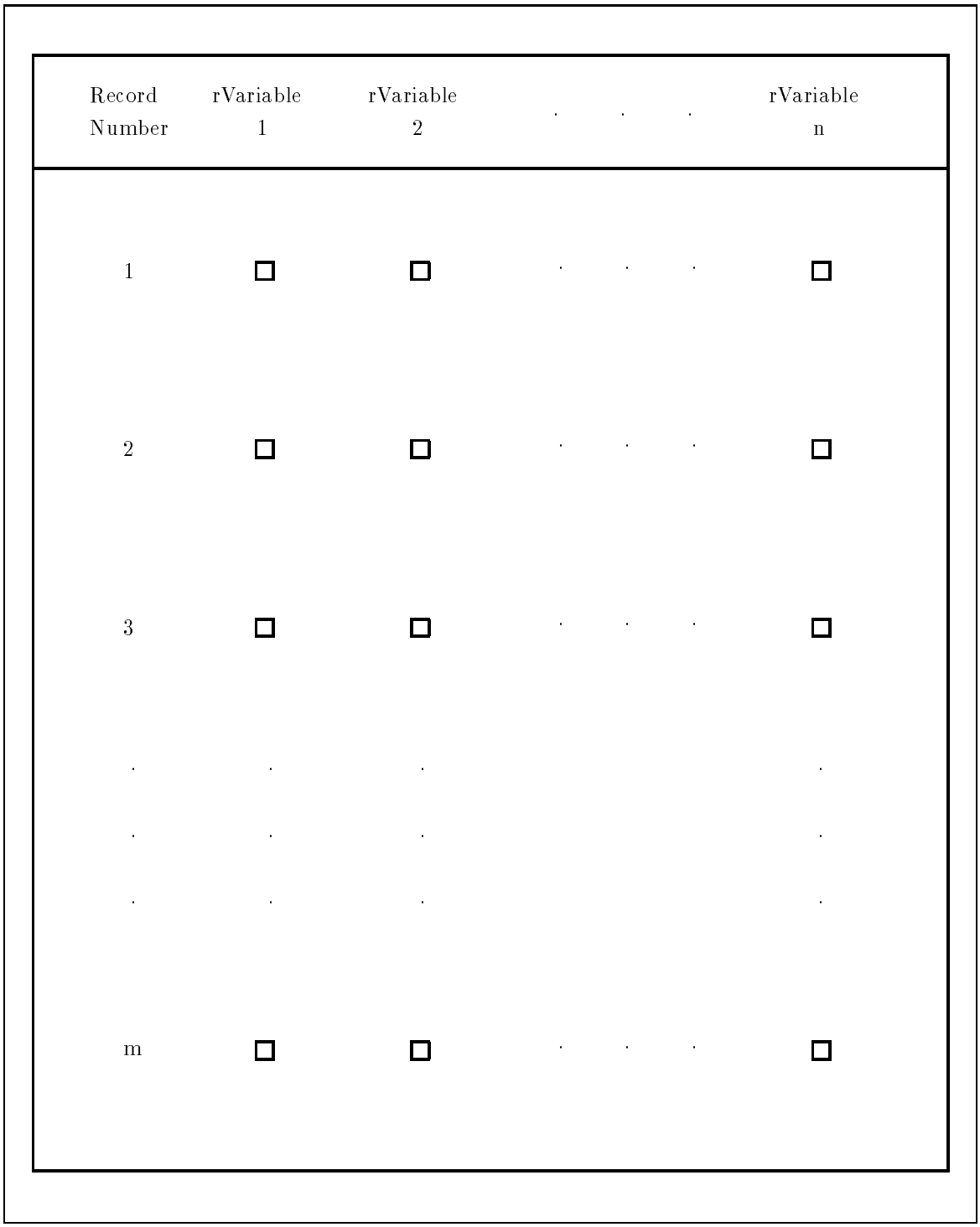


Figure 1.1: Conceptual View of a CDF, 0-Dimensional rVariables

### 1.4.2 Data Encoding Options

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. Using the host encoding scheme, the data in the CDF will be stored

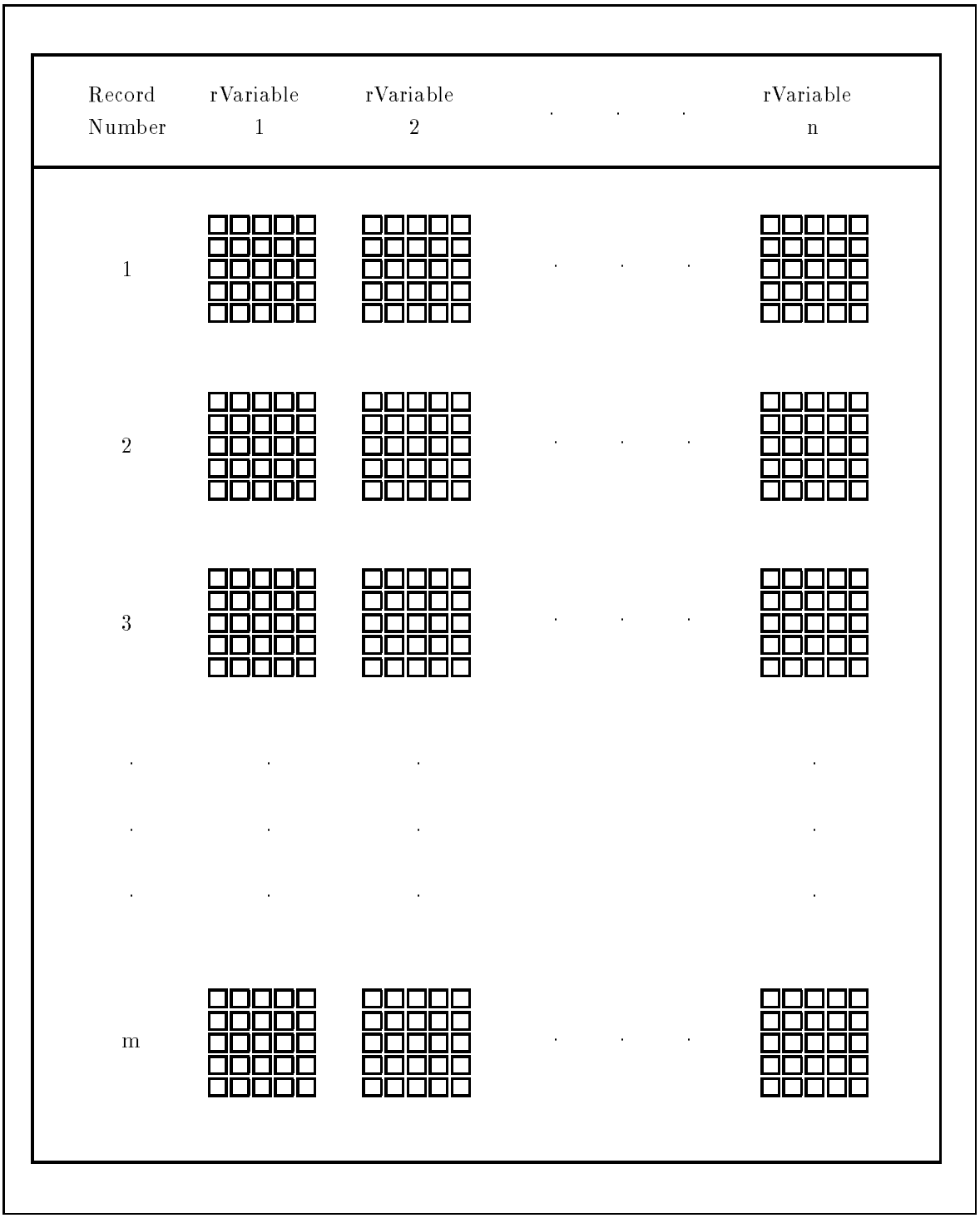


Figure 1.2: Conceptual View of a CDF, 2-Dimensional rVariables

in the physical representation used by the machine on which the CDF library is running (VAX, Sun, SGi Personal Iris and Power Series, DECstation, DEC Alpha/OSF1, DEC Alpha/OpenVMS, IBM RS6000 series,

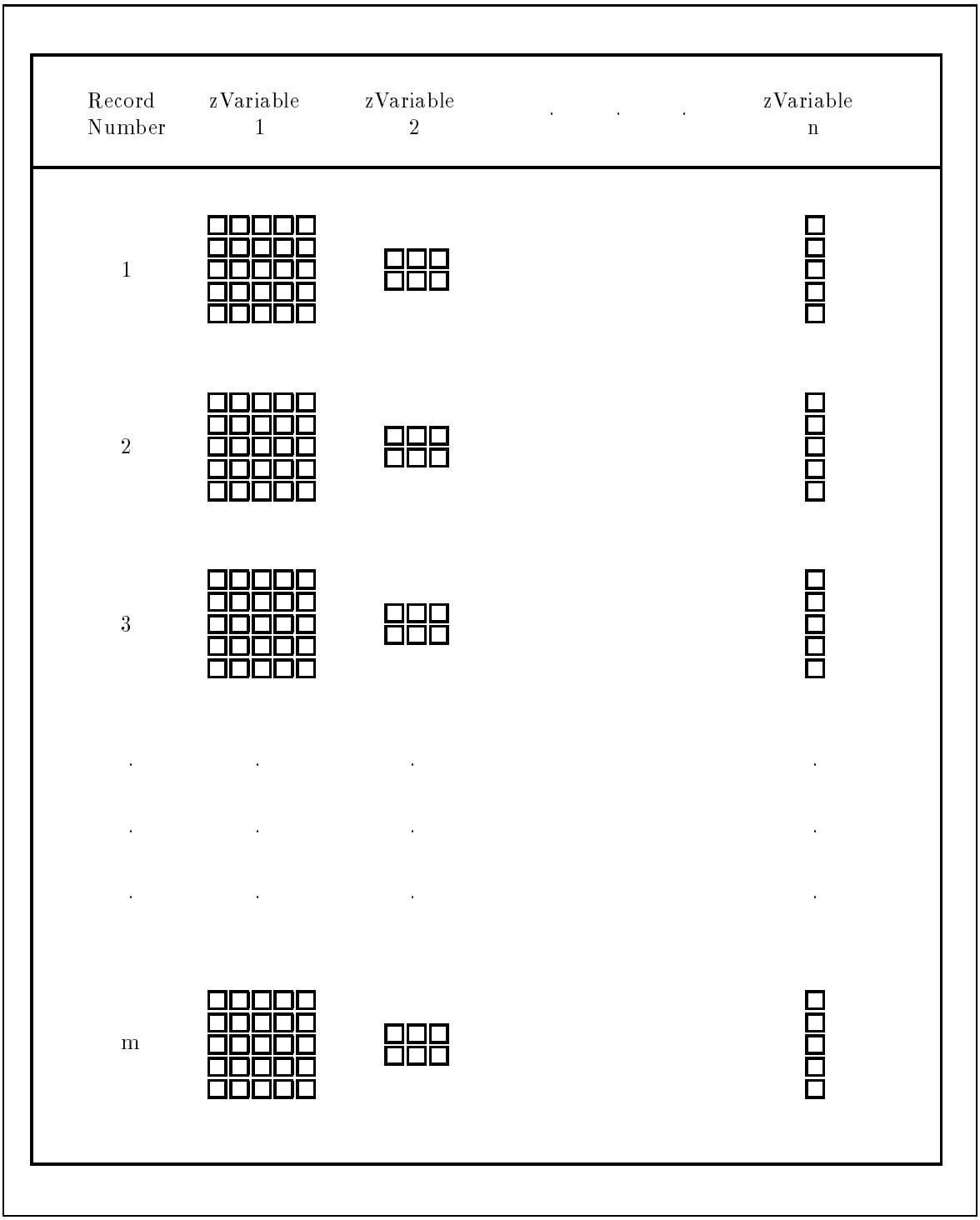


Figure 1.3: Conceptual View of a CDF, zVariables

HP 9000 series, NeXT, or IBM PC). The second encoding scheme is network encoding. When the network encoding scheme is used, the data values are transparently converted from the host representation to the

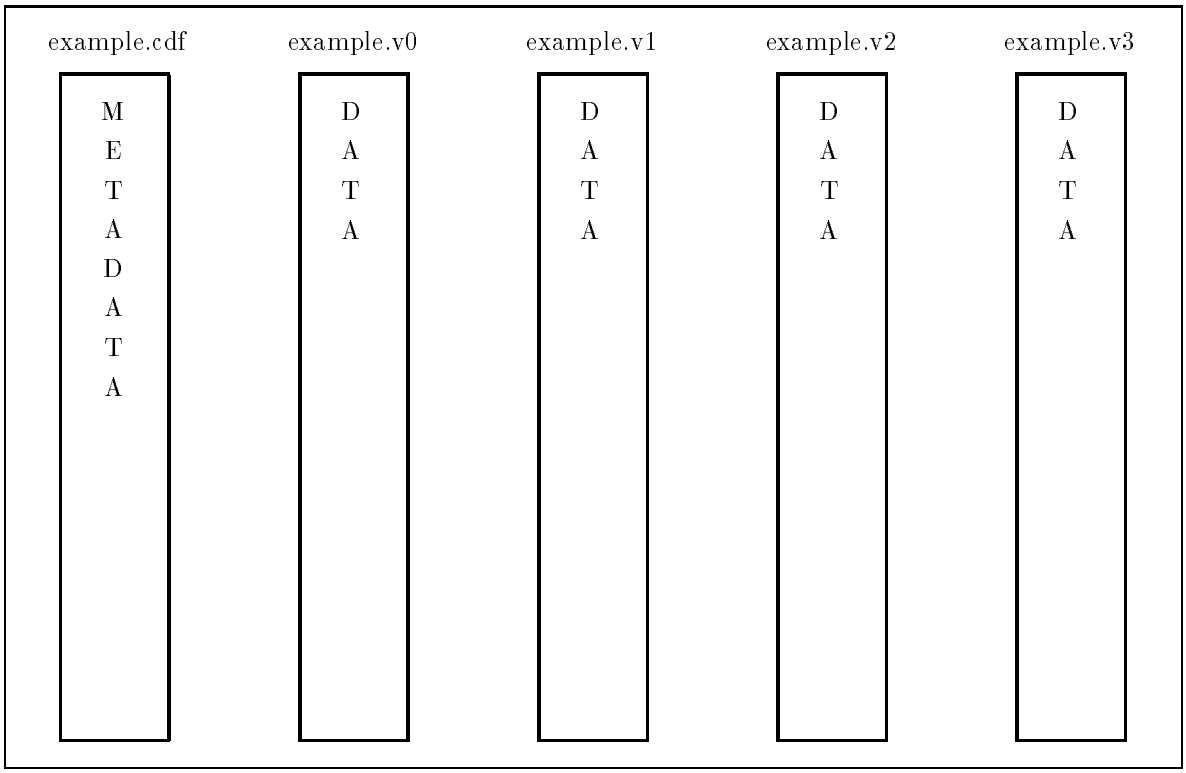


Figure 1.4: Multi-File Format

XDR[3] data representation when the data values are written and from the XDR representation to the host representation when the data values are read. This gives the user the ability to transfer the CDFs to other machines on which the CDF library runs and to read the data without having to be concerned with the data conversion.[4]

### 1.4.3 Variable Data Access Options

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

[3] eXternal Data Representation
[4] Beginning with CDF V2.3, all machine encodings are readable on all machines.

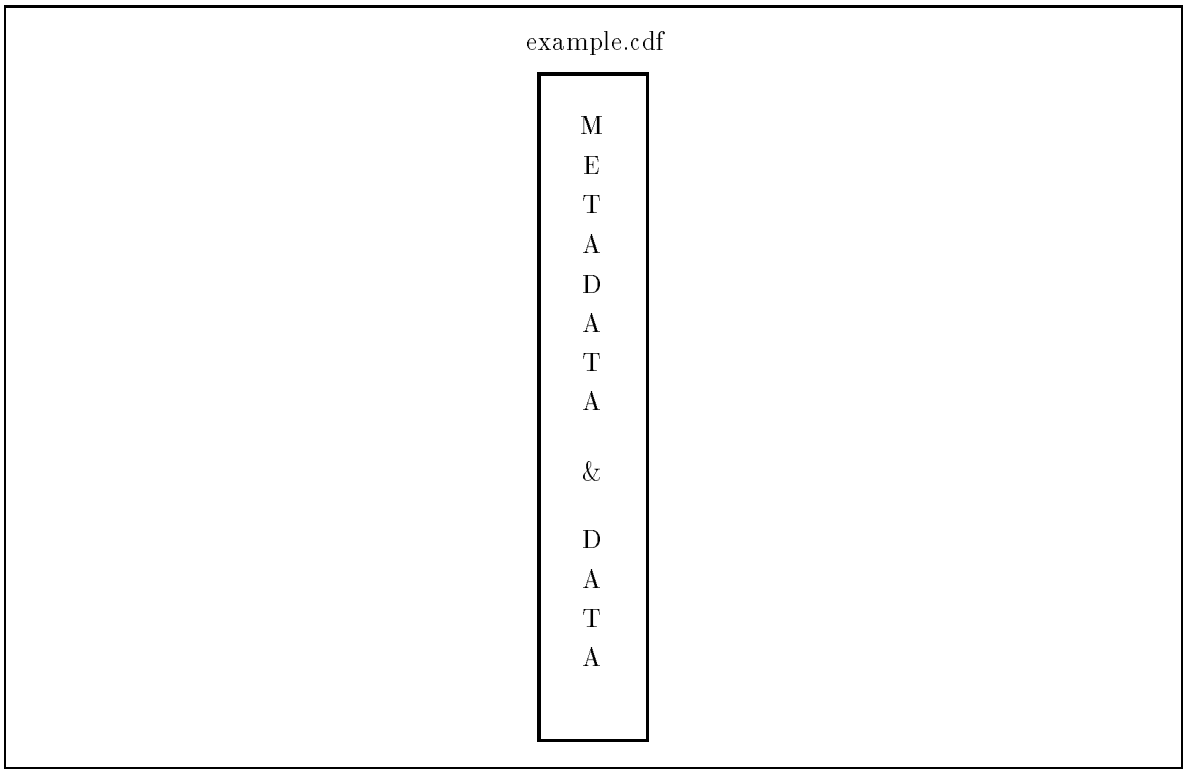


Figure 1.5: Single-File Format

## 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 that can be organized into arrays. Two types of variables are supported: rVariables and zVariables.

#### rVariables

rVariables all have the same dimensionality (number of dimensions and dimension sizes). An example of the type of data set that may be stored in a CDF's rVariables 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" (0-dimensional) 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 2-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 2-dimensional array per rVariable (Table 1.2). The array structure defines the dimensionality of the rVariables in the CDF. Although there are four rVariables, 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 CDF with 2-dimensional rVariables. The number of discrete values for each rVariable that 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.

| Record | rVariables | | | |
|---|---|---|---|---|
| Number | `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 (0-Dimensional)

Adding another independent rVariable, for instance `Pressure`, poses no difficulty for the example. `Temperature` would then be dependent on a specific `Longitude`, `Latitude`, and `Pressure` — a 3-dimensional array structure. In this 3-dimensional example `Longitude`, `Latitude`, and `Pressure` define the number of dimensions for the rVariables in the CDF, where the size of each dimension is determined by the number of discrete values contained in each of those rVariables. Additional dependent rVariables would be stored in the same way as `Temperature`.

Although conceptually there is a 2-dimensional array structure for each rVariable 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 2-dimensional array (Table 1.2). This problem is circumvented by specifying "variances." For each rVariable there are variances associated with the array dimensions as well as the records. "Record variance" indicates whether or not an rVariable 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

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

Table 1.2: Example CDF — 2-Dimensional Representation (Conceptual)

this example are shown in Table 1.3.

Similarly, the term "dimension variance" indicates whether or not an rVariable changes with respect to the CDF dimensions. In the example above with 2-dimensional rVariables, the `Longitude` rVariable 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` rVariable 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.

| | rVariables | | | |
|---|---|---|---|---|
| | `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 2-Dimensional Representation

When the record and dimension variances have been defined correctly, the amount of physical storage needed for the CDF is drastically reduced. In the above example, 2-dimensional arrays are not physically stored for each rVariable in a CDF record. Instead, the physical storage for each rVariable consists of just one value for `Time` in each CDF record, a single 1-dimensional array of values for the `Longitude` and `Latitude` rVariables (in only the first CDF record), and a full 2-dimensional array of values for `Temperature` in each CDF record. The actual physical storage (physical view) is shown in Table 1.4. The conceptual view of the CDF, however, is still that of one 2-dimensional array per rVariable in each CDF record as shown in

Table 1.2 (the physically stored values are shown in boldface type).

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

Table 1.4: Example CDF — 2-Dimensional Representation (Physical)

### zVariables

zVariables are identical to rVariables in all but one respect. The dimensionality of a zVariable may be different from that of the rVariables (and other zVariables). This allows variables that do not "fit" into the dimensionality of the rVariables to be included in the same CDF. Using only rVariables to store certain data sets could result in wasted disk space or a CDF with rVariables whose dimensionality is not consistent with some of the data in the data set. zVariables are intended to solve these problems. Note that a CDF may also contain just zVariables.

Consider a data set that consists of some number of images, each containing 1024 by 1024 pixels. The data set also contains a palette that is used to map pixel values to the actual color/shade to be displayed. Palettes are also referred to as lookup tables or color lookup tables. For this example assume that each image pixel is stored in an 8-bit byte and the palette is a 1-dimensional array of 256 colors/shades. Indexing into the palette array with a pixel value gives the appropriate color/shade to use.

Attempting to store the images and the palette using only rVariables would result in one of two undesirable situations. If the CDF's rVariables are 2-dimensional with sizes `[1024,1024]` (to store the images), the palette would have to be stored in a 1024 by 1024 array that does not make sense logically and would waste disk space regardless of how the dimension variances are set. If the CDF's rVariables are 3-dimensional with sizes `[1024,1024,256]`, the images could be stored in an rVariable having dimension variances `[TRUE,TRUE,false]` and the palette could be stored in an rVariable having dimension variances `[false,false,TRUE]`. This would not waste any disk space but is not the intuitive way to store the data — nothing in the data set is 3-dimensional.

Using a zVariable to store the palette would solve both problems. The CDF's rVariables would be defined as

being 2-dimensional with sizes `[1024,1024]`. The images would be stored in an rVariable having dimension variances `[TRUE,TRUE]`, but the palette would be stored in a zVariable defined to be 1-dimensional with a size of `[256]` (and dimension variance of `[TRUE]`). This would waste no disk space and logically makes sense. Note that the images could also have been stored in a zVariable defined as 2-dimensional with sizes `[1024,1024]` and dimension variances `[TRUE,TRUE]`.

## 1.6 Attributes

The second component of a CDF is the metadata. Metadata values consist of user-supplied descriptive information about the CDF and the variables in the CDF by way of attributes and attribute entries. Attributes can be divided into two categories: attributes of global scope (gAttributes) and attributes of variable scope (vAttributes). gAttributes describe the CDF as a whole while vAttributes describe some property of each variable (rVariables and zVariables) in the CDF. Any number of attributes may be stored in a single CDF. Section 2.5 describes standard CDF attributes used by the NSSDC. The term "attribute" is used when describing a property that applies to both gAttributes and vAttributes.

gAttributes 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. gAttributes may contain multiple entries (called gEntries). An example of this would be a modification history kept in the optional gAttribute, `MODS`. This attribute could be specified at CDF creation time and a gEntry 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 writing additional gEntries to `MODS`.

vAttributes further describe the individual variables and their values. Examples of vAttributes 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). The entries of a vAttribute correspond to the variables in the CDF. Each rEntry corresponds to an rVariable and each zEntry corresponds to a zVariable. Sample vAttribute rEntries for the `Temperature` rVariable from the example above are shown in Table 1.5.

The term "entry" is used when describing a property that applies to gEntries, rEntries, and zEntries.

| vAttribute | rEntry 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: vAttribute rEntries for the `Temperature` rVariable

## 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:

| | |
|---|---|
| `CDFedit`[5] | Allows the display, creation, and modification of attribute and variable data in a CDF. |
| `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` | 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 Library Interface Routines

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 available using the Internal Interface (e.g., all access to zVariables). The Internal Interface makes CDF an easily extensible software package.

The Standard and Internal interfaces are callable from both C and Fortran. The Obsolete Interface is callable only from Fortran.

[5] `CDFedit` has replaced `CDFbrowse`. The alias/symbol `CDFbrowse` still exists in the "definitions" file on UNIX/VMS systems but now executes `CDFedit` in a browse-only mode.

### 1.8.1 Standard Interface

The Standard Interface consists of three categories of software functions that are utilized to manipulate the components that make up a CDF: general CDF functions, rVariable functions, and attribute functions.

The general CDF functions are as follows:

| 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 rVariable dimensionality, etc. |
| `CDFclose()` | `CDF_close()` | Closes a CDF. |
| `CDFdelete()` | `CDF_delete()` | Deletes a CDF. |
| `CDFerror()` | `CDF_error()` | Inquires error (status) code meaning. |

The rVariable functions are as follows:

| Callable from C | Callable from Fortran | Purpose |
|---|---|---|
| `CDFvarCreate()` | `CDF_var_create()` | Creates an rVariable. |
| `CDFvarNum()` | `CDF_var_num()` | Determines an rVariable number. |
| `CDFvarRename()` | `CDF_var_rename()` | Renames an rVariable. |
| `CDFvarInquire()` | `CDF_var_inquire()` | Inquires about an rVariable. |
| `CDFvarPut()` | `CDF_var_put()` | Writes an rVariable value. |
| `CDFvarGet()` | `CDF_var_get()` | Reads an rVariable value. |
| `CDFvarHyperPut()` | `CDF_var_hyper_put()` | Writes one or more rVariable values. |
| `CDFvarHyperGet()` | `CDF_var_hyper_get()` | Reads one or more rVariable values. |

The attribute functions are as follows:

| 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 rEntry. |
| `CDFattrPut()` | `CDF_attr_put()` | Writes an attribute rEntry. |
| `CDFattrGet()` | `CDF_attr_get()` | Reads an attribute rEntry. |

The Standard Interface may be used to access only rVariables and the vAttribute rEntries for rVariables. The Internal Interface must be used to access zVariables and the vAttribute zEntries for zVariables.

### 1.8.2 Internal Interface

The Internal Interface consists of one routine: `CDFlib` when called from C and `CDF_lib` when called from Fortran. The Internal Interface is used to perform all CDF operations. (In reality the Standard Interface is implemented via the Internal Interface.) The Internal Interface is used to add new CDF features (e.g., zVariables) without having to change the Standard Interface.

## 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 Hard Way (But Not That Hard)

The first example program, written in C, creates a CDF with 2-dimensional rVariables 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 (minus some of the vAttributes/rEntries). 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 1.0, 5-Jan-94, CDF, Inc.
*
*  Modification history:
*
*   V1.0   5-Jan-94, Joe Programmer     Original version.
*
******************************************************************************/

/******************************************************************************
*
* Note(s):
*
*   This program would have to be modified to run on a DEC Alpha because the
* C language 'long' data type is 8 bytes rather than 4 (the CDF data type of
* CDF_INT4 is always 4 bytes).
*
******************************************************************************/

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

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

#include "cdf.h"

/******************************************************************************
* 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;
}

/******************************************************************************
* 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, rVariables. */
  static long dimSizes[2] = {2,2};   /* Dimension sizes, rVariables. */

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

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

  long Time;           /* 'Time' rVariable value. */
  float Lat;           /* 'Latitude' rVariable value. */
```

```
float Lon;           /* 'Longitude' rVariable value. */
float Tmp;           /* 'Temperature' rVariable value. */

long TimeValidMin = 0;          /* 'Time' valid minimum (0000). */
long TimeValidMax = 2359;       /* 'Time' valid maximum (2359). */

float LonValidMin = -180.0;     /* 'Longitude' valid minimum. */
float LonValidMax = 180.0;      /* 'Longitude' valid maximum. */

float LatValidMin = -90.0;      /* 'Latitude' valid minimum. */
float LatValidMax = 90.0;       /* 'Latitude' valid maximum. */

float TmpValidMin = -40.0;      /* 'Temperature' valid minimum. */
float TmpValidMax = 50.0;       /* 'Temperature' valid maximum. */

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

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

/******************************************************************************
* Create rVariables.
******************************************************************************/

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

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

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

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

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

/******************************************************************************
* Write TITLE gAttribute gEntry.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"TITLE"), 0L, CDF_CHAR, 50L,
                     "An example CDF (1).                                ");
if (status != CDF_OK) StatusHandler (status);

/******************************************************************************
* Write vAttribute rEntries for 'Time' rVariable.
******************************************************************************/

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

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

/******************************************************************************
* Write vAttribute rEntries for 'Longitude' rVariable.
******************************************************************************/

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

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

/******************************************************************************
* Write vAttribute rEntries for 'Latitude' rVariable.
******************************************************************************/

status = CDFattrPut (id, CDFattrNum(id,"VALIDMIN"),
                     CDFvarNum(id,"Latitude"), CDF_REAL4, 1L, &LatValidMin);
```

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

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

/*****************************************************************************
* Write vAttribute rEntries for 'Temperature' rVariable.
*****************************************************************************/

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

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

/*****************************************************************************
* Read input values for rVariables 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;
}
```

### 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 called 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: Wed  5 Jan 1994 10:53:58

#header

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

! Variables G.Attributes V.Attributes Records  Dims  Sizes
! --------- ------------ ------------ -------  ----  -----
     4/0           1            2       1/z      2    2 2


#GLOBALattributes

! Attribute         Entry       Data
```

```
! Name              Number      Type       Value
! ---------         ------      ----       -----

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


#VARIABLEattributes

  "VALIDMIN"
  "VALIDMAX"


#variables

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

  "Time"            CDF_INT4        1          T         F F

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

    "VALIDMIN"      CDF_INT4     { 0 }
    "VALIDMAX"      CDF_INT4     { 2359 } .


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

  "Longitude"       CDF_REAL4       1          F         T F

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

    "VALIDMIN"      CDF_REAL4    { -180.0 }
    "VALIDMAX"      CDF_REAL4    { 180.0 } .

  ! 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
  ! --------          ----       -----

    "VALIDMIN"      CDF_REAL4    { -90.0 }
    "VALIDMAX"      CDF_REAL4    { 90.0 } .

  ! 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
  ! --------          ----       -----

    "VALIDMIN"      CDF_REAL4    { -40.0 }
    "VALIDMAX"      CDF_REAL4    { 50.0 } .

#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, 5-Jan-94, CDF, Inc.
C
C  Modification history:
C
C   V1.0   5-Jan-94, Joe Programmer      Original version.
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' rVariable number.
        INTEGER*4 tmp_var_num   ! 'Temperature' rVariable number.

        INTEGER*4 time          ! 'Time' rVariable value.
        REAL*4    lat           ! 'Latitude' rVariable value.
        REAL*4    lon           ! 'Longitude' rVariable value.
        REAL*4    tmp           ! 'Temperature' rVariable 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 rVariable 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 rVariables and write them to the CDF.  Not
C every value must be written to the CDF - many of the values are redundant.
C The 'Time' value only has to be written once per CDF record (every 4 input
C records).  The 'Longitude' and 'Latitude' values are not written at all
C because they had been specified in the skeleton table.  Each 'Temperature'
C value read 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` rVariables 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 Library

The CDF library is the only way to access a CDF. Various properties of the CDF library are described in the following sections.

### 2.1.1 Interfaces

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

#### 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 (e.g., access to zVariables). 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.

#### 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.

[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.

| 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 rVariable dimensionality, encoding, 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 an rVariable. |
| CDFvarNum() | CDF_var_num() | Determines an rVariable number. |
| CDFvarRename() | CDF_var_rename() | Renames an rVariable. |
| CDFvarInquire() | CDF_var_inquire() | Inquires about an rVariable. |
| CDFvarPut() | CDF_var_put() | Writes an rVariable value. |
| CDFvarGet() | CDF_var_get() | Reads an rVariable value. |
| CDFvarHyperPut() | CDF_var_hyper_put() | Writes one or more rVariable values. |
| CDFvarHyperGet() | CDF_var_hyper_get() | Reads one or more rVariable values. |
| | | |
| 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

**Obsolete Interface, VAX/VMS Systems Only**

Fortran applications written for CDF Version 1 on VAX/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 CDF library (LIBCDF_OBSOLETE.OLB). An example of this would be as follows:

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

where file1, file2, ...fileN are the application's object modules. CDF$LIB is defined by executing the definitions file, DEFINITIONS.COM, provided with the CDF distribution.

There are several conditions that 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).

#### CDF's IDL Interface

The CDF distribution contains an interface that allows full access to the CDF library (and hence CDFs) from within IDL. CDF's IDL interface consists of a set of functions that mirror the functions in the Standard and Internal interfaces for C and Fortran applications. CDF's IDL interface is described in Appendix B.

IDL also provides its own interface to the CDF library (as well as other data formats) that differs from CDF's IDL interface. The differences are mainly syntactic with the functionality of the two interfaces being essentially the same. IDL's documentation describes their built-in CDF interface. Another difference between the two interfaces is that CDF's IDL interface is only available on those computers that support dynamic linking. Appendix B lists the computers on which this is the case.

### 2.1.2 CDF Modes

Once a CDF has been opened (or created and not yet closed), the CDF library may be configured to act on that CDF in one or more modes. These modes are specified independently for each open CDF.

#### Read-Only Mode

A CDF may be placed in read-only mode via the Internal Interface using the `<SELECT_,CDF_READONLY_MODE_>` operation. Only read access will be allowed on the CDF — all attempts to modify the CDF will fail. A CDF may be toggled in and out of read-only mode any number of times. (Note that attempts to modify a CDF may also fail if insufficient access privileges exist for the CDF — the file system enforces this access.)

#### zMode

A CDF may be placed into zMode[2] via the Internal Interface using the `<SELECT_,CDF_zMODE_>` operation. When in zMode a CDF's rVariables essentially disappear and are replaced by corresponding zVariables.[3] Likewise, the rEntries for a vAttribute become zEntries (because they are now associated with zVariables). While in zMode most operations involving rVariables/rEntries will fail. (Some inquiry operations will be

[2] There are actually two types of zMode — read on.

[3] In a future release of CDF, support for rVariables will be eliminated. zMode is provided to ease the transition from rVariables to the more flexible zVariables. rVariables are essentially a subset of zVariables.

allowed. For example, inquiring the number of rVariables [always zero] is allowed.) When zMode is used, the number of variables remains the same — rVariables simply change into zVariables. Note that the existing contents of the CDF are not changed — the CDF simply appears different.

Each new zVariable has the same exact properties as the corresponding (hidden) rVariable except for dimensionality and variances. The data specification (data type and number of elements), pad value, etc. stay the same. The dimensionality/variances of each zVariable are dependent on which zMode is currently being used: zMode/1 or zMode/2. In zMode/1 the dimensionality/variances stay exactly the same. In zMode/2, however, those dimensions with a false variance (`NOVARY`) are eliminated. Consider a CDF with an rVariable dimensionality of `2:[180,360]`[4] containing the following rVariables.

| rVariable Name | Variances |
|---|---|
| EPOCH | T/FF[5] |
| LATITUDE | T/TF |
| LONGITUDE | T/FT |
| HUMIDITY | T/TT |

If this CDF were to be placed into zMode/1, the following zVariables would replace the existing rVariables.

| zVariable Name | Dimensionality | Variances |
|---|---|---|
| EPOCH | 2:[180,360] | T/FF |
| LATITUDE | 2:[180,360] | T/TF |
| LONGITUDE | 2:[180,360] | T/FT |
| HUMIDITY | 2:[180,360] | T/TT |

Note that the dimensionality of each zVariable is the same as it was for the rVariables in the CDF. However, if zMode/2 were used, the following zVariables would replace the existing rVariables.

| zVariable Name | Dimensionality | Variances |
|---|---|---|
| EPOCH | 0:[] | T/ |
| LATITUDE | 1:[180] | T/T |
| LONGITUDE | 1:[360] | T/T |
| HUMIDITY | 2:[180,360] | T/TT |

In this case the false dimensional variances were removed (which decreased the dimensionality in several of the variables).

A CDF can be placed into or taken out of zMode any number of times while it is open. Each time the zMode is changed for a CDF, it would be best to think of the CDF as being closed and reopened in that zMode. The numbering of variable/entries may or may not be as you would expect (and the scheme used could change in a future release of CDF). Most applications will simply select a zMode immediately after opening a CDF. (zMode being off is the default if a zMode is not selected.)

**NOTE:** Using zMode does not change the contents of a CDF. A CDF containing rVariables will appear to contain only zVariables when in zMode. If the same CDF is then opened without using zMode, the rVariables will still exist.

[4] This notation is used throughout this document. In this case there are two dimensions whose sizes are `180` and `360`. A dimensionality of zero is represented as `0:[]`.

[5] This notation is also used throughout this document. The record variance is before the slash and the dimension variances are after the slash. In this case the record variance is true (`VARY`) and the dimension variances are both false (`NOVARY`). The variances for a 0-dimensional variable would be represented as `T/` (if the record variance were true).

#### -0.0 to 0.0 Mode

The floating-point value -0.0 is legal on those computers which use the IEEE 754 floating-point representation (e.g., most UNIX-based computers and the IBM PC) but is illegal on VAXes and DEC Alphas running OpenVMS. Attempting to use -0.0 results in a reserved operand fault on a VAX and a high performance arithmetic fault on a DEC Alpha running OpenVMS. Because of this the CDF library can be told to convert -0.0 to 0.0 when read from a CDF. The values physically stored in the CDF file(s) are not modified — only the values returned to an application are converted. This mode is available on all supported computers but is only really necessary on VAXes and DEC Alphas running OpenVMS. The CDF library is told to convert -0.0 to 0.0 for a CDF via the Internal Interface using the <SELECT_,CDF_NEGtoPOSfp0_MODE_> operation.

### 2.1.3 Limits

#### Open CDFs

The only limit on the number of CDFs that may be open simultaneously is the operating system's limit on the number of open files that an application may have. Each open CDF will always have at least one associated open file (the .cdf file). The CDF library will open and close the variable files of a multi-file CDF as needed (see Sections 2.3.2 and 2.3.3).

### 2.1.4 Caching Scheme

The CDF library reads and writes to open files in 512-byte blocks. A cache of 512-byte memory buffers is maintained by the CDF library for each open file. The CDF library attempts to keep in the cache the set of file blocks currently being accessed. This results in fewer actual I/O operations to the file if repeated accesses to these blocks would occur. When the cache is completely full and a new block of the file is accessed, one of the cache buffers is written back to the file (if it was modified) and the new block is read into that cache buffer (unless the file is being extended in which case the cache buffer is simply cleared). This process is known as paging. By optimizing the number of cache buffers for a file, improved performance can be achieved. There is tradeoff between having too few cache buffers and having too many. Having too few cache buffers will cause excessive paging while having too many cache buffers may slow performance because of the overhead involved in maintaining the cache.

The CDF library attempts to choose optimal default cache sizes based on a CDF's format and number of variables. This is difficult since the CDF library does not know how an application will access a CDF. For that reason an application may specify, via the Internal Interface, the number of cache buffers to be used for a file. The number of cache buffers may be changed as many times as necessary while a file is open (the first time will override the default used by the CDF library). The default numbers of cache buffers used by the CDF library in various situations are as follows:

| | |
|---|---|
| Created CDF, single-file (.cdf) | MAXcacheCREATEDsingleCDF |
| Created CDF, multi-file (.cdf) | MAXcacheCREATEDmultiCDF |
| Opened CDF, single-file (.cdf) | The smaller of: "<total-number-of-variables> + PLUScacheOPENEDsingle" and MAXcacheOPENEDsingleCDF |
| Opened CDF, multi-file (.cdf) | MAXcacheOPENEDmultiCDF |

| | |
|---|---|
| Variable files (multi-file) | MAXcacheACCESSmultiVAR |

The values of the constants shown may be configured for your CDF distribution when it is built and installed. Consult your system manager for the values of these defaults. Note that these values are the maximum number of cache buffers for a file. The cache buffers are only actually allocated as needed.

The situations in which it will be necessary to specify a cache size will depend on how a CDF is accessed. For example, consider a variable in a row-major CDF having a dimensionality of `2:[10,64]`, a data specification of `CDF_REAL8/1`, and variances of `T/TT`. This variable definition results in each record of the variable being written across 10 file blocks with the second dimension varying the fastest (since the CDF's variable majority is row-major). If single value reads were used to access this variable (see Section 2.3.12), only one cache buffer would be necessary if the second dimension were incremented the fastest (i.e., `[1,1]`, `[1,2]`,...`[10,63]`, `[10,64]`). This is because the values of a record would be accessed sequentially from the first block to the last block. If, however, the first dimension were incremented the fastest (i.e., `[1,1]`, `[2,1]`,...`[9,64]`, `[10,64]`), 10 cache buffers would improve performance. The values of a record are not being accessed sequentially but rather each read would be from a different block. Since the reads would be spread access 10 blocks, having 10 cache buffers would be optimal.

A similar situation arises when accessing variables in a single-file CDF. If values are accessed for each variable at a record number, then performance will be improved by setting the number of cache buffers to be the same as the number of variables. This is because the variable values will most likely be located in that many different file blocks for a particular record number.

In a C application cache sizes are specified using the `<SELECT_,CDF_CACHESIZE_>`, `<SELECT_,rVAR_CACHESIZE_>`, `<SELECT_,rVARs_CACHESIZE_>`, `<SELECT_,zVAR_CACHESIZE_>`, and `<SELECT_,zVARs_CACHESIZE_>` operations of the `CDFlib` function (Internal Interface). In a Fortran application cache sizes are specified using the same operations of the `CDF_lib` subroutine (Internal Interface).

## 2.2 CDFs

The following sections describe various aspects of a CDF.

### 2.2.1 Accessing

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

#### 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 2.1.1).

**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.

### 2.2.2 Creating

A CDF must be created by the CDF library. In a C application CDFs are created using either the `CDFcreate` function (Standard Interface) or the `<CREATE_,CDF_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application CDFs are created using either the `CDF_create` subroutine (Standard Interface) or the `<CREATE_,CDF_>` operation of the `CDF_lib` subroutine (Internal Interface).

### 2.2.3 Opening

An application must open an existing CDF before access to that CDF is allowed by the CDF library. In a C application CDFs are opened using either the `CDFopen` function (Standard Interface) or the `<OPEN_,CDF_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application CDFs are opened using either the `CDF_open` subroutine (Standard Interface) or the `<OPEN_,CDF_>` operation of the `CDF_lib` subroutine (Internal Interface).

### 2.2.4 Closing

It is absolutely essential that a CDF that has been created or modified by an application be closed before the program exits. If the CDF is not closed it will in most cases be corrupted and unreadable. This is because the cache buffers maintained by the CDF library will not have been written to the CDF file(s). An existing CDF that has been opened and only read from should also be closed. In a C application CDFs are closed using either the `CDFclose` function (Standard Interface) or the `<CLOSE_,CDF_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application CDFs are closed using either the `CDF_close` subroutine (Standard Interface) or the `<CLOSE_,CDF_>` operation of the `CDF_lib` subroutine (Internal Interface).

### 2.2.5 Naming

The pathname specified when opening or creating a CDF can be any legal pathname for the operating system being used. This includes logical symbols on VMS systems and environment variables on UNIX systems. Trailing blanks are also allowed but will be ignored. This is so Fortran applications do not have to be concerned with the trailing blanks of a Fortran `CHARACTER` variable. (C character string variables use terminating `NUL` characters.)

In almost all cases when a CDF pathname is specified, the `.cdf` extension should not be appended. (It will be appended automatically by the CDF library.) The exception to this is when a user has renamed an existing CDF with a different extension or with no extension (for whatever reason). When a CDF is opened, the CDF library first appends the `.cdf` extension to the pathname specified and then checks to see if that file

exists.[6] If not, the CDF library will also check to see if a file exists whose pathname is exactly as specified (without `.cdf` appended). If this is the case, the CDF must be single-file. If the CDF is multi-file, an error occurs since the CDF library would have no idea as to how the variable files had been renamed. Note also that the CDF library always appends `.cdf` to the pathname specified when creating a CDF.

**NOTE:** The CDF toolkit programs will in some cases not recognize a CDF if it does not have an extension of `.cdf`.[7]

### 2.2.6 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 an existing CDF (creating a new CDF).

The default format for a created CDF was determined when your CDF distribution was built and installed. Consult your system manager for this default. In a user application, the Internal Interface must be used to change the format of a CDF. The format of an existing CDF can be changed only 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).

In a C application a CDF's format is changed by using the `<PUT_,CDF_FORMAT_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application a CDF's format is changed by using the `<PUT_,CDF_FORMAT_>` operation of the `CDF_lib` subroutine (Internal Interface).

#### 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`,... for rVariables and `.z0`,`.z1`,... for zVariables). Each variable file contains the data values for the corresponding variable. (The control information for each variable is stored in the `.cdf` file.)

#### 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 records.

**Indexing Scheme.** In single-file CDFs an indexing scheme is used to keep track of where a variable's records are located within the `.cdf` file. The order that variable (and attribute entry) values are written to a single-file CDF by an application may result in a variable's records being noncontiguous. There will be groups of contiguous records, but these groups will not be contiguous. This section will give a brief overview of how a variable's records are indexed.

[6] Actually, the CDF library will check several possible extensions: `.cdf`, `.cdf;1`, `.CDF`, and `.CDF;1`. These extensions are checked because some CD-ROM drivers (primarily on UNIX machines) do peculiar things when making the files (CDFs) on a CD-ROM visible.

[7] Or `.cdf;1` or `.CDF` or `.CDF;1`.

For each variable in a single-file CDF one or more index records will exist. Each of these index records will contain one or more index entries.[8] An index entry consists of the following fields:

| | |
|---|---|
| `starting record` | The number of the first record in a group of contiguous records. |
| `ending record` | The number of the last record in a group of contiguous records. |
| `byte offset` | The byte offset within the `.cdf` file of the group of contiguous records. |

To find a particular variable record the CDF library must search through the index entries for that variable. Improved performance will result if there are fewer index entries to search. This can be achieved by having a large number of records in each group of contiguous records (resulting in fewer groups/index entries). Sections 2.3.7 and 2.3.7 describe techniques that can be used to achieve this.

It is possible to inquire the indexing statistics for a variable. Using the Internal Interface, an application may inquire the number of index records and total number of index entries.

#### 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 that may be open at once by an application) and the variables values are accessed variable-by-variable (rather than accessing an entire variable before going to the next 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`.

### 2.2.7 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.

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

[8] Actually, all but the last index record will be completely full of index entries. The last index record will contain one or more index entries.

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 the host encoding of the computer being used or network encoding. A CDF cannot be created with the host encoding of a different computer. A CDF with any encoding, however, may be read on any computer.[9]

### Host Encoding

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

| | |
|---|---|
| `VAX_ENCODING` | VAX and microVAX computers running VMS. Double-precision floating-point values are encoded in Digital's `D_FLOAT` representation. |
| `ALPHAVMSd_ENCODING` | DEC Alpha computers running OpenVMS. Double-precision floating-point values are encoded in Digital's `D_FLOAT` representation. |
| `ALPHAVMSg_ENCODING` | DEC Alpha computers running OpenVMS. Double-precision floating-point values are encoded in Digital's `G_FLOAT` representation. |
| `SUN_ENCODING` | Sun computers. |
| `SGi_ENCODING` | Silicon Graphics Iris and Power Series computers. |
| `DECSTATION_ENCODING` | DECstation computers. |
| `ALPHAOSF1_ENCODING` | DEC Alpha computers running OSF/1. |
| `IBMRS_ENCODING` | IBM RS6000 series computers. |
| `HP_ENCODING` | HP 9000 series computers. |
| `IBMPC_ENCODING` | IBM PC personal computers. |
| `NeXT_ENCODING` | NeXT computers. |

Note that any host encoding may be read on any computer supported by CDF.

### Network Encoding

Network encoding (`NETWORK_ENCODING`) specifies that variable and attribute entry data values be written to the CDF in the XDR (External Data Representation) format. As values are written to the CDF, the CDF library encodes them into network encoding. Network encoded CDFs are readable on any machine supported by CDF.

[9] This is a change from previous releases of CDF.

### Equivalent Encodings

While an encoding parameter exists for each supported computer, not every encoding is different. The following sections describe which computers use the same encoding for the various data types.

**Character/1-Byte Integer Data Types.** Since each supported computer uses the ASCII character set and orders the bits in a byte the same way, the character and 1-byte integer data types (`CDF_CHAR`, `CDF_UCHAR`, `CDF_BYTE`, `CDF_INT1`, and `CDF_UINT1`) are encoded in the same way on each.

**Multiple-Byte Integer Data Types.** The multiple-byte integer data types (`CDF_INT2`, `CDF_UINT2`, `CDF_INT4`, and `CDF_UINT4`) are encoded in one of two ways: big-endian or little-endian. Big-endian has the least significant byte (LSB) in the highest memory location while little-endian has the LSB in the lowest memory location. The supported computers use big-endian or little-endian as shown in Table 2.3. Network (XDR) encoding uses big-endian encoding for multiple-byte integer data types.

| Big-Endian | Little-Endian |
|---|---|
| Sun | VAX |
| SGi Iris | DECstation |
| IBM RS6000 | IBM PC |
| HP 9000 | DEC Alpha (OSF/1) |
| NeXT | DEC Alpha (OpenVmS) |

Table 2.3: Equivalent Byte Orderings

**Single-Precision Floating-Point Data Types.** The single-precision floating-point encodings on the supported computers are either IEEE 754 floating-point or Digital's `F_FLOAT` floating-point. There are also two different byte orderings for the computers that use IEEE 754 (big-endian and little-endian). The single-precision floating-point encodings for each supported computer are shown in Table 2.4. Network (XDR) encoding uses IEEE 754 (big-endian) encoding for single-precision floating-point data types.

| IEEE 754 (Big Endian) | IEEE 754 (Little Endian) | Digital's `F_FLOAT` |
|---|---|---|
| Sun | DECstation | VAX |
| SGi Iris | IBM PC | DEC Alpha/OpenVMS |
| IBM RS6000 | DEC Alpha/OSF1 | |
| HP 9000 | | |
| NeXT | | |

Table 2.4: Equivalent Single-Precision Floating-Point

**Double-Precision Floating-Point Data Types.** The double-precision floating-point encodings on the supported computers are either IEEE 754 floating-point, Digital's `D_FLOAT` floating-point, or Digital's `G_FLOAT` floating-point. There are also two different byte orderings for the computers that use IEEE 754 (big-endian and little-endian). The double-precision floating-point encodings for each supported computer are shown in Table 2.5. Network (XDR) encoding uses IEEE 754 (big-endian) encoding for double-precision floating-point data types.

| IEEE 754 (Big Endian) | IEEE 754 (Little Endian) | Digital's D_FLOAT | Digital's G_FLOAT |
|---|---|---|---|
| Sun | DECstation | VAX | DEC Alpha/OpenVMS/G |
| SGi Iris | IBM PC | DEC Alpha/OpenVMS/D | |
| IBM RS6000 | DEC Alpha/OSF1 | | |
| HP 9000 | | | |
| NeXT | | | |

Table 2.5: Equivalent Double-Precision Floating-Point

**Performance**

The best performance when accessing (reading or writing) a CDF will occur when that CDF is in the host encoding of the computer being used (and host decoding is in effect — see Section 2.2.8). This is because no encoding or decoding has to be performed by the CDF library. A CDF that must be portable between two or more different types of computers should normally be network encoded. There may be cases, however, where it would be desirable to create a CDF with host encoding (e.g., on a slow machine) and then transfer it to a faster machine for processing or conversion to network encoding. 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 the host encoding of the machine being used (with `CDFconvert`) before being accessed.

### 2.2.8 Decoding

The decoding of a CDF determines how attribute entry and variable data values are passed to a calling application program from the CDF library. The default decoding when a CDF is initially opened is host decoding (the native encoding of the computer being used). When host decoding is in effect, all data values read by an application are immediately ready for manipulation and display. Almost all of your applications will simply use the default of host decoding and not be concerned with selecting a decoding. There are some situations, however, where selecting a different decoding will be advantageous. Some possibilities are as follows:

1. A client/server model where a number of CDFs are maintained on a server computer (in any of the supported encodings). Clients on different type computers could request data from a CDF on the server computer. The server computer would then select a decoding for the CDF based on the client's computer type and then read the data value(s). The value(s) could then be sent directly to the client computer by the server computer without a conversion being necessary by the server computer.

2. If data values were being read from a CDF and written in binary form to a file for use on a different type computer. The proper decoding could be selected for the CDF before any of the data values are read. No conversions would be necessary by the application program.

A CDF's decoding may be selected and reselected at any time after the CDF has been opened and as many times as necessary. A CDF's decoding is selected via the Internal Interface with the `<SELECT_,CDF_DECODING_>` operation. Also, a CDF's decoding does not affect the values that already exist in a CDF or any values subsequently written. A CDF's encoding determines how the values are written to the CDF file(s). Section 2.2.7 describes a CDF's encoding.

The supported decodings correspond to the supported encodings. They are as follows:

| | |
|---|---|
| HOST_DECODING | The data representation of the host computer. |
| NETWORK_DECODING | The External Data Representation (XDR). |
| VAX_DECODING | VAX and microVAX data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation. |
| ALPHAVMSd_DECODING | DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation. |
| ALPHAVMSg_DECODING | DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's G_FLOAT representation. |
| SUN_DECODING | Sun data representation. |
| SGi_DECODING | Silicon Graphics Iris and Power Series data representation. |
| DECSTATION_DECODING | DECstation data representation. |
| ALPHAOSF1_DECODING | DEC Alpha running OSF/1 data representation. |
| IBMRS_DECODING | IBM RS6000 series data representation. |
| HP_DECODING | HP 9000 series data representation. |
| IBMPC_DECODING | IBM PC data representation. |
| NeXT_DECODING | NeXT data representation. |

#### Performance

The best performance when reading a CDF will occur when the CDF's decoding is the same as the CDF's encoding since no conversion will have to be performed by the CDF library. Since host decoding is the only directly usable decoding by an application, CDFs with host encoding will provide the best performance. Care should be taken when selecting the encoding for a CDF.

### 2.2.9 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.

#### 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. |

**NOTE:** When using C on a DEC Alpha running OSF/1, keep in mind that a `long` is 8 bytes and that an `int` is 4 bytes. Use an `int` with the data types `CDF_INT4` and `CDF_UINT4` rather than a `long`.

**NOTE:** When using C on an IBM PC (MS-DOS), keep in mind that an `int` is 2 bytes and that a `long` is 4 bytes. Use a `long` with the data types `CDF_INT4` and `CDF_UINT4` rather than an `int`.

### Floating Point Data Types

`CDF_REAL4 & CDF_FLOAT`
: 4-byte, single-precision floating-point.

`CDF_REAL8 & CDF_DOUBLE`
: 8-byte, double-precision floating-point.

A special case exists with respect to the value `-0.0` (negative floating-point zero). This value is legal on those computers that use the IEEE 754 floating-point representation (e.g., most UNIX-based computers and the IBM PC) but is illegal on VAXes and DEC Alphas running OpenVMS. Attempting to use `-0.0` will result in a reserved operand fault on a VAX and a high performance arithmetic fault on a DEC Alpha running OpenVMS. A warning is returned whenever `-0.0` is read by an application on a VAX or DEC Alpha running OpenVMS. The CDF library can be put into a mode where `-0.0` will be converted to `0.0` when detected (see Section 2.1.2). If `-0.0` is not being converted to `0.0`, the CDF toolkit programs are designed to display `-0.0` in all cases. This includes those computers that normally suppress the negative sign.

### 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.

### 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`.[10] `CDF_EPOCH` values are the number of milliseconds since

[10] 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.

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 that parse, encode, compute, and break down CDF_EPOCH values. These functions are described in Section 4.9 for C applications and in Section 5.10 for Fortran applications.

#### 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.

### 2.2.10 Limits

Limits within a CDF are defined in the appropriate include files: cdf.h for C applications and cdf.inc for Fortran applications. The following limits exist:[11]

| | |
|---|---|
| CDF_MAX_DIMS | The maximum number of dimensions that rVariables/zVariables 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. Contact CDF User Support if that becomes necessary.

[11] Previous releases of CDF limited the number of variables a CDF could contain. That limit has been eliminated.

## 2.3 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. Two types of variables are supported: rVariables and zVariables.[12] The main difference is that all rVariables in a CDF have the same dimensionality whereas zVariables can have differing dimensionalities. In the following sections differences between the two types will be noted where appropriate.

### 2.3.1 Accessing

The Standard Interface deals exclusively with rVariables. No access to zVariables is provided. The Internal Interface may be used to access either rVariables or zVariables.

### 2.3.2 Opening

The CDF library automatically opens the variable files in a multi-file CDF as the variables are accessed. An application never has to concern itself with opening variables.

The opening of variables does not apply to single-file CDFs since individual files do not exist for each variable.

### 2.3.3 Closing

The CDF library automatically closes the variable files in a multi-file CDF when the CDF itself is closed by an application.[13] Variable files are also closed automatically by the CDF library as other variables are accessed if insufficient file pointers exist to keep all of the variables open at once. This would be due to an open file quota enforced by the operating system being used.

A case also exists where it may be beneficial for an application to close a variable in a multi-file CDF. Since each open variable file uses some number of cache buffers, a large amount of system memory could be in use (see Section 2.1.4). This may not be a problem on VAX or UNIX machines but could result in a program crashing on an MS-DOS machine. If memory is limited, an application may want to close variables after they have been accessed in order to minimize the total number of cache buffers being used. In a C application rVariables are closed using either the `CDFvarClose` function (Standard Interface) or the `<CLOSE_,rVAR_>` operation of the `CDFlib` function (Internal Interface). zVariables are closed using the `<CLOSE_,zVAR_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application rVariables are closed using either the `CDF_var_close` subroutine (Standard Interface) or the `<CLOSE_,rVAR_>` operation of the `CDF_lib` subroutine (Internal Interface). zVariables are closed using the `<CLOSE_,zVAR_>` operation of the `CDF_lib` subroutine (Internal Interface).

The closing of variables does not apply to single-file CDFs since individual files do not exist for each variable.

[12] The letters "r" and "z" don't stand for anything in particular. "r" sort of stands for "regular" since rVariables have always been supported by CDF.

[13] It is required that an application close a CDF before exiting.

### 2.3.4 Naming

Each variable in a CDF has a unique name. This applies to rVariables and zVariables together (i.e., an rVariable cannot have the same name as a zVariable). Variable names are case sensitive regardless of the operating system being used and may consist of up to `CDF_VAR_NAME_LEN` 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.

### 2.3.5 Numbering

The rVariables in a CDF are numbered consecutively starting at one (1) for Fortran applications and starting at zero (0) for C applications. Likewise, the zVariables in a CDF are numbered consecutively starting at one (1) for Fortran applications and starting at zero (0) for C applications. The CDF library assigns variable numbers as the variables are created.

### 2.3.6 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 rVariable in a CDF has the same dimensionality. An array of values exists for each rVariable at each record in a CDF. The values may not be physically stored but may be virtual (see Sections 2.3.7, 2.3.9, and 2.3.10).

A zVariable may have a dimensionality which is different from that of the rVariables and the other zVariables. An array of values exists for each zVariable at each record in a CDF. As with rVariables the values may not be physically stored but may be virtual. zVariables are intended for use in those situations where using an rVariable would waste disk space or not logically make sense.

A variable array having two or more dimensions also contains subarrays. For instance, in a 3-dimensional array with dimension sizes `[10,20,30]`, each 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.

### 2.3.7 Records

A CDF record is a set of variable arrays, one per rVariable and one per zVariable in the CDF. The variable arrays in a particular record are generally related to each other in some way (often time). This does not have to be the case and is not enforced by the CDF library in any way. A variable record is simply the corresponding variable array within a CDF record.

Physical variable records are actually stored in the CDF file(s). 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 and have the same values as the first record (only the first record is physically stored). If a record has not yet been written to that variable, then all of its records are virtual and contain the pad value for that variable.

2. If a variable's values do vary from record to record (record variance of `VARY`), then the records beyond the last record actually written are virtual and contain the pad value for that variable.

Record variance is described in Section 2.3.9.

The maximum record written is maintained by the CDF library for each variable in the CDF. The "maximum CDF record" is simply the maximum rVariable record written (of all the rVariables). This quantity is available through the Standard Interface when inquiring about a CDF. Because the Standard Interface does not allow access to zVariables, zVariables are not considered when determining the "maximum CDF record." The "maximum CDF record" would be used by applications dealing only with rVariables. The maximum record written for each rVariable and zVariable is available through the Internal Interface.

Figure 2.1 illustrates the relationships between physical and virtual variable records. Variable 1 has five records that were physically written. Only two records were physically written to variable 2 so the following records are virtual (containing the pad value for that variable). Only one record can be physically written to variable 3 because its record variance is `NOVARY`. The other 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 pad value for that variable. Likewise, since no records have been written to variable 5, all of its records are also virtual and contain the pad value for that variable. Note that if these are rVariables, the "maximum CDF record" would be five (when displayed by a toolkit program or in a Fortran application — it would be four in a C application).

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 beyond the current maximum record for a variable is written, the intervening records will be physically written and contain the pad value for that variable. Also, when one or more values are written to a new physical record, the entire record is physically written with the pad value for the variable being used for the unspecified values. Variable pad values are described in Section 2.3.16.

### Numbering

The record numbers in a CDF are numbered starting at one (1) for Fortran applications and starting at zero (0) for C applications.

### Initial Records

The Internal Interface may be used to specify an initial number of records for a variable. The pad 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 pad value for

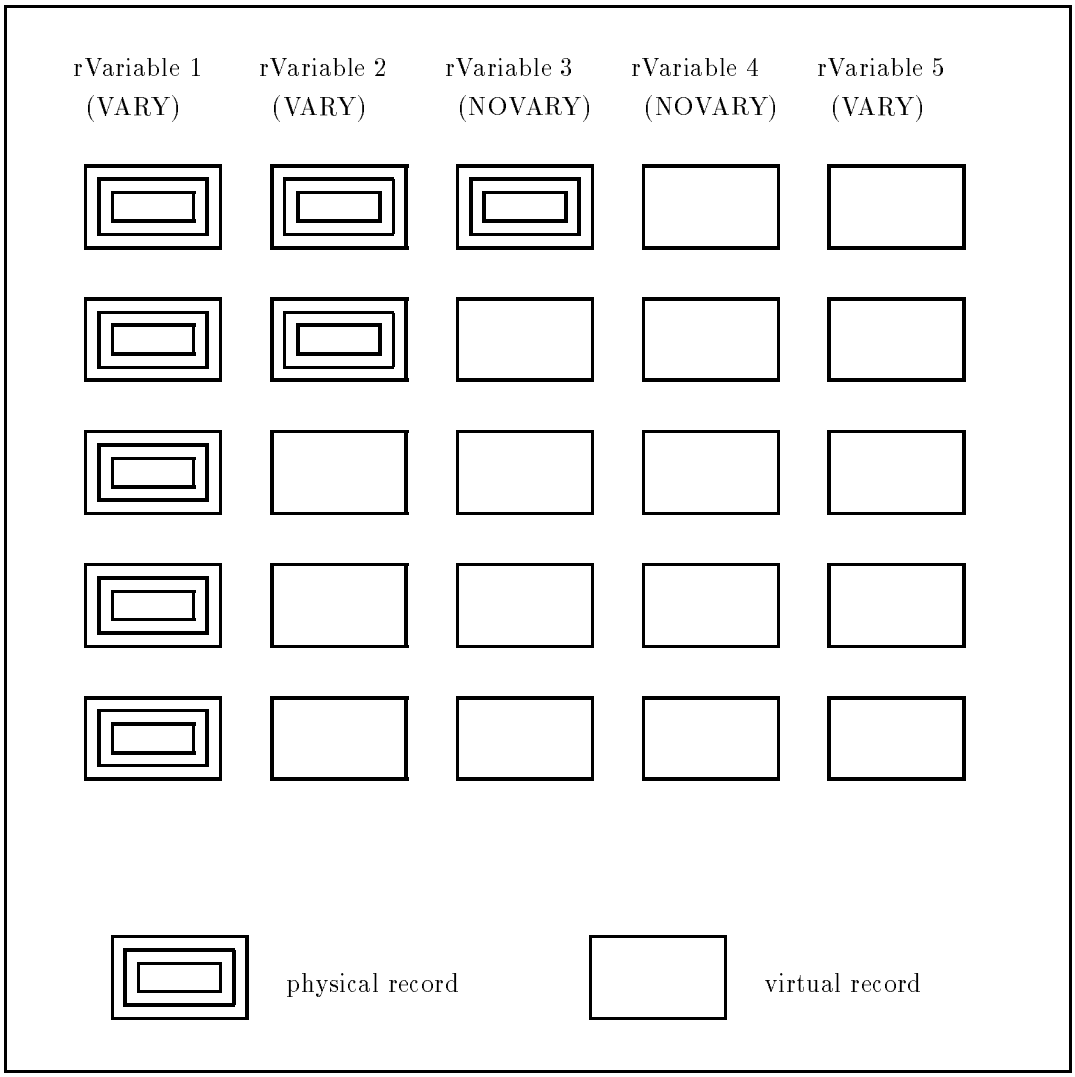


Figure 2.1: Physical vs. Virtual Records

the variable's data type will be used unless a pad value is specified for the variable (also using the Internal Interface). The pad value for a variable must be specified before the number of initial records is specified.

Specifying a number of initial records for a variable would usually be done only for a CDF with the single-file format. Because the records would be allocated contiguously within the CDF file, the indexing scheme (see Section 2.2.6) 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 pad value, the actual data values would be written by the application. Note that for NRV variables the number of initial records must always be specified as one (1). This is because at most only one physical record will ever actually be written.

In a C application initial records are written to rVariables/zVariables using the `<PUT_,rVAR_INITIALRECS_>`

and <PUT_,zVAR_INITIALRECS_> operations of the CDFlib function (Internal Interface). In a Fortran application initial records are written using the same operations of the CDF_lib subroutine (Internal Interface).

#### 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 (see Section 2.2.6) 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 only has an effect on CDFs with the single-file format and for variables that are not NRV (since at most only one physical record will ever be written for an NRV variable).

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.

In a C application extend records are specified for rVariables/zVariables using the <PUT_,rVAR_INITIALRECS_> and <PUT_,zVAR_INITIALRECS_> operations of the CDFlib function (Internal Interface). In a Fortran application extend records are specified using the same operations of the CDF_lib subroutine (Internal Interface).

### 2.3.8 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 is true.

1. Values have not yet been written to the variable (including an explicitly written pad value — see Section 2.3.16).

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 2.2.9.

#### Data Type

The supported data types are described in Section 2.2.9. 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).[14] An array of elements per variable value is not allowed for non-character string data types.

### 2.3.9 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 (i.e., if a record more than one beyond the maximum record is written, the intervening records are also physically written and contain pad values). If a record is read beyond the maximum record written to a variable, the pad 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 is physically written to the variable. Each record contains the same values (including virtual records beyond the first record). Variables of this type are referred to as non-record-variant (NRV). |

Section 2.3.7 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 pad value may have been specified however.)

### 2.3.10 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. |

[14] A future release of CDF may relax this restriction. Note that for attribute entries, more than one element is allowed for any data type.

| | |
|---|---|
| `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 is the same (including virtual values/subarrays beyond the first value/subarray). |

Figure 2.2 illustrates the effect of dimension variances on a variable with 2-dimensional arrays (for a particular record). For 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). The same is true 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).

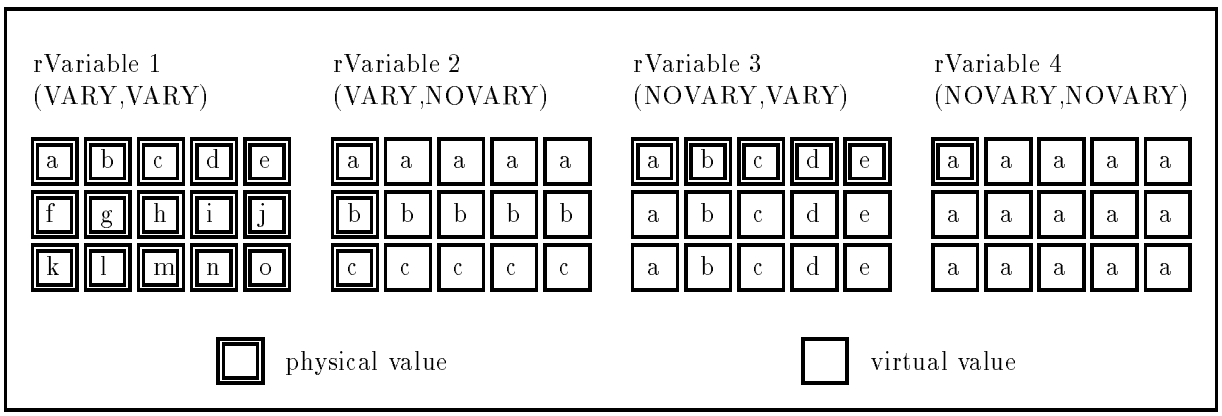


Figure 2.2: Physical vs. Virtual Dimensions

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 pad value may have been specified, however.)

### 2.3.11 Majority

The variable 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 an rVariable 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 variable hyper read, the values placed in the buffer by the CDF library will be in the variable majority of the CDF. The application must process the values according to that majority.

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

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

3. When single value reads/writes are performed, the majority could have an effect. The CDF library uses a caching scheme to optimize[15] the random access to variable values. If all of the values of a record are to be read/written, there may be an increase in performance if the values are accessed with (rather than against) the majority. For example, if the majority is row-major, increment the last index the fastest.

4. When performing a multiple variable read/write, the full-physical records in the buffer will/must be in the variable majority of the CDF.

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 pad values may have been specified, however.)

### 2.3.12 Single Value Access

Single value access allows only one value to be read from or written to a variable with a single call to the CDF library. Two parameters are specified when performing a single value read/write:

| | |
|---|---|
| `record number` | The record number at which to perform the access. |
| `dimension indices` | The indices within the record at which to perform the access. |

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

Single value access is sensitive to the record and dimension variances of a variable. For instance, if a variable has a record variance of `NOVARY` (with one record written) and a value is read from the fourth record, the CDF library will actually read the value from the first record (the record that is physically stored). If a value were written to the fourth record, the CDF library would actually write the value to the first record (the only record that actually physically exists). If the record variance is `VARY`, the values are written to the actual records. (The physical records are the same as the virtual records.) The same applies to any dimension variances that are `NOVARY`. When a set of indices is specified for a single value read/write, the

[15] Since an application knows how it will be accessing a variable, it knows best how to optimize the caching scheme used. See Section 2.1.4 for details on how an application can control the CDF library caching scheme.

index for a dimension whose variance is NOVARY is forced to the first index regardless of the actual index specified for that dimension (see Section 2.3.10).

In a C application single value access for rVariables is performed using either the CDFvarGet and CDFvarPut functions (Standard Interface) or the <GET_,rVAR_DATA_> and <PUT_,rVAR_DATA_> operations of the CDFlib function (Internal Interface). Single value access for zVariables is performed using the <GET_,zVAR_DATA_> and <PUT_,zVAR_DATA_> operations of the CDFlib function (Internal Interface). In a Fortran application single value access for rVariables is performed using either the CDF_var_get and CDF_var_put subroutines (Standard Interface) or the <GET_,rVAR_DATA_> and <PUT_,rVAR_DATA_> operations of the CDF_lib subroutine (Internal Interface). Single value access for zVariables is performed using the <GET_,zVAR_DATA_> and <PUT_,zVAR_DATA_> operations of the CDF_lib subroutine (Internal Interface).

### 2.3.13 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. |
| 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 sensitive to the record and dimension variances of a variable. For instance, if a variable has a record variance of NOVARY (with one record written) 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 that 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 variable array with sizes `[2,4]`, row majority, a record variance of `VARY`, 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 pad value if they did not already exist. If they did already exist, they would not be affected.

In a C application hyper reads and writes for rVariables are performed using either the CDFvarHyperGet and CDFvarHyperPut functions (Standard Interface) or the <GET_,rVAR_HYPERDATA_> and <PUT_,rVAR_HYPERDATA_> operations of the CDFlib function (Internal Interface). Hyper reads and writes for zVariables are performed using the <GET_,zVAR_HYPERDATA_> and <PUT_,zVAR_HYPERDATA_> operations of the CDFlib function (Internal Interface). In a Fortran application hyper reads and writes for rVariables are performed using either the CDF_var_hyper_get and CDF_var_hyper_put subroutines (Standard Interface) or the <GET_,rVAR_HYPERDATA_> and <PUT_,rVAR_HYPERDATA_> operations of the CDF_lib subroutine (Internal Interface). Hyper reads and writes for zVariables are performed using the <GET_,zVAR_HYPERDATA_> and <PUT_,zVAR_HYPERDATA_> operations of the CDF_lib subroutine (Internal Interface).

### 2.3.14 Sequential Access

Sequential access provides a way to sequentially read/write the values physically stored for a variable. To use sequential access, a starting value must first be selected by specifying a record number and dimension indices. This selects the "current sequential value." A sequential read will return the value at the current sequential value and then automatically increment the current sequential value to the next value. Likewise, a sequential write will store a value at the current sequential value and then increment the current sequential value to the next value. Sequential reads are allowed until the end of the physical records has been reached (not the end of the virtual records [they never end]). Sequential reading will increment to the beginning of the next physical record if necessary. Sequential writing can be used to extend the physical records for a variable (as well as to overwrite existing values).

#### Example (Fortran application)

Assume a 2-dimensional array with sizes [2,3], column majority, a record variance of VARY, dimension variances of [VARY,VARY], nine (9) physical records written, and that the current sequential value 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 sequential value caused physical value 7(1,2) to actually be selected (because the first dimension variance is NOVARY).

In a C application sequential access for rVariables is performed using the <GET_,rVAR_SEQDATA_> and <PUT_,rVAR_SEQDATA_> operations of the CDFlib function (Internal Interface). Sequential access for zVariables is performed using the <GET_,zVAR_SEQDATA_> and <PUT_,zVAR_SEQDATA_> operations. In a Fortran application sequential access is performed using the same operations of the CDF_lib subroutine (Internal Interface).

### 2.3.15 Multiple Variable Access

Multiple variable access allows an application to read from or write to multiple variables in a single operation. Multiple variable access works on either the rVariables or the zVariables of a CDF — not a mixture of the two. Up to all of the rVariables/zVariables may be accessed with a single call to the CDF library. For each variable specified in a multiple variable access, a full-physical record for that variable will be read/written. A full-physical record consists of all of the values exactly as they are physically stored in each variable record (the physical values). Virtual values do not apply when performing a multiple variable access (see Section 2.3.10). Three parameters are specified when performing a multiple variable read/write.

| | |
|---|---|
| variable count | The number of rVariables/zVariables that are being accessed. |
| variable list | The rVariables/zVariables being accessed (specified by number). |
| record numbers | The record numbers at which the reads/writes will take place. For rVariables the record numbers must all be the same. For zVariables the record numbers can vary (but for most applications will all be the same). |

Multiple variable access is sensitive to the record variances of the variables being accessed. (Dimension variances do not apply since full-physical records are being read/written.) If a variable has a record variance of NOVARY, then a read/write to that variable will always occur at the first record regardless of the actual record number specified (since at most only one physical record will ever exist). If the record variance were VARY, the reads/writes would take place at the actual record numbers specified.

For a multiple variable write operation an application must place into a memory buffer each of the full-physical records to be written. The order of the full-physical records must correspond to the order of the list of variables specified, and the memory buffer must be contiguous — there can be no gaps between the full-physical records. This memory buffer is then passed to the CDF library which scans through the buffer writing the full-physical records to the corresponding variables.

Likewise, for a multiple variable read operation the CDF library places into a memory buffer provided by the application the full-physical records read. The order of the full-physical records will correspond to the order of the list of variables specified and the full-physical records will be contiguous. The application must then process the buffer as needed.

Care must be used when generating and processing the memory buffer containing the full-physical records. If C struct objects or Fortran STRUCTURE variables are being used, it may be necessary to order the variables being read/written such that there are no gaps between elements of the structures (assuming you are defining structures containing one element per full-physical record where an element is a scalar variable or an array depending on the corresponding variable definition). On some computers the C and Fortran compilers will

place gaps between the elements of these structures so that memory alignment errors are not generated when the elements are accessed. In general, defining the structures so that "larger" data types are before "smaller" data types should result in no gaps (e.g., the Fortran `REAL*8` data type is "larger" than a `INTEGER*2`, which is "larger" than a `BYTE`). The list of variables would be adjusted accordingly.

The variable majority must also be considered when performing a multiple variable read/write since full-physical records are being accessed. The majority of the values in the full-physical records retrieved from/placed into the memory buffer must be the same as the variable majority of the CDF.

For example, consider a column-major CDF containing the following three zVariables (as well as others):

| zVariable Name | Data Specification | Dimensionality | Variances |
|---|---|---|---|
| zVar1 | CDF_INT2/1[16] | 0:[] | T/ |
| zVar2 | CDF_CHAR/7 | 1:[5] | T/T |
| zVar3 | CDF_REAL8/1 | 2:[2,4] | T/TT |

If a Fortran application were to perform a multiple variable read on these three zVariables, it could define a `STRUCTURE` to receive the physical records as follows:

```
STRUCTURE /inputStruct/
  REAL*8      zVar3values(2,4)
  INTEGER*2   zVar1value
  CHARACTER*7 zVar2values(5)
END STRUCTURE
```

Note that because a full-physical record for the zVariable `zVar2` is an odd number of bytes it would most likely cause a gap in the `STRUCTURE` if not placed at the end (on some computers). An approach that would work on all computers would be to use `EQUIVALENCE` statements as follows:

```
INTEGER*2   zVar1value
CHARACTER*7 zVar2values(5)
REAL*8      zVar3values(2,4)
BYTE        buffer(101)
EQUIVALENCE (zVar3values,buffer(1))
EQUIVALENCE (zVar1value,buffer(65))
EQUIVALENCE (zVar2values,buffer(67))
```

The `EQUIVALENCE` statements ensure that the full-physical records will be contiguous. In each of the above examples, the order of the zVariables would be `zVar3`, `zVar1`, `zVar2`.

C applications must also be concerned with the ordering of full-physical records in the memory buffer. Even if a `void` memory buffer is used with type casting to access individual values, the alignment of the values in the memory buffer is important (on some computers).

[16] This notation is used throughout this document. The data type is before the slash and the number of elements is after the slash. In this case the data type is (`CDF_INT2`) and the number of elements is one (1).

### 2.3.16 Variable Pad Values

Variable pad[17] values are used in several situations.

1. When the first value is written to a new record, the other values in that record will contain the pad value. This also applies to hyper writes if less than the entire record is written. (The unwritten values will contain the pad value.)
2. When a new record is written that is more than one record beyond the last record already written, the intervening records will contain pad 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, pad values will be returned.

The pad 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 pad value will be used. The pad value may be changed at any time (and any number of times) and will be in effect for all subsequent operations.

The default pad value for each data type is as follows:[18]

```
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)
```

## 2.4 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.

### 2.4.1 Naming

Each attribute in a CDF has a unique name. Attribute names are case sensitive regardless of the operating system being used and may consist of up to `CDF_ATTR_NAME_LEN` printable characters (including blanks).

[17] These were previously known as fill values but were renamed to avoid confusion with the `FILLVAL` attribute (an NSSDC standard attribute).

[18] These default pad values can be changed by your system manager when the CDF distribution is built.

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.

### 2.4.2 Numbering

The attributes in a CDF are numbered consecutively starting at one (1) for Fortran applications and starting at zero (0) for C applications. The CDF library assigns attribute numbers as the attributes are created.

### 2.4.3 Attribute Scopes

Attribute scopes declare the intended purpose of an attribute. Global scope attributes (gAttributes) describe some aspect of the entire CDF. Variable scope attributes (vAttributes) describe some property of each variable.

An attribute's scope exists to assist in the interpretation of its entries by CDF toolkit programs and user applications (e.g., entries of a vAttribute should correspond to variables). The CDF library also places some restrictions on the operations that may be performed on an attribute of a particular scope.[19] These restrictions consist of the following:

1. A zEntry operation may not be performed on a gAttribute.
2. While in zMode, only zEntry operations may be performed on vAttributes (see Section 2.1.2).

All other operations involving attributes and their entries remain available.

#### Assumed Scopes

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 CDF toolkit programs encounter an assumed scope, they cannot always determine the correct scope. In those cases in which a new CDF is being created, they 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) or the `CDFedit` toolkit program (see Section 3.13). The assumed scopes may also be corrected by a user application via the CDF library.

[19] This was not necessarily the case in previous releases of CDF. These new restrictions should not, however, cause any conflicts with existing applications.

### 2.4.4 Attribute Entries

Attribute entries are used to actually store metadata. Each attribute in a CDF may have zero or more associated entries. For vAttributes two types of entries are supported: rEntries and zEntries. rEntries describe some property of the corresponding rVariable, and zEntries describe some property of the corresponding zVariable. Note that an entry does not have to exist for each variable in the CDF. For gAttributes only one type of entry is supported and is referred to as a gEntry. The gEntries are independent of anything else in the CDF and have meaning only to the application. Note that gEntries are sometimes referred to simply as "entries."

#### Accessing

The Standard Interface deals exclusively with rEntries (for vAttributes) and gEntries (for gAttributes). No access to zEntries is provided. The Internal Interface may be used to access any type of attribute entry.

#### Numbering

The rEntries and zEntries for a vAttribute and the gEntries for a gAttribute are numbered starting at one (1) for Fortran applications and starting at zero (0) for C applications. For vAttributes the entry numbers are in fact the variable numbers of the variables being described. rEntries correspond to rVariables and zEntries correspond to zVariables. For gAttributes the gEntry numbers have meaning only to the application.

The entry numbers used need not be contiguous (as are variable and attribute numbers). An application may choose to write any combination of entries for a particular attribute (keeping in mind that the entry numbers used for a vAttribute correspond to the existing variables).

#### Data Specification

Each entry for an attribute has a data specification and an associated value. A data specification consists of a data type and a number of elements of that data type. The supported data types are described in Section 2.2.9. The entries for an attribute may have any combination of data specifications.

For character string data types the number of elements is the number of characters in the string. For example, if a gEntry value for a gAttribute 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 the number of elements is the size of an array of the data type. For example, if a zEntry value of a vAttribute 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).

## 2.5 NSSDC Standard

CDFs generated for use by NSSDC applications (e.g., 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. The NSSDC standard does not deal with zVariables or zEntries (for vAttributes).

### 2.5.1 Standard Attributes

There are several attributes that are required to exist in a CDF if it is to conform to the NSSDC standard. These required attributes are shown in Table 2.6. Several optional attributes are also shown.

| | Name | Data Type | Number of Elements[a] |
|---|---|---|---|
| Required gAttributes: | | | |
| | TITLE | Character (CDF_CHAR) | 50 |
| Required vAttributes: | | | |
| | 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 gAttributes: | | | |
| | TEXT | Character (CDF_CHAR) | 80 |
| | MODS | Character (CDF_CHAR) | 80 |
| Optional vAttributes: | | | |
| | SCALEMIN | (same as variable) | (same as variable) |
| | SCALEMAX | (same as variable) | (same as variable) |
| | MONOTON[b] | Character (CDF_CHAR) | 8 |
| | CATDESC | Character (CDF_CHAR) | 40 |
| | FILLVAL | (same as variable) | (same as variable) |

[a] If the data type is CDF_CHAR, blank pad the value on the right to the required number of elements (characters).
[b] We strongly suggest using the MONOTON attribute.

Table 2.6: NSSDC Standard CDF Attributes

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

**TITLE** A title for the CDF data set. Only one gEntry should be written (gEntry number zero [0] for C applications, gEntry number one [1] for Fortran applications).

**FIELDNAM** The field name for an rVariable. One rEntry per rVariable should be written.

**VALIDMIN** The minimum valid value for an rVariable. Values below this minimum are treated as invalid or fill data. One rEntry per rVariable should be written.

| | |
|---|---|
| VALIDMAX | The maximum valid value for an rVariable. Values above this maximum are treated as invalid or fill data. One rEntry per rVariable should be written. |
| UNITS | The units of measurement for an rVariable. One rEntry per rVariable should be written. |
| FORMAT | The Fortran format specification to use when displaying the values for an rVariable. One rEntry per rVariable should be written. |
| TEXT | Documentation text about the CDF. Any number of gEntries may be written. |
| MODS | A modification history for the CDF. Each time the CDF is modified, a new gEntry would be added with an explanation of the changes. |
| SCALEMIN | The scale minimum for an rVariable. This is used when scaling the rVariable's values for display. If omitted, the VALIDMIN vAttribute rEntry will be used. One rEntry per rVariable should be written. |
| SCALEMAX | The scale maximum for an rVariable. This is used when scaling the rVariable's values for display. If omitted, the VALIDMAX vAttribute rEntry will be used. One rEntry per rVariable should be written. |
| MONOTON | The monotonicity of an rVariable. 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 an rVariable's monotonicity can greatly improve the efficiency of certain NSSDC applications (namely, the NSSDC Graphics System). One rEntry per rVariable should be written. |
| CATDESC | A catalog description for an rVariable. This is used for cataloging purposes by some NSSDC applications. One rEntry per rVariable should be written. |
| FILLVAL | The fill value for an rVariable. This value should not be confused with the pad value that may be specified for an rVariable (with the Internal Interface) that the CDF library actually uses when writing records. The values should agree but the FILLVAL vAttribute is not used by the CDF library in any way. It is, however, used by CDF applications (including the CDF toolkit). One rEntry per rVariable should be written. |

### 2.5.2 Standard Variables

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

The optional rVariable EPOCH is used by many NSSDC applications to store date/time values. Section 2.2.9 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.

# Chapter 3

# CDF Toolkit Reference

## 3.1 Introduction

The CDF toolkit is a set of utility programs that 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 available 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 (e.g., `'/disk3/sst*'`).
6. On VMS systems and DEC Alphas running OpenVMS, qualifiers begin with a '/'. On UNIX and MS-DOS systems, qualifiers begin with a '-'. When describing qualifiers, both notations may be shown (e.g., `/[NO]LOG`, `-[no]log`). Use only the notation appropriate for the operating system being used.
7. On MS-DOS systems the executable names may be different from the names shown in this chapter (file names are limited to 8.3 characters). Where the names differ, the actual name will be noted.

   If you add the directory containing the toolkit executables to your path, you will have to use the 8-character (or fewer) names. If you use a command aliasing program, you could specify the aliases to be the names shown in this chapter with each pointing to the corresponding executable file name.

8. On UNIX systems all parameters/qualifiers entered at the command line are case sensitive. On VMS, OpenVMS, and MS-DOS systems parameters/qualifiers are not case sensitive. Note that variable names (when entered on the command line) are always case sensitive regardless of the operating system being used.

Most of the toolkit programs allow a user to select which types of status codes from the CDF library should be reported: errors, warnings, and/or informationals. This selection is usually made using the `/REPORT`,`-report` qualifier. Note that disabling the reporting of error status codes may have undesirable side effects. (An error may have occurred, but you won't know about it.)

In the following sections the available qualifiers and options for each of the toolkit programs will be presented. The default settings for these qualifiers and options will not be shown since they can be configured for a particular CDF distribution. The file `config.h` contains the default settings for the toolkit programs. Use `CDFinquire` with the `/ID`,`-id` qualifier to determine these defaults.

## 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 rVariable files and then the zVariable files (if either exist in a multi-file CDF) in numerical order.

### 3.2.2 Executing the CDFdir Program

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

Usage:

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 saved 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] [/[NO]PERCENT]
             [/[NO]ETC] [/[NO]NEG2POSFP0] [/ZMODES=(<mode1>,<mode2>)]
             [/[NO]LOCATION] [/REPORT=(<types>)] <cdf-spec-1> <cdf-spec-2>
```

UNIX:

```
% cdfcompare [-[no]log] [-[no]attr] [-[no]var] [-[no]number] [-[no]percent]
             [-[no]etc] [-[no]neg2posfp0] [-zmodes "<mode1>,<mode2>"]
             [-[no]location] [-report "<types>"] <cdf-spec-1> <cdf-spec-2>
```

MS-DOS:[1]

```
> cdfcompare [-[no]log] [-[no]attr] [-[no]var] [-[no]number] [-[no]percent]
             [-[no]etc] [-[no]neg2posfp0] [-zmodes "<mode1>,<mode2>"]
             [-[no]location] [-report "<types>"] <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.

`/[NO]PERCENT`, `-[no]percent`

Specifies whether or not the percentage of a variable's values compared is displayed during the comparison of that variable. Message logging must also be enabled.

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

`/[NO]VAR`, `-[no]var` Specifies whether or not variables are to be compared. Note that an rVariable will never be compared with a zVariable.

[1] On MS-DOS systems the executable is named `CDFCMP.EXE`.

/[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.

/[NO]ETC, -[no]etc

Specifies whether or not differences transparent to an application will be displayed. These would consist of the version/release/increment of the creating CDF library, format, encoding, etc.

/ZMODES=(<mode1>,<mode2>), -zmodes "<mode1>,<mode2>"

Specifies the zModes that should be used with the CDF(s) being compared. <mode1> applies to the first CDF specification and <mode2> applies to the second CDF specification. The zModes may each be one of the following:

- 0 Indicates that zMode should be disabled.
- 1 Indicates that zMode/1 should be used. (The dimension variances of rVariables will be preserved.)
- 2 Indicates that zMode/2 should be used. (The dimensions of rVariables having a variance of NOVARY [false] are removed.)

/[NO]NEG2POSFP0, -[no]neg2posfp0

Specifies whether or not -0.0 is converted to 0.0 by the CDF library when read from a CDF. (The CDF is not modified.) -0.0 is an illegal floating point value on VAXes and DEC Alphas running OpenVMS.

/[NO]LOCATION, -[no]location

Specifies whether or not the locations of variable value differences are displayed. The locations are displayed in the form
<record-number>:[<index1>,<index2>,...,<indexN>]

/REPORT=(<types>), -report "<types>"

Specifies the types of return status codes from the CDF library that should be reported/displayed. <types> is a comma-separated list of zero or more of the following symbols: errors, warnings, or informationals.

Example(s):

VMS:

```
$ CDFCOMPARE GISS_WETL GISS_WETL1
$ CDFCOMPARE/LOG/NOATTR/NUMBER/REPORT=(ERRORS) GISS_WETL CDF$SMPL:GISS_WETL
$ CDFCOMPARE/NOVAR/NOETC/ZMODES=(1,2) NCDS$SMPL: NCDS$DATA:
```

UNIX:

```
% cdfcompare giss_wetl giss_wetl1
```

```
% cdfcompare -log -noattr -number -report "errors" giss_wetl ../giss_wetlx
% cdfcompare -novar -noetc -zmodes "1,2" /user5/CDFs /user6/CDFs
```

MS-DOS:

```
> cdfcompare gisswetl c:\gisswetl
> cdfcompare -log -noattr -number -report "errors" gisswetl ..\..\giswetlx
> cdfcompare -novar -noetc -zmodes "1,2" 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 2.2.6).
3. The data encoding of the CDF may be changed (see Section 2.2.7).
4. The variable majority of the CDF may be changed (see Section 2.3.11).

### 3.4.2 Executing the CDFconvert Program

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

Usage:

VMS:

```
$ CDFCONVERT [/SKELETON=<cdf-path>] [/[NO]LOG]
             [/[NO]PERCENT] [/REPORT=(<types>)]
```

```
                <src-cdf-spec> [/ZMODE=<mode>] [/[NO]NEG2POSFP0]
                <dst-cdf-spec> [/SINGLE | /MULTI] [/ROW | /COLUMN]
                               [/HOST | /NETWORK] [/[NO]DELETE]
```

UNIX:

```
% cdfconvert [-skeleton <cdf-path>] [-[no]log]
            [-[no]percent] [-report "<types>"]
            <src-cdf-spec> [-zmode <mode>] [-[no]neg2posfp0]
            <dst-cdf-spec> [-single | -multi] [-row | -column]
                           [-host | -network] [-[no]delete]
```

MS-DOS:[2]

```
> cdfconvert [-skeleton <cdf-path>] [-[no]log]
             [-[no]percent] [-report "<types>"]
             <src-cdf-spec> [-zmode <mode>] [-[no]neg2posfp0]
             <dst-cdf-spec> [-single | -multi] [-row | -column]
                            [-host | -network] [-[no]delete]
```

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 that 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] On MS-DOS systems the executable is named `CDFCVT.EXE`.

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.

`/[NO]PERCENT`, `-[no]percent`

Specifies whether or not the percentage of a variable's values converted is displayed during the conversion of that variable. Message logging must also be enabled.

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

Specifies whether or not a destination CDF is deleted if it already exists.

`/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 a destination CDF will be the same as that of the source CDF.

`/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 a destination CDF will be the same as that of the source CDF.

`/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 a destination CDF will be the same as that of the source CDF.

`/ZMODE=<mode>`, `-zmode <mode>`

Specifies the zMode that should be used with the source CDF(s). `<mode>` may be one of the following:

`0` Indicates that zMode should be disabled.

`1` Indicates that zMode/1 should be used. (The dimension variances of rVariables will be preserved.)

`2` Indicates that zMode/2 should be used. (The dimensions of rVariables having a variance of `NOVARY` [false] are removed.)

Note that using zMode/1 or zMode/2 on a source CDF that contains

rVariables will produce a destination CDF containing only zVariables. The zMode "view" provided for the source CDF is written to the destination CDF during the conversion.

`/[NO]NEG2POSFP0`, `-[no]neg2posfp0`

Specifies whether or not `-0.0` is converted to `0.0` by the CDF library when read from a CDF. (The CDF is not modified.) `-0.0` is an illegal floating point value on VAXes and DEC Alphas running OpenVMS.

`/REPORT=(<types>)`, `-report "<types>"`

Specifies the types of return status codes from the CDF library that should be reported/displayed. `<types>` is a comma-separated list of zero or more of the following symbols: `errors`, `warnings`, or `informationals`.

Example(s):

VMS:

```
$ CDFCONVERT CDF$SMPL:TEMPLATE0 TEMPLATE0X
$ CDFCONVERT/LOG/REPORT=(ERRORS) 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 -report "errors" ../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 -report "errors" ..\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 2.4.3 explains attribute scopes and how assumed scopes can arise.

**NOTE:** This program is not a permanent one — it may be deleted from a future release of the CDF distribution. Version 1 CDFs should be converted to Version 2 CDFs as soon as possible. Any assumed attribute scopes should then be corrected using this program.

### 3.5.2 Executing the CDFscope Program

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

Usage:

VMS:

```
$ CDFSCOPE [/[NO]CONFIRM] [/[NO]LOG] [/REPORT=(<types>)] <cdf-path>
```

UNIX:

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

MS-DOS:

```
> cdfscope [-[no]confirm] [-[no]log] [-report "<types>"] <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.

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

`/REPORT=(<types>), -report "<types>"`
Specifies the types of return status codes from the CDF library that should be reported/displayed. `<types>` is a comma-separated list of zero or more of the following symbols: `errors`, `warnings`, or `informationals`.

Example(s):

VMS:

```
$ CDFSCOPE DUA5:[CDF.TEST]TOGA_ECMWF
$ CDFSCOPE/CONFIRM/NOLOG/REPORT=(ERRORS) TOGA_ECMWF
```

UNIX:

```
% cdfscope /disk5/cdf/toga_ecmwf
% cdfscope -confirm -nolog -report "errors" toga_ecmwf
```

MS-DOS:

```
> cdfscope a:\dir5\cdf\toga
> cdfscope -confirm -nolog -report "errors" 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. Both rVariables and zVariables are analyzed. For each variable it determines the actual minimum and maximum values (in all of the variable records), the minimum and maximum values within the valid range (if the `VALIDMIN` and `VALIDMAX` vAttributes and corresponding entries are present in the CDF), and the monotonicity. An option exists to allow fill values (specified by the `FILLVAL` vAttribute) to be ignored 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 CDF with the 2-dimensional rVariables shown in Table 3.1.

| rVariable | Record Variance | Dimension Variances |
|---|---|---|
| EPOCH | VARY | NOVARY,NOVARY |
| LATITUDE | NOVARY | VARY,NOVARY |
| LONGITUDE | NOVARY | NOVARY,VARY |
| ELEVATION | NOVARY | VARY,VARY |
| TEMPERATURE | VARY | VARY,VARY |

Table 3.1: Example rVariables, CDFstats Monotonicity Checking

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

An additional option exists that allows the VALIDMIN, VALIDMAX, SCALEMIN, and SCALEMAX attribute entries to be updated based on the actual minimum and maximum values found for a variable. The MONOTON attribute entries may also be updated based on the monotonicity found for a variable.

### 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>] [/[NO]FORMAT]
           [/[NO]PAGE] [/[NO]UPDATE_VALIDS] [/[NO]UPDATE_SCALES]
           [/[NO]UPDATE_MONOTONIC] [/ZMODE=<mode>] [/[NO]NEG2POSFP0]
           [/REPORT=(<types>)] <cdf-path>
```

UNIX:

```
% cdfstats [-[no]range] [-[no]fill] [-output <file-name>] [-[no]format]
           [-[no]page] [-[no]update_valids] [-[no]update_scales]
           [-[no]update_monotonic] [-zmode <mode>] [-[no]neg2posfp0]
           [-report "<types>"] <cdf-path>
```

MS-DOS:

```
> cdfstats [-[no]range] [-[no]fill] [-output <file-name>] [-[no]format]
           [-[no]page] [-[no]update_valids] [-[no]update_scales]
           [-[no]update_monotonic] [-zmode <mode>] [-[no]neg2posfp0]
           [-report "<types>"] <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.

`/[NO]FILL`, `-[no]fill` Specifies whether or not fill values are ignored when collecting statistics (using the `FILLVAL` attribute if it exists).

`/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.

`/[NO]FORMAT`, `-[no]format`

Specifies whether or not the `FORMAT` attribute is used when displaying variable values (if the `FORMAT` attribute exists and an entry exists for the variable).

`/[NO]PAGE`, `-[no]page` Specifies whether or not the output is displayed a page (screen) at a time. If the output is directed to a file, paging is not allowed.

`/[NO]UPDATE_VALIDS`, `-[no]update_valids`

Specifies whether or not the `VALIDMIN` and `VALIDMAX` attribute entry values are updated for each variable based on the actual minimum and maximum values found (with fill values being ignored if requested). If the `VALIDMIN` and `VALIDMAX` attributes do not exist, they are created.

`/[NO]UPDATE_SCALES`, `-[no]update_scales`

Specifies whether or not the `SCALEMIN` and `SCALEMAX` attribute entry values are updated for each variable based on the actual minimum and maximum values found (with fill values being ignored if requested). If the `SCALEMIN` and `SCALEMAX` attributes do not exist, they are created.

`/[NO]UPDATE_MONOTONIC`, `-[no]update_monotonic`

Specifies whether or not the `MONOTONIC` attribute entry values are updated for each variable based on the monotonicity found (with fill values being ignored if requested). If the `MONOTONIC` attribute does not exist, it is created.

`/ZMODE=<mode>`, `-zmode <mode>`

Specifies the zMode that should be used with the CDF. `<mode>` may be one of the following:

`0` Indicates that zMode should be disabled.

1 Indicates that zMode/1 should be used. (The dimension variances of rVariables will be preserved.)

2 Indicates that zMode/2 should be used. (The dimensions of rVariables having a variance of NOVARY [false] are removed.)

/[NO]NEG2POSFP0, -[no]neg2posfp0

Specifies whether or not -0.0 is converted to 0.0 by the CDF library when read from a CDF. (The CDF is not modified.) -0.0 is an illegal floating point value on VAXes and DEC Alphas running OpenVMS.

/REPORT=(<types>), -report "<types>"

Specifies the types of return status codes from the CDF library that should be reported/displayed. <types> is a comma-separated list of zero or more of the following symbols: errors, warnings, or informationals.

Example(s):

VMS:

```
$ CDFSTATS TEST1
$ CDFSTATS/REPORT=(ERRORS) GISS_SOIL
$ CDFSTATS/NOFILL/OUTPUT=TEMPLATE3/NORANGE CDF$SMPL:TEMPLATE3
```

UNIX:

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

MS-DOS:

```
> cdfstats gisssoil
> cdfstats -range -nofill -report "errors" 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 (rVariables and zVariables),

```
<number>.  <name>  <n-dims>:[<dim-sizes>]  <rec-vary>/<dim-varys>
```

```
                                                    (<data-type>, <n-elems>)

          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 is disabled, the corresponding fields will not be displayed. The fields are defined as follows:

| | |
|---|---|
| `<number>` | The variable number. |
| `<name>` | The variable name. |
| `<rec-vary>` | The record variance of the variable — either a `T` or `F`. |
| `<dim-varys>` | The dimension variances of the variable — for each dimension either a `T` or `F`. This field is not present if there are zero (0) dimensions. |
| `<data-type>` | The data type of the variable (e.g., `CDF_REAL4`). |
| `<n-elems>` | The number of elements of the variable's data type. |
| `<n-dims>` | The number of dimensions of a zVariable. This field is not present for an rVariable. |
| `<dim-sizes>` | The dimension sizes of a zVariable. This field is not present for an rVariable or if the zVariable has zero (0) dimensions. |
| `<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 varies 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 gAttributes and vAttributes, number of variables (rVariables and zVariables), and number of dimensions and dimension sizes for the rVariables.
4. Global scope attribute definitions — the name, number, and number of entries for each gAttribute.
5. Global scope attribute entries — for each gAttribute entry, the data type, number of elements, and value(s).
6. Variable scope attribute definitions — the name, number, and number of entries for each vAttribute.
7. Variable definitions — the name, number, data type, number of elements, record variance, and dimension variances for each variable. For zVariables the number of dimensions and dimension sizes are also inquired.

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

### 3.7.2 Executing the CDFinquire Program

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

Usage:

VMS:

```
$ CDFINQUIRE [/DISPLAY=<mask>] [/OUTPUT=<file-path>] [/[NO]PAGE] [/ID]
             [/VARIABLES=<set>] [/ZMODE=<mode>] [/[NO]NEG2POSFP0]
             [/[NO]FORMAT] [/REPORT=(<types>)] <cdf-path>
```

UNIX:

```
% cdfinquire [-display <mask>] [-output <file-path>] [-[no]page] [-id]
             [-variables <set>] [-zmode <mode>] [-[no]neg2posfp0]
             [-[on]format] [-report "<types>"] <cdf-path>
```

MS-DOS:[3]

```
> cdfinquire [-display <mask>] [-output <file-path>] [-[no]page] [-id]
             [-variables <set>] [-zmode <mode>] [-[no]neg2posfp0]
             [-[on]format] [-report "<types>"] <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 display mask indicating the type of inquiry to be done. `<mask>` consists of one or more of the following:

- `i` CDF copyright and version/release/increment.
- `h` Header information.
- `g` gAttribute definitions.

[3] On MS-DOS systems the executable is named `CDFINQ.EXE`.

`g+` gAttribute definitions and entries.

`a` vAttribute definitions.

`r` rVariable definitions.

`r+` rVariable definitions and vAttribute rEntries.

`z` zVariable definitions.

`z+` zVariable definitions and vAttribute zEntries.

`v` rVariable/zVariable values. The `/VARIABLES`,`-variables` qualifier may be used to select the variables for which values are to be displayed.

Do not specify both `g` and `g+`, both `r` and `r+`, or both `z` and `z+`.

`/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.

`/[NO]PAGE`, `-[no]page` Specifies whether or not the output is displayed a page (screen) at a time. If the output is directed to a file, paging is not allowed.

`/VARIABLES=<set>`, `-variables <set>`

Specifies the set of variables (rVariables and/or zVariables) for which values should be displayed. `<set>` may be one of the following:

`all` All of the variables.

`nrv` Only those variables that are non-record variant (NRV, a record variance of `NOVARY`).

`novary+` Those variables for which one or more record/dimension variances are NOVARY.

`(<name1>,<name2>,...,<nameN>)`

A named set of variables. Each name must be delimited with a unique character. The names are separated by commas and enclosed in parentheses. **NOTE:** On VMS and OpenVMS systems the entire set must be enclosed in double quote marks, and double quote marks should not be used as delimiters [e.g., `"('LATITUDE','SST')"`]. **NOTE:** On UNIX systems the entire set must be enclosed in single quote marks and single quote marks should not be used as delimiters [e.g., `'("LATITUDE","SST")'`]. **NOTE:** On MS-DOS systems do not use double quote marks as delimiters.

Note that "`v`" must be present in the display mask for this qualifier to have an effect.

`/ZMODE=<mode>, -zmode <mode>`

Specifies the zMode that should be used with the CDF. `<mode>` may be one of the following:

- `0` Indicates that zMode should be disabled.
- `1` Indicates that zMode/1 should be used. (The dimension variances of rVariables will be preserved.)
- `2` Indicates that zMode/2 should be used. (The dimensions of rVariables having a variance of `NOVARY` [false] are removed.)

`/[NO]FORMAT, -[no]format`

Specifies whether or not the `FORMAT` attribute is used when displaying variable values (if the `FORMAT` attribute exists and an entry exists for the variable).

`/[NO]NEG2POSFP0, -[no]neg2posfp0`

Specifies whether or not `-0.0` is converted to `0.0` by the CDF library when read from a CDF. (The CDF is not modified.) `-0.0` is an illegal floating point value on VAXes and DEC Alphas running OpenVMS.

`/REPORT=(<types>), -report "<types>"`

Specifies the types of return status codes from the CDF library that should be reported/displayed. `<types>` is a comma-separated list of zero or more of the following symbols: `errors`, `warnings`, or `informationals`.

`/ID,-id` Causes the version of your CDF distribution and the default toolkit qualifiers to be displayed. All other parameters/qualifiers are ignored.

Example(s):

VMS:

```
$ CDFINQUIRE GISS_VEG
$ CDFINQUIRE/DISPLAY=ihzv/OUTPUT=GISS_VEG/REPORT=(ERRORS) GISS_VEG
$ CDFINQUIRE/ID
```

UNIX:

```
% cdfinquire giss_veg
% cdfinquire -display ihzv -output giss_veg -report "errors" giss_veg
% cdfinquire -id
```

MS-DOS:

```
> cdfinquire giss_veg
> cdfinquire c:\cdfs\giss\giss_veg
> cdfinquire -display ihzv -output ..\giss_veg -report "errors" 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+ar+z+v`):

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

<copyright>

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


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

  CDF name.............. <cdf-name>
  Encoding.............. <encoding>
  Majority.............. <majority>
  Format................ <format>
  rRecords.............. <num-r-records>
  Attributes............ <num-attributes> (<num-global-scope> global scope,
                                          <num-variable-scope> variable scope)
  rVariables............ <num-r-variables>
  rDimensions........... <num-dimensions>, sizes: [<dimension-sizes>]
  zVariables............ <num-z-variables>


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

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

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

                     .
                     .
                     .

      .
      .
      .
```

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

  <attr-num>.  Name.............. <attr-name>
               rEntries.......... <num-r-entries>
               zEntries.......... <num-z-entries>
       .
       .
       .


rVARIABLES
----------

  <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)
                 Entry value(s)... <entry-value(s)>

                    Values... (<num-records> records)

                              <record-num>:[<indices>] = <var-value>
                              <record-num>:[<indices>] = <var-value>
                                  .
                                  .
                                  .
                              <record-num>:[<indices>] = <var-value>

                       .
                       .
                       .

    .
    .
    .

zVARIABLES
----------

  <var-num>.  Name......... <var-name>
              Data type.... <var-data-type> (<var-num-elems> elements)
              Dimensions... <num-dimensions>, sizes: <dimension-sizes>
              Variances.... <rec-variance>/<dim-variances>

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

```
                    Values... (<num-records> records)

                              <record-num>:[<indices>] = <var-value>
                              <record-num>:[<indices>] = <var-value>
                                   .
                                   .
                                   .
                              <record-num>:[<indices>] = <var-value>

                    .
                    .
                    .

     .
     .
     .
```

Each field is defined as follows:

| | |
|---|---|
| `<attr-name>` | The attribute name. |
| `<attr-num>` | The attribute number. |
| `<cdf-name>` | The name of the CDF. |
| `<copyright>` | The CDF copyright notice. |
| `<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 2.3.10. If there are zero (0) dimensions, this field is omitted. |
| `<dimension-sizes>` | The dimension sizes for the rVariables or a zVariable. If there are zero (0) dimensions, this field is omitted. |
| `<encoding>` | The data encoding of the CDF. Data encodings are described in Section 2.2.7. |
| `<entry-data-type>` | The attribute entry data type. Data types are described in Section 2.2.9. |
| `<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 2.4.4. |
| `<entry-value(s)>` | The attribute entry value(s). |
| `<format>` | The format of the CDF. CDF formats are described in Section 2.2.6. |
| `<increment>` | The increment number of the CDF library that created/modified the CDF. |
| `<indices>` | The dimension indices separated by commas. An index for which the corresponding dimension variance is FALSE (`NOVARY`) will be |

| | |
|---|---|
| | represented as an asterisk (*) indicating that all of the values (or subarrays) along that dimension are the same. |
| <majority> | The variable majority of the CDF. Variable majorities are described in Section 2.3.11. |
| <num-attributes> | The number of attributes in the CDF. |
| <num-dimensions> | The number of dimensions for the rVariables or a zVariable. |
| <num-g-entries> | The number of gEntries for the gAttribute. |
| <num-global-scope> | The number of gAttributes in the CDF. |
| <num-r-entries> | The number of rEntries for the vAttribute. |
| <num-r-records> | Of all the rVariables the maximum number of records written. (zVariables are not considered.) |
| <num-r-variables> | The number of rVariables in the CDF. |
| <num-records> | The number of records actually written for a variable (physical, not virtual). |
| <num-variable-scope> | The number of vAttributes in the CDF. |
| <num-z-entries> | The number of zEntries for the vAttribute. |
| <num-z-variables> | The number of zVariables in the CDF. |
| <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 2.3.9. |
| <record-num> | The record number. If the variable's record variance is FALSE (NOVARY), this will be an asterisk (*) indicating that the values are the same for all records. |
| <release> | The release number of the CDF library that created/modified the CDF. |
| <var-data-type> | The variable data type. Data types are described in Section 2.2.9. |
| <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 2.3.8. |
| <var-value> | The variable value. |
| <version> | The version number of the CDF library that created/modified the CDF. |

If a display mask specifying less than "ihg+ar+z+v" is used, the information not requested will be omitted from 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 about a given CDF. It reads a CDF and writes into the skeleton table the following information.

1. Data encoding, majority, and format.
2. Number of dimensions and dimension sizes for the rVariables.
3. gAttribute definitions (and gEntry values).
4. rVariable and zVariable definitions and vAttribute definitions (and rEntry/zEntry values).
5. Non-record-variant (NRV) data for all of the CDF's NRV variables (if requested). NRV variables are described in Section 2.3.9

The above information is written in a format that 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] [/ZMODE <mode>]
                [/NONRV | /NRVTABLE] [-[NO]NEG2POSFP0] [/[NO]FORMAT]
                [/REPORT=(<types>)] <cdf-path>
```

UNIX:

```
% skeletontable [-skeleton <skeleton-path>] [-[no]log] [-zmode <mode>]
                [-nonrv | -nrvtable] [-[no]neg2posfp0] [-[no]format]
                [-report "<types>"] <cdf-path>
```

MS-DOS:[4]

```
> skeletontable [-skeleton <skeleton-path>] [-[no]log] [-zmode <mode>]
                [-nonrv | -nrvtable] [-[no]neg2posfp0] [-[no]format]
                [-report "<types>"] <cdf-path>
```

[4] On MS-DOS systems the executable is named `CDF2SKT.EXE`.

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`

Only 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.)

`/NRVTABLE`, `-nrvtable`

Put NRV data values in the skeleton table.

`/[NO]LOG`, `-[no]log` Specifies whether or not messages are displayed as the program executes.

`/ZMODE=<mode>`, `-zmode <mode>`

Specifies the zMode that should be used with the CDF. `<mode>` may be one of the following:

`0` Indicates that zMode should be disabled.

`1` Indicates that zMode/1 should be used. (The dimension variances of rVariables will be preserved.)

`2` Indicates that zMode/2 should be used. (The dimensions of rVariables having a variance of `NOVARY` [false] are removed.)

`/[NO]FORMAT`, `-[no]format`

Specifies whether or not the `FORMAT` attribute is used when writing NRV variable values (if the `FORMAT` attribute exists and an entry exists for the variable).

`/[NO]NEG2POSFP0`, `-[no]neg2posfp0`

Specifies whether or not `-0.0` is converted to `0.0` by the CDF library when read from a CDF. (The CDF is not modified.) `-0.0` is an illegal floating point value on VAXes and DEC Alphas running OpenVMS.

`/REPORT=(<types>)`, `-report "<types>"`

Specifies the types of return status codes from the CDF library that should be reported/displayed. `<types>` is a comma-separated list of zero or more of the following symbols: `errors`, `warnings`, or `informationals`.

Example(s):

VMS:

```
$ SKELETONTABLE/NOLOG/REPORT=(ERRORS) FGGE3B
$ SKELETONTABLE/SKELETON=FGGE3B/NONRV FGGE3B
```

UNIX:

```
% skeletontable -nolog -report "errors" fgge3b
% skeletontable -skeleton fgge3b -nonrv ../cdfs/fgge3b
```

MS-DOS:

```
> skeletontable -nolog -report "errors" 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 is 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. The skeleton table contains the information necessary to create a CDF that 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 and dimension sizes for the rVariables, format, data encoding, and variable majority.
2. The gAttribute definitions and any number of gEntries for each.
3. The rVariable and zVariable definitions.

4. The vAttribute 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.

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] [/[NO]FILLVAL]
              [/REPORT=(<types>)] <skeleton-path>
```

UNIX:

```
% cdfskeleton [-cdf <cdf-path>] [-[no]log] [-[no]delete] [-[no]fillval]
              [-report "<types>"] <skeleton-path>
```

MS-DOS:[5]

```
> cdfskeleton [-cdf <cdf-path>] [-[no]log] [-[no]delete] [-[no]fillval]
              [-report "<types>"] <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.

[5] On MS-DOS systems the executable is named `SKT2CDF.EXE`.

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

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

`/[NO]FILLVAL`, `-[no]fillval`

Specifies whether or not entries of the `FILLVAL` vAttribute are used to set the pad values for the corresponding variables. If this qualifier is specified, the `FILLVAL` vAttribute must exist and only those variables with an entry for the `FILLVAL` vAttribute will be affected.

`/REPORT=(<types>)`, `-report "<types>"`

Specifies the types of return status codes from the CDF library that should be reported/displayed. `<types>` is a comma-separated list of zero or more of the following symbols: `errors`, `warnings`, or `informationals`.

Example(s):

VMS:

```
$ CDFSKELETON FGGE3B
$ CDFSKELETON/NOLOG/CDF=[-.TEMP]FGGE3B_X/REPORT=(ERRORS) FGGE3B
```

UNIX:

```
% cdfskeleton fgge3b
% cdfskeleton -nolog -cdf ../fgge3b_x -report "errors" fgge3b
```

MS-DOS:

```
> cdfskeleton fgge3b
> cdfskeleton -nolog -cdf ..\fgge3b_x -report "errors" 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. The normal method of creating and using a skeleton table would be to use `SkeletonTable` on an existing CDF that is similar to the CDF you want to create. Then edit the created skeleton table to meet your needs, and use `CDFskeleton` to create the new CDF. The skeleton table could also be created from scratch with any text editor.

The format of the skeleton table is described in Appendix A.

## 3.10 CDFbrowse

`CDFbrowse` has been replaced by `CDFedit` (see Section 3.13).[6]

## 3.11 CDFlist

### 3.11.1 Introduction

The `CDFlist` program allows you to selectively list all (or a subset of) the rVariable 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 will most likely 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 [/INITIAL=(<defaults>)] [/SPECIFICATION=<cdf-spec>] [<cdf-spec>]
```

UNIX:

```
% cdflist [-initial "<defaults>"] [-specification <cdf-spec>] [<cdf-spec>]
```

MS-DOS:

```
> cdflist [-initial "<defaults>"] [-specification <cdf-spec>] [<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. |

Qualifier(s):

[6] The alias/symbol `CDFbrowse` still exists in the "definitions" file on UNIX/VMS systems but now executes `CDFedit` in a browse-only mode.

`/INITIAL=(<defaults>)`,`-initial "<defaults>"`

The default settings that initially appear on the `Input Options`, `Output Options`, and `Listing Options` windows. Whether or not to automatically select zMode may also be specified. `<defaults>` is a comma separated list of settings consisting of one or more of the following:

| | |
|---|---|
| `zmode` | When specified, zMode is automatically selected when a CDF is opened. |
| `[no]erase` | Specifies whether or not to erase the `Variable Selection` window when a new CDF is selected. |
| `[no]filter` | Specifies whether or not to use the variable filters when generating output. |
| `[no]fill` | Specifies whether or not to use the `FILLVAL` attribute when filtering. |
| `[no]format` | Specifies whether or not to use the `FORMAT` when encoding values to be displayed. |
| `[no]neg2posfp0` | Specifies whether or not to convert `-0.0` to `0.0` when detected. |
| `single \| multi` | Specifies the format of created CDFs. |
| `host \| network` | Specifies the encoding of created CDFs. |
| `vertical \| horizontal` | Specifies the type of text output desired while in zMode. |
| `epoch` or `epoch1` | Specifies the style in which EPOCH values should be displayed. |

These settings are only the settings initially in effect. The user may change any of them at any time. More detailed descriptions of each option may be found in the appropriate sections that follow.

`/SPECIFICATION=<cdf-spec>`,`-specification <cdf-spec>`

A default CDF specification (`<cdf-spec>`) for the CDF name field. The user may choose this specification or enter another.

Example(s):

VMS:

```
$ CDFLIST
$ CDFLIST GISS_WETL
$ CDFLIST NCDS$DATA:FGGE3B
$ CDFLIST/INITIAL=(ZMODE,HORIZONTAL)/SPECIFICATION=NCDS$DATA:
```

UNIX:

```
% cdflist
% cdflist ../cac_sst_blended
% cdflist ~/CDFs
% cdflist -initial "zmode,horizontal" -specification ~/CDFs
```

MS-DOS:

```
> cdflist
> cdflist ..\cac_sst
> cdflist a:\cdfs
> cdflist -initial "zmode,horizontal" -specification a:\cdfs
```

### 3.11.3 Interaction with CDFlist

The user interface to `CDFlist` is contained on a single screen 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 the `Action` key. Use the `Up` and `Down` keys to highlight a choice in the `Action Menu` and then enter the `Select` key to execute that function. The `Action Menu` appears in Figure 3.1.

```
Generate Output
Select Input CDF name
Select Output Device
Select Variables
Select Monotonic Variables
Save Variables to file
Restore Variables from file
Select r/z/zMODE Variables
Select CDF Input Options
Select CDF Output Options
Select CDF Listing Options
Help
Exit
```

Figure 3.1: `Action Menu`, `CDFlist`

Two very important windows are the `Keypad Definition` and `Message` windows. They contain information on keyboard functionality (i.e., which keys to use to perform the operations described below) 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 disk/directory/wildcard specification (without a CDF name), all of the CDFs meeting the specification will appear in a separate pop-up window. Use the `Up` and `Down` keys to choose a CDF and then enter the `Select` key to select that CDF.

The `Output` window will then be displayed with the valid choices. Use the `Up` and `Down` keys to move up and down within the window and then enter the `Select` key to select the desired output option. If output to an ASCII text 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 being 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 ASCII text file is `.lis`.

The second window (that remains constant on the screen) is the `Variable Selection` window. This window allows the user to select the variables 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 requesting the `Action Menu` and then selecting the `Select Variables` option.

There are four fields on each line in the `Variable Selection` window: variable name, minimum filter value, maximum filter value, and output toggle. You can move forward to the next field by entering the `Next Field` key or backwards to the previous field with the `Prev Field` key. You may also use the `Up` and `Down` keys to move vertically through the fields. When the cursor is positioned in a variable name field, entering the `Pop-up` key will display the variables in the CDF in a pop-up window. Use the arrow keys to highlight a variable and then enter the `Select` key to select it. You can quit the pop-up window without selecting a variable by entering the `Quit` key. You may also use the `Page Up` and `Page Down` keys to traverse through the pop-up window. The first line in this pop-up menu contains `**SELECT ALL**`. Selecting this item causes all of the variables to be selected. The second line in this pop-up window contains `**RECORD NUM**`. This simply gives the user the opportunity to display/filter on the record numbers in the CDF.

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 the CDF contains a `VALIDMIN` and `VALIDMAX` attribute entry for the selected variable, these values will be displayed as default minimum/maximum filter values. 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 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 the `Pop-up` key (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. (The `Page Up` and `Page Down` keys may also be used to traverse through the window.)

The minimum/maximum fields (filters) will initially be highlighted. This indicates that the corresponding filter will be used for a variable (if filter values are present). A filter for a particular variable may also be turned off (without erasing the filter values) by entering the `Filter On/Off` key when on the variable name field. This will toggle off the highlighting of the filter values and indicates that no filtering is to be performed on that variable. Entering the `Filter On/Off` key again toggles filtering back on for that variable. Note that erasing the filter values is another way to turn off filtering for a variable.

The output toggle field simply gives the user the option not to view a variable they have selected. A user may only wish to filter on a variable and not view it. When positioned on this field, the user simply enters the `Out on` key to display the variable and `Out Off` key to not display the variable. Note that the user may also enter `Out On/Off` keys from the variable name field.

When all of the desired variables and filters have been selected, enter the `Action` key to display the `Action Menu`. Selecting `Generate Output` will cause the listing or new CDF to be generated. Note that the amount of time needed to generate the output will in most cases be significantly reduced if the `MONOTON` attribute and corresponding entries are present in the CDF (and the variables being filtered are monotonic). See the

section explaining `Specifying Monotonic Variables` for more information on monotonic variables. 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 on-line help, select special options, or exit the program.

**NOTE:** On MS-DOS systems a listing may be terminated early by entering the `Quit` key.

When on the `Action Menu` options are available to save/restore the current selections in the `Variable Selection` window. A user may wish to exit the program and upon returning restore the current selections quickly to the window. When the `Save Variables To File` option is selected, the user is prompted for a file name. (Do not enter an extension.) All of the selections in the `Variable Selection` window are stored in that file. A default file name of `cdflist` is available by just entering the `Select` key at the prompt. When the `Restore Variables From File` option is selected, the user is again prompted for a file name. (`cdflist` is again the default.) The selections are restored to the `Variable Selection` window from that file. `CDFlist` will check the current CDF for the existence of each variable (displaying only those variables that do exist).

Another option on the `Action Menu` is `Select r/z/zMODE Variables`. `CDFlist` groups variables in three separate manners: rVariables, zVariables, and zMode. All of the rVariables are contained in one group; zVariables are grouped together by dimensionality; and zMode consists of all of the variables in the CDF with the rVariables being made to appear as zVariables. Variables from only one of these groups at a time may be selected for filtering/output. See Section 2.3 for more information on r/zVariables and Section 2.1.2 for more information on zMode.

On the `Action Menu` three windows of special options are available. Each option brings up a window containing a set of small option selection lists (an example of which is shown in Figure 3.2). The user makes selections by moving the `*` (the current choice) with the `Up` and `Down` keys and moving to the next list by entering the `Select` key. When entering an option selection list, if the current choice is the one desired, just enter the `Select` key to move to the next list. To return back to the `Action Menu`, enter the `Quit` key at any time.

```
(*) Single
( ) Multi
```

Figure 3.2: Option Selection List, `CDFlist`

Selecting `Select CDF Input Options` causes a window to be displayed with the following option selection lists:

1. Whether or not to erase the variables in the `Variable Selection` window each time a new input CDF is selected. If users select a CDF that contains duplicate variables, they may wish to reuse the current selections. If users choose not to erase selected variables, `CDFlist` will check to make sure that the current variables actually exist when a new CDF is selected.

Selecting `Select CDF Output Options` causes a window to be displayed with the following option selection lists:

1. When selecting output to a CDF, whether the file format of the new CDF is single-file or multi-file.
2. When selecting output to a CDF, whether the encoding of the new CDF is network or host.

Selecting `Select CDF Listing Options` causes a window to be displayed with the following option selection lists:

1. The format used when displaying EPOCH values may be selected. (The `FORMAT` attribute is always ignored when displaying EPOCH values.)
2. Whether or not to use the `FORMAT` attribute when displaying values may be selected. (The `FORMAT` attribute and corresponding entries must exist in the CDF.)
3. Whether or not to use the `FILLVAL` attribute when filtering (assuming the `FILLVAL` attribute exists). If the `FILLVAL` attribute is to be used, fill values in the data are ignored when determining whether or not to filter out a set (line) of values (i.e., fill values outside the filter range will appear in the output).
4. Whether or not to ignore all of the filters when generating the output. This is a more convenient way to turn off filtering than to do so manually for each variable on the `Variable Selection` window.
5. Whether or not to convert `-0.0` to `0.0` when detected. Some machines will not accept `-0.0` as a valid value. (`CDFlist` may abort with a system error.)
6. Whether to display values vertically or horizontally while in zMode.

On the `Action Menu` the option `Specify Monotonic Variables` gives the user the opportunity to specify that a variable is monotonic. This is useful when a CDF does not contain the attribute `MONOTON` or has the entries incorrectly specified. When this option is executed, a pop-up window is displayed containing all of the variables in the CDF and their monotonicities. If a `MONOTON` attribute entry exists for a variable, it is set accordingly. An `I` is displayed if the values increase, `D` if the values decrease, and `F` if the variable is not monotonic (or if an entry does not exist). If a variable varies with more than one "dimension", `N/A` is displayed — these variables can never be specified monotonic. Once inside the pop-up window, the user traverses through the window as in any other pop-up window but with one exception — the cursor is positioned to the monotonic value of the variable. The user then may change that value by entering the `INCREASE` key for increasing values, the `DECREASE` key for decreasing values, or the `FALSE` key for not monotonic.

## 3.12 CDFwalk

### 3.12.1 Introduction

The `CDFwalk` program allows direct access to all variable data in a CDF. 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 [/INITIAL=(<defaults>)] [/SPECIFICATION=<cdf-spec>] [<cdf-spec>]
```

UNIX:

```
% cdfwalk [-initial "<defaults>"] [-specification <cdf-spec>] [<cdf-spec>]
```

MS-DOS:

```
> cdfwalk [-initial "<defaults>"] [-specification <cdf-spec>] [<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.

Qualifier(s):

`/INITIAL=(<defaults>)`,`-initial "<defaults>"`

The default settings that initially appear on the `Special Options` window. `<defaults>` is a comma-separated list of settings consisting of one or more of the following:

| | |
|---|---|
| `[no]format` | Specifies whether or not to use the `FORMAT` when encoding values to be displayed. |
| `[no]neg2posfp0` | Specifies whether or not to convert `-0.0` to `0.0` when detected. |

These setting are only the settings initially in effect. The user may change any of them at any time. More detailed descriptions of each option may be found in the appropriate sections that follow.

`/SPECIFICATION=<cdf-spec>`,`-specification <cdf-spec>`

A default CDF specification (`<cdf-spec>`) for the CDF name field. The user may choose this specification or enter another.

Example(s):

VMS:

```
$ CDFWALK
$ CDFWALK [-.TEMP]NOAA7
$ CDFWALK/INITIAL=(NOFORMAT,NEG2POSFP0)/SPECIFICATION=NCDS$DATA:
```

UNIX:

```
% cdfwalk
% cdfwalk ../cac_sst
% cdfwalk /disk3/CDFs/ssclimate
% cdfwalk -initial "noformat,neg2posfp0" -specification ~/CDFs
```

MS-DOS:

```
> cdfwalk
> cdfwalk ..\cac_sst
> cdfwalk c:\cdfs
> cdfwalk -initial "noformat,neg2posfp0" -specification c:\cdfs
```

### 3.12.3 Interaction with CDFwalk

The user interface to `CDFwalk` is contained on a single screen 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", select r/zVariables, select `CDFwalk` special options, browse on-line help, or exit the program. This menu may be displayed at any time by entering the `Action` key. Use the `Up` and `Down` keys to highlight a choice in the `Action Menu`, and then enter the `Select` key to execute that function. The `Action Menu` appears in Figure 3.3.

```
Walk Thru CDF data
Select Input CDF name
Select Special Options
Select r/z Variables
Help
Exit
```

Figure 3.3: `Action Menu`, `CDFwalk`

Two very important windows are the `Keypad Definition` and `Message` windows. They contain information on keyboard functionality (i.e., which keys to use to perform the operations described below) and program status/error messages.

When `CDFwalk` is executed (and a CDF and/or directory/wildcard specification was not entered on the command line), you must first specify the CDF through which you wish to "walk." If you enter a disk/directory/wildcard specification, all of the CDFs meeting that specification will appear in a separate pop-up window. Use the `Up` and `Down` keys to choose a CDF and then enter the `Select` key to "walk" through that CDF.

After the input CDF is selected, the `Record/Indices` window appears. The size of this window depends on the dimensionality of the variables being displayed (initially the rVariables if any exist). Vertically,

the window will contain one row for the record number and one row for each dimension of the variables. 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, a pop-up window will be displayed containing the list of variables. 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. If a variable meeting the requirements mentioned above does not exist, only the record number field will be used.

The variable selected for each dimension line will be one that only varies 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. If for a particular dimension a variable meeting the requirements mentioned above does not exist, only the dimension index field will be used.

After the `Record/Indices` window has been specified (which will usually 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 (assuming that a variable was found meeting the above requirements; if not, the cursor will start at the record number field). You may now begin to "walk" through the CDF by using the right arrow to increment to and the left arrow to decrement to a new record. For each new record 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 advance quickly to a desired record. You may do this by entering the `Input` key. `CDFwalk` enters an input mode and prompts for 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 the `Pop-up` key. 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 if the CDF is multidimensional) 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 indices rather than data values. (This would also be necessary if variables meeting the above requirements were not found.) This is done by entering the `Next Field` key, which causes the cursor to advance to the next field (i.e., from the data value field to the record number/indices field). Entering the `Back Field` key 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 `Pop-up` option is available with these fields. Also, in input mode (after entering the `Input` key) you may input either an absolute record number/index to which to move or a relative offset from the current record number/index. Prefixing the value with a `+/-` will increment/decrement the record number/index by that amount.

If more than 27 variables exist and a particular variable is not currently displayed in the `Variable Values` window, an option is provided to scroll new variables into that window. This is done by entering the `Scroll` key. A small pop-up window is displayed in which you may use the `Next/Prev Variable` keys to scroll

in or scroll out variables. When the desired set of variables is displayed, enter the `Select` key to continue "walking" through the CDF.

When you are finished "walking" through the CDF, enter the `Action` key to display the `Action Menu`. You may input a new CDF, resume "walking" through the current CDF, select special options, read on-line help, or exit the program.

On the `Action Menu` selecting `Special Options` causes a window to be displayed with the following option selection lists:

1. Whether or not to use the `FORMAT` attribute when displaying values may be selected. (The `FORMAT` attribute and corresponding entries must exist in the CDF.)
2. Whether or not to convert `-0.0` to `0.0` when detected. Some machines will not accept `-0.0` as a valid value. (`CDFwalk` may abort with a system error.)

The user makes a selection by moving the `*` (the current choice) with the `Up` and `Down` keys and moving to the next options selection list (if another exists) by entering the `Select` key. An example option selection list is shown in Figure 3.4. To return to the `Action Menu`, enter the `Quit` key at any time.

```
(*) CDF FORMAT attribute
( ) CDFWALK defaults
```

Figure 3.4: Option Selection List, `CDFwalk`

Another option on the `Action Menu` is `Select r/z Variables`. `CDFwalk` groups variables in two separate manners: rVariables and zVariables. All of the rVariables are in one group and the zVariables are grouped together by dimensionality. Only one of these groups may be "walked" through at a time. For more information on the r/zVariables see Section 2.3.

## 3.13 CDFedit

### 3.13.1 Introduction

The `CDFedit` program allows the display and/or modification of practically all of the contents of a CDF by way of a full-screen interface. It is also possible to run `CDFedit` in a browse-only mode in order to prevent accidental modifications.[7]

### 3.13.2 Executing the CDFedit Program

The command line syntax for `CDFedit` is as follows:

Usage:

VMS:

[7] Running `CDFedit` in a browse-only mode provides the same functionality as `CDFbrowse` once did.

```
$ CDFEDIT [/[NO]BROWSE] [/ZMODE=<mode>] [/[NO]FORMAT] [/[NO]PROMPT]
          [/[NO]NEG2POSFP0] [/REPORT=(<types>)] <cdf-spec>
```

UNIX:

```
% cdfedit [-[no]browse] [-zmode <mode>] [-[no]format] [-[no]prompt]
          [-[no]neg2posfp0] [-report "<types>"] <cdf-spec>
```

MS-DOS:

```
> cdfedit [-[no]browse] [-zmode <mode>] [-[no]format] [-[no]prompt]
          [-[no]neg2posfp0] [-report "<types>"] <cdf-spec>
```

Parameter(s):

`<cdf-spec>` The specification of the CDF(s) to edit. (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. If the `/PROMPT`,`-prompt` qualifier is used, ¡cdf-spec¿ will appear as the initial specification at the prompt. If this parameter is omitted, the `/PROMPT`,`-prompt` qualifier must be specified (and the initial specification at the prompt will be the current directory).

Qualifier(s):

`/[NO]BROWSE`, `-[no]browse`

Specifies whether or not a browsing mode is desired. In browsing mode the creation, modification, or deletion of a CDF is not allowed.

`/ZMODE=<mode>`, `-zmode <mode>`

Specifies which zMode should be used. `<mode>` may be one of the following:

- `0` Indicates that zMode should be disabled.
- `1` Indicates that zMode/1 should be used. (The dimension variances of rVariables will be preserved.)
- `2` Indicates that zMode/2 should be used. (The dimensions of rVariables having a variance of `NOVARY` [false] are removed.)

`/[NO]FORMAT`, `-[no]format`

Specifies whether or not the `FORMAT` attribute is used when displaying variable values (if the `FORMAT` attribute exists and an entry exists for the variable).

`/[NO]PROMPT`, `-[no]prompt`

Specifies whether or not a prompt is issued for the CDF(s) specification. When enabled the prompt will be issued both at program startup and after editing the current CDF(s) specification (at which point a new CDF[s] specification may be specified). If a CDF(s) specification is entered at the command line along with this qualifier, the CDF(s) specification will appear at the prompt. (Otherwise, the current directory will appear at the prompt.)

`/[NO]NEG2POSFP0`, `-[no]neg2posfp0`

Specifies whether or not `-0.0` is converted to `0.0` by the CDF library when read from a CDF. (The CDF is not modified.) `-0.0` is an illegal floating point value on VAXes and DEC Alphas running OpenVMS.

`/REPORT=(<types>)`, `-report "<types>"`

Specifies the types of return status codes from the CDF library that should be reported/displayed. `<types>` is a comma-separated list of zero or more of the following symbols: `errors`, `warnings`, or `informationals`.

Example(s):

VMS:

```
$ CDFEDIT [.SAMPLES]
$ CDFEDIT/ZMODE=2/NOFORMAT GISS_WETLX
$ CDFEDIT/BROWSE/PROMPT/REPORT=(ERRORS)
```

UNIX:

```
% cdfedit samples
% cdfedit -zmode 2 -noformat giss_wetl
% cdfedit -browse -prompt -report "errors"
```

MS-DOS:

```
> cdfedit samples
> cdfedit -zmode 2 -noformat giss_wetl
> cdfedit -browse -prompt -report "errors"
```

### 3.13.3 Interaction with CDFedit

Interaction with `CDFedit` is through a series of menus and windows. Extensive on-line help is provided and will not be repeated here.[8] The on-line help does refer to the sections of a window by name. Figure 3.5 illustrates the various sections of the possible types of windows.

[8] It is our intention that the use of `CDFedit` be as intuitive as possible. You may not even need the on-line help. We're sure you'll let us know.

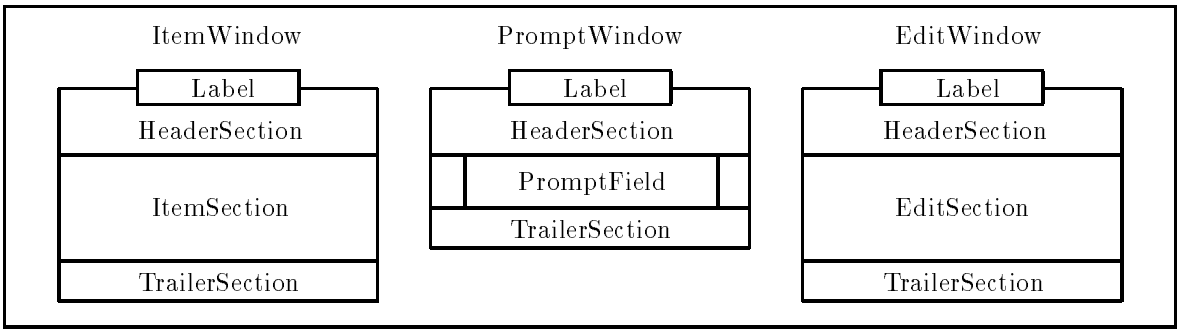


Figure 3.5: Window Sections, `CDFedit`

ItemWindows are used when a choice is to be made from a list of one or more items (e.g., functions to perform, CDFs to edit, variable names, etc.). In some cases the entire list of items may not fit on the screen at once. When this occurs, the ItemSection may be scrolled to display hidden items. Some ItemWindows have a percentage indicator at the bottom right portion of the ItemSection. The percentage indicator shows which part of the ItemSection is being displayed.

PromptWindows are used when a textual response is required (e.g., a CDF specification, a new attribute name, a variable value, etc.). If the text is too long to fit into the PromptField, the "more" indicators ("<" and ">") at the left and right ends of the PromptField will display where hidden characters exist.

EditWindows are used to display/edit a text file or group of lines. EditWindows are currently only used to display on-line help.

# 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 that calls the CDF library or references CDF parameters must include `cdf.h`. On VMS systems a logical name, `CDF$INC`, that specifies the location of `cdf.h` is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_INC`, that serves the same purpose is defined in the definitions file `definitions.<shell>` (where `<shell>` is the shell being used: `csh`, `ksh`, `sh`, or `tcsh`). This section assumes that you are using the appropriate definitions file.

One of two methods may be used to include `cdf.h`. They are described in the following sections.

### 4.2.1 Specifying `cdf.h` Location in the Compile Command

The first method involves including the following line at/near 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

An example of the command to compile a source file on VMS systems would be as follows:

```
$ CC/INCLUDE_DIRECTORY=CDF$INC <source-name>
```

where `<source-name>` is the name of the source file being compiled. (The `.C` extension does not have to be specified.) The object module created will be named `<source-name>.OBJ`.

**NOTE:** If you are running OpenVMS on a DEC Alpha and are using a CDF distribution built for a default double-precision floating-point representation of `D_FLOAT`, you will also have to specify `/FLOAT=D_FLOAT` on the `CC` command line in order to correctly process double-precision floating-point values.

### UNIX Systems

An example of the command to compile a source file on UNIX systems would be as follows:

```
% cc -c -I${CDF_INC} <source-name>.c
```

where `<source-name>.c` is the name of the source file being compiled (the `.c` extension is required). The `-c` option specifies that only an object module is to be produced. (The link step is described in Section 4.3.) The object module created will be named `<source-name>.o`.

### MS-DOS Systems

An example of the command to compile a source file on MS-DOS systems would be as follows (for Microsoft C and Borland C, respectively):

```
> CL /c /AL /I<inc-path> <source-name>.c
> BCC -c -ml -I<inc-path> <source-name>.c
```

where `<source-name>.c` is the name of the source file being compiled (the `.c` extension is required) 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.

You may also need to specify the location of system include files. For Microsoft C this is usually accomplished by setting MS-DOS environment variables. For Borland C it may be necessary to also specify `-I<inc-path>` where `<inc-path>` is the location of the Borland C system include files. Consult the documentation for the compiler being used for more information.

The `/c` (for Microsoft C) and `-c` (for Borland C) options specify that only an object module is to be produced. (The link step is described in Section 4.3.) The object module will be named `<source-name>.obj`.

The `/AL` (for Microsoft C) and `-ml` (for Borland C) options specify that the object module 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. The CDF libraries supplied with the CDF distribution for MS-DOS systems are compiled using the large 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, 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/near 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.

On VMS systems `CDF$INC:` may be used for `<inc-path>`. On UNIX and MS-DOS systems `<inc-path>` must be a relative or absolute pathname. (An environment variable may not be used for `<inc-path>` on UNIX systems.) You will need to know 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).[1] Both the Standard and Internal interfaces for C applications are built into the CDF library. On VMS systems a logical name, `CDF$LIB`, which specifies the location of the CDF library, is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_LIB`, which serves the same purpose, is defined in the definitions file `definitions.<shell>` (where `<shell>` is the shell being used: `csh`, `ksh`, `sh`, or `tcsh`). This section assumes that you are using the appropriate definitions file.

### 4.3.1 VMS Systems

An example of the command to link your application with the CDF library on VMS systems would be as follows:

```
$ LINK <object-file(s)>, CDF$LIB:LIBCDF/LIBRARY
```

[1] A shareable version of the CDF library is also available on VMS and some flavors of UNIX. Its use is described in Section 4.4.

where <object-file(s)> is your application's object module(s). (The .OBJ extension is not necessary.) The name of the executable created will be the name part of the first object file listed with .EXE appended. A different executable name may be specified by using the /EXECUTABLE qualifier.

### 4.3.2 OpenVMS Systems

An example of the command to link your application with the CDF library on OpenVMS systems would be as follows:

```
$ LINK <object-file(s)>, CDF$LIB:LIBCDF/LIBRARY, SYS$LIBRARY:<crtl>/LIBRARY
```

where <object-file(s)> is your application's object module(s) (the .OBJ extension is not necessary) and <crtl> is VAXCRTL if your CDF distribution is built for a default double-precision floating-point representation of G_FLOAT or VAXCRTLD for a default of D_FLOAT. The name of the executable created will be the name part of the first object file listed with .EXE appended. A different executable name may be specified by using the /EXECUTABLE qualifier.

### 4.3.3 UNIX Systems

An example of the command to link your application with the CDF library on UNIX systems would be as follows:

```
% cc <object-file(s)>.o ${CDF_LIB}/libcdf.a
```

where <object-file(s)>.o is your application's object module(s). (The .o extension is required.) The name of the executable created will be a.out by default. It may also be explicitly specified using the -o option. Some UNIX systems may also require that -lc (the C runtime library), -lm (the math library), and/or -ldl (the dynamic linker library) be specified at the end of the command line. This may depend on the particular release of the operating system being used.

#### Combining the Compile and Link

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

```
% cc -I${CDF_INC} <source-name(s)>.c ${CDF_LIB}/libcdf.a
```

where <source-name(s)>.c is the name of the source file(s) being compiled/linked. (The .c extension is required.) Some UNIX systems may also require that -lc, -lm, and/or -ldl be specified at the end of the command line.

### 4.3.4 MS-DOS Systems

An example of the command to link your application with the CDF library on MS-DOS systems would be as follows (for Microsoft C[2] and Borland C, respectively):

```
> LINK /ST:<stack-size> <object-file(s)>,<executable>,nul.map,
       <lib-path>libcdf.lib;
> TLINK /x /L<bc>\lib C0L <object-file(s)>,<executable>,,
        <lib-path>libcdf.lib CL EMU MATHL
```

where `<object-file(s)>` is your application's object module(s) (the `.obj` extension is not necessary); `<executable>` is the name of the executable file to be created (with an extension of `.exe`); 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.

For Microsoft C `<stack-size>` is the size (in decimal bytes) of the executable's stack. A value large enough to prevent runtime stack overflow errors should be specified. (The default value is 2048 [decimal].) A map file is created by default unless the special name `nul.map` is used (as shown). If a map file is desired, the map file parameter should be omitted (in which case the name of the map file will be the name part of the executable file with `.map` appended) or a map file (other than `nul.map`) should be explicitly specified. Note that if `<executable>` is omitted, the name of the executable created will be the name part of the first object module listed with an extension of `.exe` appended.

For Borland C `<bc>` is the directory path of your Borland C software installation top-level directory. This helps Borland C find the necessary startup module (`C0L`) and system libraries (`CL`, `EMU`, and `MATHL`). This example assumes you are using the large memory model. (If the huge memory model were being used, `C0H` and `MATHH` would have been specified instead of `C0L` and `MATHL`, respectively.) The `EMU` library specified indicates that floating-point emulation should be used if a math coprocessor is not present at runtime. If you have a math coprocessor chip, you may want to specify `FP87` instead for increased performance (but the executable will not run on machines that do not have a math coprocessor chip). The omitted parameter (indicated by `,,`) is the name of the map file to be created. A map file is created by default unless the `/x` option is used (as shown). 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).

**NOTE:** The same memory model must have been used to compile your application's source files and the CDF library. The CDF library can be compiled using either the large or huge memory model. The CDF libraries supplied with the CDF distribution for MS-DOS systems are compiled using the large 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, 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.

[2] This example assumes you have properly set the MS-DOS environment variables used by the Microsoft C compiler and linker. There were some changes to the Microsoft C 6.00 and Microsoft C 7.00 runtime libraries (regarding system function names). Because the CDF library is compiled (and distributed) using Microsoft C 6.00, you should be able to successfully link your application with either Microsoft C compiler (since Microsoft C 7.00 is backwards compatible with object modules/libraries created with Microsoft C 6.00).

**Combining the Compile and Link**

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

```
> CL /AL /F <stack-size> /I<inc-path> <source-name(s)>.c <lib-path>libcdf.lib
> BCC -ml -I<inc-path> <source-name(s)>.c <lib-path>libcdf.lib
```

where `<source-name(s)>.c` is the name of the source file(s) being compiled (the `.c` extension is required); `<inc-path>` is the pathname of the directory containing `cdf.h`; and `<lib-path>` is the pathname of the directory containing the CDF library. You will need to know where on your system `cdf.h` and the CDF library have been installed.

For Microsoft C `<stack-size>` is the size (in hexadecimal bytes) of the executable's stack. A value large enough to prevent runtime stack overflow errors should be specified. (The default value is 800 [hexadecimal].)

## 4.4 Linking, Shared CDF Library

A shareable version of the CDF library is also available on VMS systems and some flavors of UNIX. The shared version is put in the same directory as the non-shared version and is named as follows:

| Machine/Operating System | Shared CDF Library |
|---|---|
| VAX/VMS & DEC Alpha/OpenVMS | `LIBCDF.EXE` |
| Sun/SunOS & Sun/SOLARIS | `libcdf.so` |
| HP 9000/HP-UX | `libcdf.sl` |
| IBM RS6000/AIX | `libcdf.o` |

The commands necessary to link to a shareable library vary among operating systems. Examples are shown in the following sections.

### 4.4.1 VAX/VMS

```
$ ASSIGN CDF$LIB:LIBCDF.EXE CDF$LIBCDFEXE
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
  CDF$LIBCDFEXE/SHAREABLE
  SYS$SHARE:VAXCRTL/SHAREABLE
  <Control-Z>
$ DEASSIGN CDF$LIBCDFEXE
```

where `<object-file(s)>` is your application's object module(s). (The `.OBJ` extension is not necessary.) The name of the executable created will be the name part of the first object file listed with `.EXE` appended. A different executable name may be specified by using the `/EXECUTABLE` qualifier.

**NOTE:** On VMS systems the shareable CDF library may also be installed in `SYS$SHARE`.

### 4.4.2 DEC Alpha/OpenVMS

```
$ ASSIGN CDF$LIB:LIBCDF.EXE CDF$LIBCDFEXE
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
  CDF$LIBCDFEXE/SHAREABLE
  SYS$LIBRARY:<crtl>/LIBRARY
  <Control-Z>
$ DEASSIGN CDF$LIBCDFEXE
```

where <object-file(s)> is your application's object module(s) (the .OBJ extension is not necessary) and <crtl> is VAXCRTL if your CDF distribution is built for a default double-precision floating-point representation of G_FLOAT or VAXCRTLD for a default of D_FLOAT. The name of the executable created will be the name part of the first object file listed with .EXE appended. A different executable name may be specified by using the /EXECUTABLE qualifier.

**NOTE:** On OpenVMS systems the shareable CDF library may also be installed in SYS$SHARE.

### 4.4.3 Sun/SunOS

```
% cc -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lm -ldl
```

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created.

### 4.4.4 Sun/SOLARIS

```
% cc -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lc -lm
```

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created.

### 4.4.5 HP 9000/HP-UX

```
% cc -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.sl
```

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created.

### 4.4.6 IBM RS6000/AIX

```
% cc -o <exe-file> <object-file(s)>.o -L${CDF_LIB} ${CDF_LIB}/libcdf.o -lc -lm
```

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created.

## 4.5 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. Note that both rVariables and zVariables are numbered starting at zero (0).

## 4.6 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. The file cdf.h should be #include'd in all application source files referencing CDF functions/parameters.

### 4.6.1 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 C lists the possible status codes along with their explanations. Section 4.8 describes how to interpret status codes. |
| CDFid | An identifier (or handle) for a CDF that 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. |

### 4.6.2 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 4.8 describes how to use these constants to interpret status codes.

### 4.6.3 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. |

### 4.6.4 CDF Data Types

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).

**NOTE:** When using a DEC Alpha running OSF/1 keep in mind that a long is 8 bytes and that an int is 4 bytes. Use int C variables with the CDF data types CDF_INT4 and CDF_UINT4 rather than long C variables.

**NOTE:** When using an IBM PC (MS-DOS) keep in mind that an int is 2 bytes and that a long is 4 bytes. Use long C variables with the CDF data types CDF_INT4 and CDF_UINT4 rather than int C variables.

### 4.6.5 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 passed into the CDF library (to be written to a CDF) should always be in the host machine's native encoding. Attribute entry and variable values read from a CDF by the CDF library and passed out to an application will be in the currently selected decoding for that CDF (see Section 2.2.8).

| | |
|---|---|
| `HOST_ENCODING` | Indicates host machine data representation (native). This encoding will provide the greatest performance when reading/writing on a machine of the same type. |
| `NETWORK_ENCODING` | Indicates network transportable data representation (XDR). |
| `VAX_ENCODING` | Indicates VAX data representation. Double-precision floating-point values are encoded in Digital's `D_FLOAT` representation. |
| `ALPHAVMSd_ENCODING` | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's `D_FLOAT` representation. |
| `ALPHAVMSg_ENCODING` | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's `G_FLOAT` representation. |
| `SUN_ENCODING` | Indicates SUN data representation. |
| `SGi_ENCODING` | Indicates Silicon Graphics Iris and Power Series data representation. |
| `DECSTATION_ENCODING` | Indicates DECstation data representation. |
| `ALPHAOSF1_ENCODING` | Indicates DEC Alpha running OSF/1 data representation. |
| `IBMRS_ENCODING` | Indicates IBMRS data representation (IBM RS6000 series). |
| `HP_ENCODING` | Indicates HP data representation (HP 9000 series). |
| `IBMPC_ENCODING` | Indicates IBM PC data representation. |
| `NeXT_ENCODING` | Indicates NeXT data representation. |

When creating a CDF (via the Standard Interface) or respecifying a CDF's encoding (via the Internal Interface), you must specify `NETWORK_ENCODING`, `HOST_ENCODING`, or the specific encoding of the host machine (e.g., `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.)

### 4.6.6 Data Decodings

A CDF's decoding affects how its attribute entry and variable data values are passed out to a calling application. The decoding for a CDF may be selected and reselected any number of times while the CDF is open. Selecting a decoding does not affect how the values are stored in the CDF file(s) — only how the

values are decoded by the CDF library. Any decoding may be used with any of the supported encodings. Section 2.2.8 describes a CDF's decoding in more detail.

| | |
|---|---|
| HOST_DECODING | Indicates host machine data representation (native). This is the default decoding. |
| NETWORK_DECODING | Indicates network transportable data representation (XDR). |
| VAX_DECODING | Indicates VAX data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation. |
| ALPHAVMSd_DECODING | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation. |
| ALPHAVMSg_DECODING | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's G_FLOAT representation. |
| SUN_DECODING | Indicates SUN data representation. |
| SGi_DECODING | Indicates Silicon Graphics Iris and Power Series data representation. |
| DECSTATION_DECODING | Indicates DECstation data representation. |
| ALPHAOSF1_DECODING | Indicates DEC Alpha running OSF/1 data representation. |
| IBMRS_DECODING | Indicates IBMRS data representation (IBM RS6000 series). |
| HP_DECODING | Indicates HP data representation (HP 9000 series). |
| IBMPC_DECODING | Indicates IBM PC data representation. |
| NeXT_DECODING | Indicates NeXT data representation. |

The default decoding is HOST_DECODING. The other decodings may be selected via the Internal Interface with the <SELECT_,CDF_DECODING_> operation. Section 2.2.8 describes those situations in which a decoding other than HOST_DECODING may be desired.

### 4.6.7 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). The majority is the same for rVariable and zVariables.

| | |
|---|---|
| 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 majority of a CDF's variables is necessary when performing hyper reads and writes. During a hyper read the CDF library will place the variable data values into the memory buffer in the same majority

as that of the variables. The buffer must then be processed according to that majority. Likewise, during a hyper write, the CDF library will expect to find the variable data values in the memory buffer in the same majority as that of the variables.

The majority must also be considered when performing sequential reads and writes. When sequentially reading a variable, the values passed out by the CDF library will be ordered according to the majority. When sequentially writing a variable, the values passed into the CDF library are assumed (by the CDF library) to be ordered according to the majority.

### 4.6.8 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 of the other records are virtual and contain the same values.)

If a variable has a dimension variance of `VARY`, then each value/subarray along that dimension is physically stored. If the dimension variance is `NOVARY`, then only one value/subarray along that dimension is physically stored. (All other values/subarrays along that dimension are virtual and contain the same values.)

### 4.6.9 Attribute Scopes

Attribute scopes are simply a way to explicitly declare the intended use of an attribute by user applications (and the CDF toolkit).

| | |
|---|---|
| `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 rEntry or zEntry corresponds to an rVariable or zVariable, respectively.) |
| `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 2.4.3 explains attribute scopes and how assumed scopes can arise in a CDF.

### 4.6.10 Read-Only Modes

Once a CDF has been opened, it may be placed into a read-only mode to prevent accidental modification (such as when the CDF is simply being browsed). Read-only mode is selected via the Internal Interface using the <SELECT_,CDF_READONLY_MODE_> operation.

| | |
|---|---|
| READONLYon | Turns on read-only mode. |
| READONLYoff | Turns off read-only mode. |

### 4.6.11 zModes

Once a CDF has been opened, it may be placed into one of two variations of zMode. zMode is fully explained in Section 2.1.2. A zMode is selected for a CDF via the Internal Interface using the <SELECT_,CDF_zMODE_> operation.

| | |
|---|---|
| zMODEoff | Turns off zMode. |
| zMODEon1 | Turns on zMode/1. |
| zMODEon2 | Turns on zMode/2. |

### 4.6.12 -0.0 to 0.0 Modes

Once a CDF has been opened, the CDF library may be told to convert -0.0 to 0.0 when read from that CDF. This mode is selected via the Internal Interface using the <SELECT_,CDF_NEGtoPOSfp0_MODE_> operation.

| | |
|---|---|
| NEGtoPOSfp0on | Convert -0.0 to 0.0 when read from the CDF. |
| NEGtoPOSfp0off | Do not convert -0.0 to 0.0 when read from the CDF. |

### 4.6.13 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 for the rVariables or a zVariable. |

The CDF library imposes no limit on the number of variables, attributes, or attribute entries that a CDF may have. On the IBM PC, however, the number of rVariables and zVariables will be limited to 100 of each in a multi-file CDF because of the 8.3 naming convention imposed by MS-DOS.

### 4.6.14 Limits of Names and Other Character Strings

| | |
|---|---|
| CDF_PATHNAME_LEN | Maximum length of a CDF pathname (excluding the NUL[3] 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 CDF copyright text (excluding the NUL terminator). |
| CDF_STATUSTEXT_LEN | Maximum length of the explanation text for a status code (excluding the NUL terminator). |

## 4.7 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 4.8). 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. Note that zVariables and vAttribute zEntries are only accessible via the Internal Interface.

Each section begins with an ANSI-style function prototype. The include file cdf.h contains function prototypes for each of the functions in the Standard and Internal interfaces (as well as the EPOCH utility routines). Note that many of the Standard Interface functions are implemented as macros (which call the Internal Interface).

### 4.7.1 CDFcreate (Standard Interface)

```
CDFstatus CDFcreate(
char  *CDFpath,        /* in  -- CDF pathname. */
long  numDims,         /* in  -- Number of dimensions, rVariables. */
long  dimSizes[],      /* in  -- Dimension sizes, rVariables. */
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 overwrite 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 and .z0,.z1,... files.

The arguments to CDFcreate are defined as follows:

[3] The ASCII null character, 0 octal.

| | |
|---|---|
| CDFpath | The pathname of the CDF to create. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters (excluding the NUL terminator). 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).<br><br>**NOTE:** On UNIX systems pathnames are case-sensitive. |
| numDims | Number of dimensions the rVariables in the CDF are 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). If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| encoding | The encoding for variable data and attribute entry data. Specify one of the encodings described in Section 4.6.5. |
| majority | The majority for variable data. Specify one of the majorities described in Section 4.6.7. |
| 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 specified in the configuration file of your CDF distribution. Consult your system manager for this default. The CDFlib function (Internal Interface) may be used to change a CDF's format to single-file.

**NOTE:** 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 4.7.5).

### Example(s)

The following example will create a 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,
                                                rVariables. */
static long dimSizes[3] = {180,360,10};      /* Dimension sizes,
                                                rVariables. */
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.

### 4.7.2 CDFopen (Standard Interface)

```
CDFstatus CDFopen(
char  *CDFpath,        /* 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 cannot get write access to the CDF.)

The arguments to CDFopen are defined as follows:

| | |
|---|---|
| CDFpath | The pathname of the CDF to open. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters (excluding the NUL terminator). 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).<br><br>**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. |

**NOTE:** 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 4.7.5).

#### 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 CDFpath[] = { "NOAA1" };   /* Pathname of CDF. */
.
.
status = CDFopen (CDFpath, &id);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

### 4.7.3 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 that created the CDF is provided (e.g., CDF V2.4 is version 2, release 4) 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 that created the CDF. |
| `release` | The release number of the CDF library that created the CDF. |
| `copyRight` | The copyright notice of the CDF library that 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 (e.g., CDF V2.4 is version 2, release 4).

#### 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);
}
.
.
```

### 4.7.4 CDFinquire (Standard Interface)

```
CDFstatus CDFinquire(
CDFid id,                       /* in  -- CDF identifier */
long  *numDims,                 /* out -- Number of dimensions, rVariables. */
long  dimSizes[CDF_MAX_DIMS],   /* out -- Dimension sizes, rVariables. */
long  *encoding,                /* out -- Data encoding. */
long  *majority,                /* out -- Variable majority. */
long  *maxRec,                  /* out -- Maximum record number in the
                                           CDF, rVariables. */
long  *numVars,                 /* out -- Number of rVariables 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 rVariable dimensions and their sizes before it can access rVariable data. Knowing the variable majority can be used to optimize performance and is necessary to properly use the variable hyper functions (for both rVariables and zVariables).

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 for the rVariables in the CDF. |
| `dimSizes` | The dimension sizes of the rVariables in the CDF. `dimSizes` is a 1-dimensional array containing one element per dimension. Each element of `dimSizes` receives the corresponding dimension size. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| `encoding` | The encoding of the variable data and attribute entry data. The encodings are defined in Section 4.6.5. |

| | |
|---|---|
| majority | The majority of the variable data. The majorities are defined in Section 4.6.7. |
| maxRec | The maximum record number written to an rVariable in the CDF. Note that the maximum record number written is also kept separately for each rVariable in the CDF. The value of maxRec is the largest of these. Some rVariables may have fewer records actually written. CDFlib (Internal Interface) may be used to inquire the maximum record written for an individual rVariable (see Section 4.7.25). |
| numVars | The number of rVariables 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, rVariables. */
long dimSizes[CDF_MAX_DIMS];    /* Dimension sizes, rVariables
                                   (allocate to allow the maximum number
                                   of dimensions). */
long encoding;                  /* Data encoding. */
long majority;                  /* Variable majority. */
long maxRec;                    /* Maximum record number, rVariables. */
long numVars;                   /* Number of rVariables 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);
.
.
```

### 4.7.5 CDFclose (Standard Interface)

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

CDFclose closes the specified CDF. The CDF's cache buffers are flushed; the CDF's open file is closed (or files in the case of a multi-file CDF); and the CDF identifier is made available for reuse.

**NOTE:** 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 cache buffers are left unflushed.

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);
.
.
```

### 4.7.6 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 and `.z0`,`.z1`,... files.

You must open a CDF before you are allowed to delete 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);
}
.
.
```

### 4.7.7 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 4.8 explains how to interpret status codes and Appendix C 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);
}
.
.
```

### 4.7.8 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 4.6.9. |
| `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);
.
.
```

### 4.7.9 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 (e.g., 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 4.7.10.

```
.
.
#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);
```

### 4.7.10 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 4.7.9). |
| `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);
.
.
```

### 4.7.11 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. */
```

`CDFattrInquire` is used to inquire about the specified attribute. To inquire about a specific attribute entry, use `CDFattrEntryInquire` (Section 4.7.12).

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 4.7.9). |
| `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 4.6.9. |
| `maxEntry` | For gAttributes this is the gEntry number used. For vAttributes this is the maximum rEntry number used. In either case 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 (see Section 4.7.25). |

**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);
}
.
.
```

### 4.7.12 CDFattrEntryInquire (Standard Interface)

```
CDFstatus CDFattrEntryInquire(
CDFid id,             /* in  -- CDF identifier. */
long  attrNum,        /* in  -- Attribute number. */
long  entryNum,       /* in  -- Entry number. */
long  *dataType,      /* out -- 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` (see Section 4.7.11). `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 for which to inquire an entry. This number may be determined with a call to `CDFattrNum` (see Section 4.7.9). |
| `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 rVariable (the rVariable being described in some way by the rEntry). |
| `dataType` | The data type of the specified entry. The data types are defined in Section 4.6.4. |
| `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 rVariable 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 used. */
long      dataType;                        /* 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 */
     .
     .
   }
}
```

### 4.7.13 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 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 4.7.9). |
| `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 rVariable (the rVariable being described in some way by the rEntry). |
| `dataType` | The data type of the specified entry. Specify one of the data types defined in Section 4.6.4. |
| `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 gEntry number zero (0) of the gAttribute `TITLE`. The second is to the variable scope attribute `VALIDs` for the rEntry that corresponds to the rVariable `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 number 0. */
static short TMPvalids = {15,30};     /* Value(s) of VALIDs attribute,
                                         rEntry for rVariable TMP. */
.
.
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);
.
.
```

### 4.7.14 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 4.7.9). |
| `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 rVariable (the rVariable being described in some way by the rEntry). |
| `value` | The value read. This buffer must be large enough to hold the value. The function `CDFattrEntryInquire` would be used to determine the entry data type and number of elements (of that data type). The 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 rEntry corresponding to the `PRES_LVL` rVariable (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 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");         /* The rEntry number
                                                is the rVariable
                                                number. */
if (entryN < 0) UserStatusHandler (entryN);  /* 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 + 1);
  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);
}
.
.
```

### 4.7.15 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 2.4.3 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 4.7.9). |
| `attrScope` | The new scope for the attribute. Specify one of the definite scopes defined in Section 4.6.9. |

**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 2.4.3). 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 rVariable record number in
                                       CDF. */
long numVars;                       /* Number of rVariables 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. */
    }
  }
  .
  .
```

### 4.7.16 CDFvarCreate (Standard Interface)

```
CDFstatus CDFvarCreate(
CDFid id,               /* in --  CDF identifier. */
char  *varName,         /* in --  rVariable 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 -- rVariable number. */
```

`CDFvarCreate` is used to create a new rVariable in a CDF. A variable (rVariable or zVariable) 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 rVariable 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 rVariable. Specify one of the data types defined in Section 4.6.4. |
| `numElements` | The number of elements of the data type at each value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (each value consists of the entire string). For all other data types this must always be one (1) — multiple elements at each value are not allowed for non-character string data types. |
| `recVariance` | The rVariable's record variance. Specify one of the variances defined in Section 4.6.8. |

| | |
|---|---|
| `dimVariances` | The rVariable's dimension variances. `dimVariances` is a 1-dimensional array containing one element per rVariable dimension. Each element of `dimVariances` specifies the corresponding dimension variance. For each dimension specify one of the variances defined in Section 4.6.8. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| `varNum` | The number assigned to the new rVariable. This number must be used in subsequent CDF function calls when referring to this rVariable. An existing rVariables's number may be determined with the `CDFvarNum` function. |

**Example(s)**

The following example will create several rVariables in a CDF whose rVariables are 2-dimensional. In this case `EPOCH`, `LAT`, and `LON` are independent rVariables, and `TMP` is a dependent rVariable.

```
.
.
#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 rVariable number. */
long        LONvarNum;                         /* LON rVariable number. */
long        TMPvarNum;                         /* TMP rVariable 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);
.
.
```

### 4.7.17 CDFvarNum (Standard Interface)

```
long CDFvarNum(
CDFid id,             /* in -- CDF identifier. */
char  *varName);      /* in -- rVariable name. */
```

`CDFvarNum` is used to determine the number associated with a given rVariable name. If the rVariable is found, `CDFvarNum` returns its number — which will be equal to or greater than zero (0). If an error occurs (e.g., the rVariable 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 rVariable 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 an rVariable number is needed.

#### Example(s)

In the following example `CDFvarNum` is used as an embedded function call when inquiring about an rVariable.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                          /* CDF identifier. */
CDFstatus status;                      /* Returned status code. */
char      varName[CDF_VAR_NAME_LEN+1]; /* rVariable name. */
long      dataType;                    /* Data type of the rVariable. */
long      numElements;                 /* Number of elements (of the
                                          data type). */
long      recVariance;                 /* 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 rVariable named `LATITUDE` was inquired. Note that if `LATITUDE` did not exist in the CDF, the call to `CDFvarNum` would have returned an error code. Passing that error code to `CDFvarInquire` as an rVariable 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 rVariable names will be unknown — `CDFvarInquire` would be used to determine them. `CDFvarInquire` is described in Section 4.7.19.

### 4.7.18 CDFvarRename (Standard Interface)

```
CDFstatus CDFvarRename(
CDFid id,             /* in -- CDF identifier. */
long  varNum,         /* in -- rVariable number. */
char  *varName);      /* in -- New name. */
```

`CDFvarRename` is used to rename an existing rVariable. A variable (rVariable or zVariable) with the same 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 rVariable to rename. This number may be determined with a call to `CDFvarNum` (see Section 4.7.17). |
| `varName` | The new rVariable 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 rVariable named `TEMPERATURE` is renamed `TMP` (if it exists). Note that if `CDFvarNum` returns a value less than zero (0) then that value is not an rVariable number but rather a warning/error code.

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

```
long      varNum;   /* rVariable 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);
}
.
.
```

### 4.7.19 CDFvarInquire (Standard Interface)

```
CDFstatus CDFvarInquire(
CDFid id,                          /* in  -- CDF identifier. */
long  varNum,                      /* in  -- rVariable number. */
char  varName,                     /* out -- rVariable 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 rVariable. This function would normally be used before reading rVariable 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 rVariable to inquire. This number may be determined with a call to `CDFvarNum` (see Section 4.7.17). |
| `varName` | The rVariable'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 rVariable. The data types are defined in Section 4.6.4. |
| `numElements` | The number of elements of the data type at each rVariable value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string. (Each value consists of the entire string.) For all other data types, this will always be one (1) — multiple elements at each value are not allowed for non-character string data types. |
| `recVariance` | The record variance. The record variances are defined in Section 4.6.8. |

`dimVariances` The dimension variances. `dimVariances` is a 1-dimensional array containing one element per rVariable dimension in the CDF. Each element of `dimVariances` receives the corresponding dimension variance. The dimension variances are defined in Section 4.6.8. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary).

**Example(s)**

The following example inquires about an rVariable named `HEAT_FLUX` in a CDF. Note that the rVariable 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];  /* rVariable name, +1
                                           for NUL terminator. */
long      dataType;                     /* Data type of the rVariable. */
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);
.
.
```

### 4.7.20 CDFvarPut (Standard Interface)

```
CDFstatus CDFvarPut(
CDFid id,             /* in -- CDF identifier. */
long  varNum,         /* in -- rVariable number. */
long  recNum,         /* in -- Record number. */
long  indices[],      /* in -- Dimension indices. */
void  *value);        /* in -- Value. */
```

`CDFvarPut` is used to write a single value to an rVariable. `CDFvarHyperPut` may be used to write more than one rVariable value with a single call (see Section 4.7.22).

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 rVariable to which to write. This number may be determined with a call to `CDFvarNum` (see Section 4.7.17). |
| `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 rVariable dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional rVariables 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 rVariable named `LATITUDE` in a CDF whose rVariables are 2-dimensional with 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;                     /* rVariable 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
                                            rVariable 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 are "virtually" the same at each index of the first dimension.)

### 4.7.21 CDFvarGet (Standard Interface)

```
CDFstatus CDFvarGet(
CDFid id,             /* in  -- CDF identifier. */
long  varNum,         /* in  -- rVariable number. */
long  recNum,         /* in  -- Record number. */
long  indices[],      /* in  -- Dimension indices. */
void  *value);        /* out -- Value. */
```

CDFvarGet is used to read a single value from an rVariable. CDFvarHyperGet may be used to read more than one rVariable value with a single call (see Section 4.7.23).

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 rVariable from which to read. This number may be determined with a call to CDFvarNum (see Section 4.7.17). |
| 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 rVariable dimension. Each element of indices specifies the corresponding dimension index. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| value | The value read. This buffer must be large enough to hold the value. CDFvarInquire would be used to determine the rVariable's data type and number of elements (of that data type) at each 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 an rVariable. The CDF's rVariables are 3-dimensional with sizes [180,91,10]. For this rVariable 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;              /* rVariable 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);
      }
   }
}
.
.
```

### 4.7.22 CDFvarHyperPut (Standard Interface)

```
CDFstatus CDFvarHyperPut(
CDFid id,             /* in -- CDF identifier. */
long  varNum,         /* in -- rVariable 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 an rVariable. 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. `CDFinquire` is used to determine the variable majority of a CDF. Section 2.3.11 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 rVariable to which to write. This number may be determined with a call to `CDFvarNum` (see Section 4.7.17). |
| `recStart` | The record number at which to start writing. |
| `recCount` | The number of records to write. |
| `recInterval` | The interval between records for subsampling[4] (e.g., 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 rVariable dimension. Each element of `indices` specifies the corresponding dimension index. If there are zero (0) dimensions, 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 rVariable dimension. Each element of `count` specifies the corresponding dimension count. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| `intervals` | For each dimension the interval between values for subsampling[5] (e.g., an interval of 2 means write to every other value). `intervals` is a 1-dimensional array containing one element per rVariable dimension. Each element of `intervals` specifies the corresponding dimension interval. If there are zero (0) dimensions, 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 rVariable `LATITUDE` of a CDF whose rVariables are 2-dimensional with 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 4.7.20 except that it 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;                   /* rVariable number. */
```

[4] "Subsampling" is not the best term to use when writing data, but you should know what we mean.
[5] Again, not the best term.

```
long recStart = 0;                    /* Record number. */
long recCount = 1;                    /* Record counts. */
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 an
                                            rVariable 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);
.
.
```

### 4.7.23 CDFvarHyperGet (Standard Interface)

```
CDFstatus CDFvarHyperGet(
CDFid id,             /* in --  CDF identifier. */
long  varNum,         /* in --  rVariable 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 an rVariable. 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. `CDFinquire` is used to determine the variable majority of a CDF. Section 2.3.11 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 rVariable from which to read. This number may be determined with a call to `CDFvarNum` (see Section 4.7.17). |
| `recStart` | The record number at which to start reading. |
| `recCount` | The number of records to read. |

| | |
|---|---|
| `recInterval` | The interval between records for subsampling (e.g., 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 rVariable dimension. Each element of `indices` specifies the corresponding dimension index. If there are zero (0) dimensions, 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 rVariable dimension. Each element of `counts` specifies the corresponding dimension count. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| `intervals` | For each dimension, the interval between values for subsampling (e.g., an interval of 2 means read every other value). `intervals` is a 1-dimensional array containing one element per rVariable dimension. Each element of `intervals` specifies the corresponding dimension interval. If there are zero (0) dimensions, 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 rVariable'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 an rVariable. The CDF's rVariables are 3-dimensional with sizes `[180,91,10]` and CDF's variable majority is `ROW_MAJOR`. For the rVariable 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 4.7.21 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;                       /* rVariable 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]` for proper indexing.

### 4.7.24 CDFvarClose (Standard Interface)

```
CDFstatus CDFvarClose(
CDFid id,             /* in --  CDF identifier. */
long  varNum);        /* in --  rVariable number. */
```

`CDFvarClose` is used to close an rVariable in a multi-file CDF. This function is not applicable to single-file CDFs. The use of `CDFvarClose` is not required since the CDF library automatically closes the rVariable files when a multi-file CDF is closed or when there are insufficient file pointers available (because of an open file quota) to keep all of the rVariable files open. `CDFvarClose` would be used by an application since it knows best how its rVariables are going to be accessed. Closing an rVariable would also free the cache buffers that are associated with the rVariable's file. This could be important in those situations where memory is limited (e.g., the IBM PC). The caching scheme used by the CDF library is described in Section 2.1.4. Note that there is not a function that opens an rVariable. The CDF library automatically opens an rVariable when it is accessed by an application (unless it is already open).

The arguments to `CDFvarClose` 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 rVariable to close. This number may be determined with a call to `CDFvarNum` (see Section 4.7.17). |

**Example(s)**

The following example will close an rVariable in a multi-file CDF.

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

```
CDFid       id;                          /* CDF identifier. */
CDFstatus   status;                      /* Returned status code. */
.
.
status = CDFvarClose (id, CDFvarNum(id,"Flux"));
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

### 4.7.25 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 (e.g., specifying a single-file format for a CDF, accessing zVariables, or specifying a pad value for an rVariable or zVariable). 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 that specify one or more operations to be performed (e.g., opening a CDF, creating an attribute, or writing a variable value). 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 (e.g., a CDF, variable, or attribute) or a state (e.g., a CDF's format, a variable's data specification, or a CDF's current attribute). 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 shows how a CDF would be created with the single-file format (assuming multi-file is the default).

```
.
.
#include "cdf.h"
.
.
CDFid       id;                          /* CDF identifier (handle). */
CDFstatus   status;                      /* Status returned from CDF
                                            library. */
static char CDFpath[] = {"test1"};       /* Pathname 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 (CDFpath, numDims, dimSizes, encoding, majority, &id);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (PUT_, CDF_FORMAT_, format,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

The call to `CDFcreate` created the CDF as expected but with a format of multi-file. 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:

| | |
|---|---|
| `PUT_` | The first operation to be performed. In this case an item is going to be put to the "current" CDF (a new format). `PUT_` is defined in `cdf.h` (as are all CDF constants). It was not necessary to select a current CDF since the call to `CDFcreate` implicitly selected the CDF created as the current CDF.[6] This is the case since all of the Standard Interface functions actually call the Internal Interface to perform their operations. |
| `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. |
| `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_, CDFpath, numDims, dimSizes, &id,
                 PUT_, CDF_ENCODING_, encoding,
                       CDF_MAJORITY_, majority,
                       CDF_FORMAT_, format,
```

[6] In previous releases of CDF, it was required that the current CDF be selected in each call to `CDFlib`. That requirement has been eliminated. The CDF library now maintains the current CDF from one call to the next of `CDFlib`.

```
                    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 that must follow. When a CDF is created (with `CDFlib`), the format, encoding, and majority default to values specified when your CDF distribution was built and installed. Consult your system manager for these defaults. |
| `CDFpath` | The pathname 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 into the CDF. |
| `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 implicitly 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 that 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 current objects/states are used by the CDF library when performing operations.

CDF (object)

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_>`[7] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.[8]

rVariable (object)

An rVariable operation is always performed on the current rVariable in the current CDF. For each open CDF a current rVariable is maintained. This current rVariable is implicitly selected when an rVariable is created (in the current CDF) or it may be explicitly selected with the `<SELECT_,rVAR_>` or `<SELECT_,rVAR_NAME_>` operations. There is no current rVariable in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

zVariable (object)

A zVariable operation is always performed on the current zVariable in the current CDF. For each open CDF a current zVariable is maintained. This current zVariable is implicitly selected when a zVariable is created (in the current CDF) or it may be explicitly selected with the `<SELECT_,zVAR_>` or `<SELECT_,zVAR_NAME_>` operations. There is no current zVariable in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

attribute (object)

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.

gEntry number (state)

A gAttribute gEntry operation is always performed on the current gEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current gEntry number is maintained. This current gEntry number must be explicitly selected with the `<SELECT_,gENTRY_>` operation. (There is no implicit or default selection of the current gEntry number for a CDF.) Note that the current gEntry number is maintained for the CDF (not each attribute) — it applies to all of the attributes in that CDF.

rEntry number (state)

A vAttribute rEntry operation is always performed on the current rEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current rEntry number is maintained. This current rEntry number must be explicitly selected with the `<SELECT_,rENTRY_>` operation. (There is no implicit or default selection of the current rEntry number for a CDF.) Note that the current rEntry number is maintained for the CDF (not each attribute) — it applies to all of the attributes in that CDF.

[7] This notation is used to specify an operation to be performed on an item. The syntax is `<operation,item>`.

[8] In previous releases of CDF, it was required that the current CDF be selected in each call to `CDFlib`. That requirement no longer exists. The CDF library now maintains the current CDF from one call to the next of `CDFlib`.

zEntry number (state)

A vAttribute zEntry operation is always performed on the current zEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current zEntry number is maintained. This current zEntry number must be explicitly selected with the <SELECT_,zENTRY_> operation. (There is no implicit or default selection of the current zEntry number for a CDF.) Note that the current zEntry number is maintained for the CDF (not each attribute) — it applies to all of the attributes in that CDF.

record number, rVariables (state)

An rVariable read or write operation is always performed at (for single and multiple variable reads and writes) or starting at (for hyper reads and writes) the current record number for the rVariables in the current CDF. When a CDF is opened or created, the current record number for its rVariables is initialized to zero (0). It may then be explicitly selected using the <SELECT_,rVARs_RECNUMBER_> operation. Note that the current record number for rVariables is maintained for a CDF (not each rVariable) — it applies to all of the rVariables in that CDF.

record count, rVariables (state)

An rVariable hyper read or write operation is always performed using the current record count for the rVariables in the current CDF. When a CDF is opened or created, the current record count for its rVariables is initialized to one (1). It may then be explicitly selected using the <SELECT_,rVARs_RECCOUNT_> operation. Note that the current record count for rVariables is maintained for a CDF (not each rVariable) — it applies to all of the rVariables in that CDF.

record interval, rVariables (state)

An rVariable hyper read or write operation is always performed using the current record interval for the rVariables in the current CDF. When a CDF is opened or created, the current record interval for its rVariables is initialized to one (1). It may then be explicitly selected using the <SELECT_,rVARs_RECINTERVAL_> operation. Note that the current record interval for rVariables is maintained for a CDF (not each rVariable) — it applies to all of the rVariables in that CDF.

dimension indices, rVariables (state)

An rVariable read or write operation is always performed at (for single reads and writes) or starting at (for hyper reads and writes) the current dimension indices for the rVariables in the current CDF. When a CDF is opened or created, the current dimension indices for its rVariables are initialized to zeroes (0,0,...). They may then be explicitly selected using the <SELECT_,rVARs_DIMINDICES_> operation. Note that the current dimension indices for rVariables are maintained for a CDF (not each rVariable) — they apply to all of the rVariables in that CDF. For 0-dimensional rVariables the current dimension indices are not applicable.

dimension counts, rVariables (state)

An rVariable hyper read or write operation is always performed using the current dimension counts for the rVariables in the current CDF. When a CDF is opened or created, the current dimension counts for its rVariables are initialized to the dimension sizes of the rVariables (which specifies the entire array). They may then be explicitly selected using the <SELECT_,rVARs_DIMCOUNTS_> operation. Note that the current dimension counts for rVariables are maintained for a CDF (not each rVariable) — they apply to all of the rVariables in that CDF. For 0-dimensional rVariables the current dimension counts are not applicable.

dimension intervals, rVariables (state)

An rVariable hyper read or write operation is always performed using the current dimension intervals

for the rVariables in the current CDF. When a CDF is opened or created, the current dimension intervals for its rVariables are initialized to ones (1,1,...). They may then be explicitly selected using the <SELECT_,rVARs_DIMINTERVALS_> operation. Note that the current dimension intervals for rVariables are maintained for a CDF (not each rVariable) — they apply to all of the rVariables in that CDF. For 0-dimensional rVariables the current dimension intervals are not applicable.

sequential value, rVariable (state)

An rVariable sequential read or write operation is always performed at the current sequential value for that rVariable. When an rVariable is created (or for each rVariable in a CDF being opened), the current sequential value is set to the first physical value (even if no physical values exist yet). It may then be explicitly selected using the <SELECT_,rVAR_SEQPOS_> operation. Note that a current sequential value is maintained for each rVariable in a CDF.

record number, zVariable (state)

A zVariable read or write operation is always performed at (for single reads and writes) or starting at (for hyper reads and writes) the current record number for the current zVariable in the current CDF. A multiple variable read or write operation is performed at the current record number of each of the zVariables involved. (The record numbers do not have to be the same.) When a zVariable is created (or for each zVariable in a CDF being opened), the current record number for that zVariable is initialized to zero (0). It may then be explicitly selected using the <SELECT_,zVAR_RECNUMBER_> operation (which only affects the current zVariable in the current CDF). Note that a current record number is maintained for each zVariable in a CDF.

record count, zVariable (state)

A zVariable hyper read or write operation is always performed using the current record count for the current zVariable in the current CDF. When a zVariable created (or for each zVariable in a CDF being opened), the current record count for that zVariable is initialized to one (1). It may then be explicitly selected using the <SELECT_,zVAR_RECCOUNT_> operation (which only affects the current zVariable in the current CDF). Note that a current record count is maintained for each zVariable in a CDF.

record interval, zVariable (state)

A zVariable hyper read or write operation is always performed using the current record interval for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current record interval for that zVariable is initialized to one (1). It may then be explicitly selected using the <SELECT_,zVAR_RECINTERVAL_> operation (which only affects the current zVariable in the current CDF). Note that a current record interval is maintained for each zVariable in a CDF.

dimension indices, zVariable (state)

A zVariable read or write operation is always performed at (for single reads and writes) or starting at (for hyper reads and writes) the current dimension indices for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current dimension indices for that zVariable are initialized to zeroes (0,0,...). They may then be explicitly selected using the <SELECT_,zVAR_DIMINDICES_> operation (which only affects the current zVariable in the current CDF). Note that current dimension indices are maintained for each zVariable in a CDF. For 0-dimensional zVariables the current dimension indices are not applicable.

dimension counts, zVariable (state)

A zVariable hyper read or write operation is always performed using the current dimension counts for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable

in a CDF being opened), the current dimension counts for that zVariable are initialized to the dimension sizes of that zVariable (which specifies the entire array). They may then be explicitly selected using the <SELECT_,zVAR_DIMCOUNTS_> operation (which only affects the current zVariable in the current CDF). Note that current dimension counts are maintained for each zVariable in a CDF. For 0-dimensional zVariables the current dimension counts are not applicable.

dimension intervals, zVariable (state)

A zVariable hyper read or write operation is always performed using the current dimension intervals for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current dimension intervals for that zVariable are initialized to ones (1,1,...). They may then be explicitly selected using the <SELECT_,zVAR_DIMINTERVALS_> operation (which only affects the current zVariable in the current CDF). Note that current dimension intervals are maintained for each zVariable in a CDF. For 0-dimensional zVariables the current dimension intervals are not applicable.

sequential value, zVariable (state)

A zVariable sequential read or write operation is always performed at the current sequential value for that zVariable. When a zVariable is created (or for each zVariable in a CDF being opened), the current sequential value is set to the first physical value (even if no physical values exist yet). It may then be explicitly selected using the <SELECT_,zVAR_SEQPOS_> operation. Note that a current sequential value is maintained for each zVariable in a CDF.

status code (state)

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.[9]

### 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 4.8 explains how to interpret status codes. Appendix C lists the possible status codes and the type of each: error, warning, or informational.

[9] The CDF library now maintains the current status code from one call to the next of `CDFlib`.

### 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_, CDFpath, numDims, dimSizes, &id,
                 PUT_, CDF_FORMAT_, format,
                       CDF_MAJORITY_, majority,
                 CREATE_, ATTR_, attrName, scope, &attrNum,
                          rVAR_, 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 rVAR_) 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_, CDFpath, numDims, dimSizes, &id, PUT_,
                 CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
                 ATTR_, attrName, scope, &attrNum, rVAR_, varName, dataType,
                 numElements, recVary, dimVarys, &varNum, NULL_);
```

The need for proper indentation to ensure the readability of your applications should be obvious.[10]

### Syntax

CDFlib takes 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.

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,
                      .
                      .
```

[10] If it isn't, call us — we can help you.

```
        itemN, arg1, arg2, ...argN,
    .
    .
    .
    opN, item1, arg1, arg2, ...argN,
        item2, arg1, arg2, ...argN,
        .
        .
        itemN, arg1, arg2, ...argN,
    NULL_);
```

where opx is an operation to perform, itemx is the item on which to perform the operation, and argx is a required argument for the operation. The NULL_ operation must be used to end the call to CDFlib. The completion status, status, is returned.

### Operations/Items/Arguments...

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, required arguments, and required preselected objects/states are listed in the following sections. The required preselected objects/states are those objects/states that must be selected (typically with the SELECT_ operation) before a particular operation may be performed. Note that some of the required preselected objects/states have default values as described beginning on page 149.

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 as follows:

in: char *CDFpath

Pathname of the CDF to be created. (Do not append an extension.) This can be at most CDF_PATHNAME_LEN characters (excluding the NUL terminator). 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.

in: `long numDims`

Number of dimensions for the rVariables. This can be as few as zero (0) and at most `CDF_MAX_DIMS`. Note that this must be specified even if the CDF will contain only zVariables.

in: `long dimSizes[]`

Dimension sizes for the rVariables. `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 rVariables this argument is ignored (but a place holder is necessary). Note that this must be specified even if the CDF will contain only zVariables.

out: `CDFid *id`

CDF identifier to be used in subsequent operations on the CDF.

A CDF is created with the default format, encoding, and variable majority as specified in the configuration file of your CDF distribution. Consult your system manager to determine these defaults. These defaults can be changed with the corresponding `PUT_` operations of `CDF_FORMAT_`, `CDF_ENCODING_`, and `CDF_MAJORITY_`.

A CDF must be closed with `<CLOSE_,CDF_>` to ensure that the CDF will be correctly written to disk.

There are no required preselected objects/states.

`rVAR_`

A new rVariable will be created in the current CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF. The created rVariable implicitly becomes the current rVariable (in the current CDF). Required arguments are as follows:

in: `char *varName`

Name of the rVariable 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 rVariable. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type at each 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 value of the variable. For the non-character string data types this must be one (1) — multiple elements are not allowed for non-character string data types.

in: `long recVary`

Record variance. Specify one of the variances described in

Section 4.6.8.

in: `long dimVarys[]`

Dimension variances. `dimVarys` is a 1-dimensional array containing one element per rVariable dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.6.8. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

out: `long *varNum`

Number assigned to the new rVariable. This number must be used in subsequent CDF function calls when referring to this rVariable. An existing rVariable's number may also be determined with the `<GET_,rVAR_NUMBER_>` operation.

The only required preselected object/state is the current CDF.

`zVAR_` A new zVariable will be created in the current CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF. The created zVariable implicitly becomes the current zVariable (in the current CDF). Required arguments are as follows:

in: `char *varName`

Name of the zVariable to be created. This can be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator). Variable names are case-sensitive.

in: `long dataType`

Data type of the new zVariable. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type at each 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 value of the variable. For the non-character string data types this must be one (1) — multiple elements are not allowed for non-character string data types.

in: `long numDims`

Number of dimensions for the zVariable. This may be as few as zero and at most `CDF_MAX_DIMS`.

in: `long dimSizes[]`

The 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 a 0-dimensional zVariable this argument is ignored (but a place holder is necessary).

in: `long recVary`

Record variance. Specify one of the variances described in Section 4.6.8.

in: `long dimVarys[]`

Dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.6.8. For a 0-dimensional zVariable this argument is ignored (but a place holder is necessary).

out: `long *varNum`

Number assigned to the new zVariable. This number must be used in subsequent CDF function calls when referring to this zVariable. An existing zVariable's number may also be determined with the `<GET_,zVAR_NUMBER_>` operation.

The only required preselected object/state is the current CDF.

`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 as follows:

in: `char *attrName`

Name of the attribute to be created. This can be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator). Attribute names are case-sensitive.

in: `long scope`

Scope of the new attribute. Specify one of the scopes described in Section 4.6.9.

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.

The only required preselected object/state is the current CDF.

`OPEN_` The following item(s) may be opened:

`CDF_` Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

in: `char *CDFpath`

Pathname of the CDF to be opened. (Do not append an extension.) This can be at most `CDF_PATHNAME_LEN` characters (excluding the `NUL` terminator). 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.

out: `CDFid *id`

CDF identifier to be used in subsequent operations on the CDF.

There are no required preselected objects/states.

`DELETE_` The following item(s) may be deleted:

`CDF_` Deletes the current CDF. A CDF must be opened before it can be deleted. When the CDF is deleted, there is no longer a current CDF.

There are no required arguments.

The only required preselected object/state is the current CDF.

`CLOSE_` The following items may be closed:

`CDF_` Closes the current CDF. When the 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 required arguments.

The only required preselected object/state is the current CDF.

`rVAR_` Closes the current rVariable (in the current CDF). This operation is only applicable to multi-file CDFs.

There are no required arguments.

The required preselected objects/states are the current CDF and its current rVariable.

`zVAR_` Closes the current zVariable (in the current CDF). This operation is only applicable to multi-file CDFs.

There are no required arguments.

The required preselected objects/states are the current CDF and its current zVariable.

`SELECT_` The following items may be selected:

`CDF_` Explicitly selects the current CDF. Required arguments are as follows:

in: `CDFid id`

Identifier of the CDF. This identifier must have been initialized by a successful `<CREATE_,CDF_>` or `<OPEN_,CDF_>` operation.

There are no required preselected objects/states.

`CDF_STATUS_` Selects the current status code. Required arguments are as follows:

in: `CDFstatus status`

CDF status code.

There are no required preselected objects/states.

`CDF_READONLY_MODE_` Selects a read-only mode (for the current CDF). Required arguments are as follows:

in: `long mode`

The read-only mode. Specify one of the read-only modes described in Section 4.6.10.

The only required preselected object/state is the current CDF.

`CDF_zMODE_` Selects a zMode (for the current CDF). Required arguments are as follows:

in: `long mode`

The zMode. Specify one of the zModes described in Section 4.6.11.

The only required preselected object/state is the current CDF.

`CDF_NEGtoPOSfp0_MODE_`

Selects a `-0.0` to `0.0` mode (for the current CDF). Required arguments are as follows:

in: `long mode`

The `-0.0` to `0.0` mode. Specify one of the `-0.0` to `0.0` modes described in Section 4.6.12.

The only required preselected object/state is the current CDF.

`CDF_DECODING_` Selects a decoding (for the current CDF). Required arguments are as follows:

in: `long decoding`

The decoding. Specify one of the decodings described in Section 4.6.6.

The only required preselected object/state is the current CDF.

`CDF_CACHESIZE_` Selects the number of cache buffers to be used for the "`.cdf`" file (for the current CDF). Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`rVAR_` Explicitly selects the current rVariable (in the current CDF) by number. Required arguments are as follows:

in: `long varNum`

rVariable number.

The only required preselected object/state is the current CDF.

`rVAR_NAME_` Explicitly selects the current rVariable (in the current CDF) by name. Required arguments are as follows:

in: `char *varName`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`rVARs_RECNUMBER_` Selects the current record number for all rVariables in the current CDF. Required arguments are as follows:

in: `long recNum`

Record number.

The only required preselected object/state is the current CDF.

`rVARs_RECCOUNT_` Selects the current record count for all rVariables in the current CDF. Required arguments are as follows:

in: `long recCount`

Record count.

The only required preselected object/state is the current CDF.

`rVARs_RECINTERVAL_` Selects the current record interval for all rVariables in the current CDF. Required arguments are as follows:

in: `long recInterval`

Record interval.

The only required preselected object/state is the current CDF.

`rVARs_DIMINDICES_` Selects the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index.

The only required preselected object/state is the current CDF.

`rVARs_DIMCOUNTS_` Selects the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long counts[]`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` specifies the corresponding dimension count.

The only required preselected object/state is the current CDF.

`rVARs_DIMINTERVALS_` Selects the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long intervals[]`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` specifies the corresponding dimension interval.

The only required preselected object/state is the current CDF.

`rVAR_SEQPOS_` Selects the current sequential value for sequential access for the current rVariable (in the current CDF). Note that a current sequential value is maintained for each rVariable individually. Required arguments are as follows:

in: `long recNum`

Record number.

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per rVariable dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_CACHESIZE_` Selects the number of cache buffers to be used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The required preselected objects/states are the current CDF and its current rVariable.

`rVARs_CACHESIZE_` Selects the number of cache buffers to be used for all of the rVariable files (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`zVAR_` Explicitly selects the current zVariable (in the current CDF) by number. Required arguments are as follows:

in: `long varNum`

zVariable number.

The only required preselected object/state is the current CDF.

`zVAR_NAME_` Explicitly selects the current zVariable (in the current CDF) by name. Required arguments are as follows:

in: `char *varName`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`zVARs_RECNUMBER_` Selects the current record number for each zVariable in the current CDF. This operation is provided to simplify the selection of the current record numbers for the zVariables involved in a multiple variable access operation (see Section 2.3.15). Required arguments are as follows:

in: `long recNum`

Record number.

The only required preselected object/state is the current CDF.

`zVAR_RECNUMBER_` Selects the current record number for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recNum`

Record number.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECCOUNT_` Selects the current record count for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recCount`

Record count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECINTERVAL_` Selects the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recInterval`

Record interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINDICES_` Selects the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMCOUNTS_` Selects the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long counts[]`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` specifies the corresponding dimension count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINTERVALS_` Selects the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long intervals[]`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` specifies the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_SEQPOS_` Selects the current sequential value for sequential access for the current zVariable (in the current CDF). Note that a current sequential value is maintained for each zVariable individually. Required arguments are as follows:

in: `long recNum`

Record number.

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_CACHESIZE_` Selects the number of cache buffers to be used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The required preselected objects/states are the current CDF and its current zVariable.

`zVARs_CACHESIZE_` Selects the number of cache buffers to be used for all of the zVariable files (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`ATTR_` Explicitly selects the current attribute (in the current CDF) by number. Required arguments are as follows:

in: `long attrNum`

Attribute number.

The only required preselected object/state is the current CDF.

`ATTR_NAME_` Explicitly selects the current attribute (in the current CDF) by name. Required arguments are as follows:

in: `char *attrName`

Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`gENTRY_` Selects the current gEntry number for all gAttributes in the current CDF. **NOTE:** Only use this operation when dealing with gAttributes. Required arguments are as follows:

in: `long entryNum`

gEntry number.

The only required preselected object/state is the current CDF.

`rENTRY_` Selects the current rEntry number for all vAttributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `long entryNum`

rEntry number.

The only required preselected object/state is the current CDF.

`rENTRY_NAME_` Selects the current rEntry number for all vAttributes (in the current CDF) by rVariable name. The number of the named rVariable becomes the current rEntry number. (The current rVariable is not changed.) **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `char *varName`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`zENTRY_` Selects the current zEntry number for all vAttributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `long entryNum`

zEntry number.

The only required preselected object/state is the current CDF.

`zENTRY_NAME_` Selects the current zEntry number for all vAttributes (in the current CDF) by zVariable name. The number of the named zVariable becomes the current zEntry number. (The current zVariable is not changed.) **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `char *varName`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`CONFIRM_` The following items may be confirmed:

`CDF_NAME_` Confirms the pathname of the current CDF. Required arguments are as follows:

out: `char *CDFpath`

Pathname of the CDF. This character string must be large enough to hold `CDF_PATHNAME_LEN + 1` characters (including the `NUL` terminator).

The only required preselected object/state is the current CDF.

`CDF_READONLY_MODE_` Confirms the read-only mode for the current CDF. Required arguments are as follows:

out: `long *mode`

The read-only mode. The read-only modes are described in Section 4.6.10.

The only required preselected object/state is the current CDF.

`CDF_zMODE_` Confirms the zMode for the current CDF. Required arguments are as follows:

in: `long *mode`

The zMode. The zModes are described in Section 4.6.11.

The only required preselected object/state is the current CDF.

`CDF_NEGtoPOSfp0_MODE_`

Confirms the `-0.0` to `0.0` mode for the current CDF. Required arguments are as follows:

in: `long *mode`

The `-0.0` to `0.0` mode. The `-0.0` to `0.0` modes are described in Section 4.6.12.

The only required preselected object/state is the current CDF.

`CDF_DECODING_` Confirms the decoding for the current CDF. Required arguments are as follows:

in: `long *decoding`

The decoding. The decodings are described in Section 4.6.6.

The only required preselected object/state is the current CDF.

`CDF_CACHESIZE_` Confirms the number of cache buffers being used for the "`.cdf`" file (for the current CDF). Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long *numBuffers`

The number of cache buffers being used.

The only required preselected object/state is the current CDF.

`rVAR_` Confirms the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *varNum`

rVariable number.

The only required preselected object/state is the current CDF.

`rVAR_EXISTENCE_` Confirms the existence of the named rVariable (in the current CDF). If the rVariable does not exist, an error code will be returned. In any case the current rVariable is not affected. Required arguments are as follows:

out: `char *varName`

The rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`rVARs_RECNUMBER_` Confirms the current record number for all rVariables in the current CDF. Required arguments are as follows:

out: `long *recNum`

Record number.

The only required preselected object/state is the current CDF.

`rVARs_RECCOUNT_` Confirms the current record count for all rVariables in the current CDF. Required arguments are as follows:

out: `long *recCount`

Record count.

The only required preselected object/state is the current CDF.

`rVARs_RECINTERVAL_` Confirms the current record interval for all rVariables in the current CDF. Required arguments are as follows:

out: `long *recInterval`

Record interval.

The only required preselected object/state is the current CDF.

`rVARs_DIMINDICES_` Confirms the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index.

The only required preselected object/state is the current CDF.

`rVARs_DIMCOUNTS_` Confirms the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long counts[]`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` receives the corresponding dimension count.

The only required preselected object/state is the current CDF.

`rVARs_DIMINTERVALS_` Confirms the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applica-

ble. Required arguments are as follows:

out: `long intervals[]`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` receives the corresponding dimension interval.

The only required preselected object/state is the current CDF.

`rVAR_SEQPOS_` Confirms the current sequential value for sequential access for the current rVariable (in the current CDF). Note that a current sequential value is maintained for each rVariable individually. Required arguments are as follows:

out: `long *recNum`

Record number.

out: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_CACHESIZE_` Confirms the number of cache buffers being used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long *numBuffers`

The number of cache buffers being used.

The required preselected objects/states are the current CDF and its current rVariable.

`zVAR_` Confirms the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *varNum`

zVariable number.

The only required preselected object/state is the current CDF.

`zVAR_EXISTENCE_` Confirms the existence of the named zVariable (in the current CDF). If the zVariable does not exist, an error code will be returned. In any case the current zVariable is not affected. Required arguments are as follows:

out: `char *varName`

The zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`zVAR_RECNUMBER_` Confirms the current record number for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *recNum`

Record number.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECCOUNT_` Confirms the current record count for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *recCount`

Record count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECINTERVAL_` Confirms the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *recInterval`

Record interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINDICES_` Confirms the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMCOUNTS_` Confirms the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long counts[]`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` receives the corresponding dimension count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINTERVALS_` Confirms the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long intervals[]`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` receives the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_SEQPOS_` Confirms the current sequential value for sequential access for the current zVariable (in the current CDF). Note that a current sequential value is maintained for each zVariable individually. Required arguments are as follows:

out: `long *recNum`

Record number.

out: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_CACHESIZE_` Confirms the number of cache buffers being used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long *numBuffers`

The number of cache buffers being used.

The required preselected objects/states are the current CDF and its current zVariable.

`ATTR_` Confirms the current attribute (in the current CDF). Required arguments are as follows:

out: `long *attrNum`

Attribute number.

The only required preselected object/state is the current CDF.

`ATTR_EXISTENCE_` Confirms the existence of the named attribute (in the current CDF). If the attribute does not exist, an error code will be returned. In any case the current attribute is not affected. Required arguments are as follows:

out: `char *attrName`

The attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`gENTRY_` Confirms the current gEntry number for all attributes in the current CDF. **NOTE:** Only use this operation when dealing with gAttributes. Required arguments are as follows:

out: `long *entryNum`

The gEntry number.

The only required preselected object/state is the current CDF.

`rENTRY_` Confirms the current rEntry number for all attributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

out: `long *entryNum`

The rEntry number.

The only required preselected object/state is the current CDF.

`zENTRY_` Confirms the current zEntry number for all attributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

out: `long *entryNum`

The zEntry number.

The only required preselected object/state is the current CDF.

`gENTRY_EXISTENCE_` Confirms the existence of the specified gEntry (for the current attribute in the current CDF). If the gEntry does not exist, an error code will be returned. In any case the current gEntry number is not affected. **NOTE:** Only use this operation when dealing with gAttributes. Required arguments are as follows:

in: `long entryNum`

The gEntry number.

The required preselected objects/states are the current CDF and its current attribute.

`rENTRY_EXISTENCE_` Confirms the existence of the specified rEntry (for the current attribute in the current CDF). If the rEntry does not exist, an error code will be returned. In any case the current rEntry number is not

affected. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `long entryNum`

The rEntry number.

The required preselected objects/states are the current CDF and its current attribute.

`zENTRY_EXISTENCE_` Confirms the existence of the specified zEntry (for the current attribute in the current CDF). If the zEntry does not exist, an error code will be returned. In any case the current zEntry number is not affected. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `long entryNum`

The zEntry number.

The required preselected objects/states are the current CDF and its current attribute.

`PUT_` The following items may be written (put):

`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 pad value) or attribute entries have been written. Required arguments are as follows:

in: `long encoding`

New data encoding. Specify one of the encodings described in Section 4.6.5.

The only required preselected object/state is the current CDF.

`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 as follows:

in: `long majority`

New variable majority. Specify one of the majorities described in Section 4.6.7.

The only required preselected object/state is the current CDF.

`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 as follows:

in: `long format`

New CDF format. Specify one of the formats described in Section 4.6.3.

The only required preselected object/state is the current CDF.

`rVAR_NAME_` Renames the current rVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *varName`

New name of the rVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATASPEC_` Respecifies the data specification (data type and number of elements) of the current rVariable (in the current CDF). An rVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see Section 2.2.9) and the number of elements are the same. Required arguments are as follows:

in: `long dataType`

New data type. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type at each 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 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.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_RECVARY_` Respecifies the record variance of the current rVariable (in the current CDF). An rVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `long recVary`

New record variance. Specify one of the variances described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DIMVARYS_` Respecifies the dimension variances of the current rVariable (in the current CDF). An rVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long dimVarys[]`

New dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_PADVALUE_` Specifies the pad value for the current rVariable (in the current CDF). An rVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). Section 2.3.16 describes variable pad values. Required arguments are as follows:

in: `void *value`

Pad value. The pad value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_INITIALRECS_` Specifies the number of records to initially write to the current rVariable (in the current CDF). This may be specified only once per rVariable and before any other records have been written to that rVariable. If a pad value has not yet been specified, the default is used (see Section 2.3.16). If a pad value has been explicitly specified, that value is written to the records. Section 2.3.7 describes initial records. Required arguments are as follows:

in: `long NinitialRecs`

Number of records to write.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_EXTENDRECS_` Specifies the number of physical records to extend the current rVariable (in the current CDF) whenever that rVariable needs to be extended. Section 2.3.7 describes extend records. **NOTE:** The number of extend records has no effect for NRV rVariables or multi-file CDFs. Required arguments are as follows:

in: `long NextendRecs`

Number of records to extend. A value of zero (0) indicates that the default number of records should be used.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATA_` Writes one value to the current rVariable (in the current CDF). The value is written at the current record number and current dimension indices for the rVariables (in the current CDF). Required arguments

are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

`rVAR_HYPERDATA_` Writes one or more values to the current rVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

in: `void *buffer`

Values. The values starting at memory address `buffer` are written to the CDF.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

`rVAR_SEQDATA_` Writes one value to the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the rVariable, the rVariable is extended as necessary. Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note that the current sequential value for an rVariable increments automatically as values are written.

`rVARs_RECDATA_` Writes full-physical records to one or more rVariables (in the current CDF). The full-physical records are written at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of rVariables to which to write. This must be at least one (1).

in: `long varNums[]`

The rVariables to which to write. This array, whose size is

determined by the value of `numVars`, contains rVariable numbers. The rVariable numbers can be listed in any order.

in: `void *buffer`

The buffer of full-physical rVariable records to be written. The order of the full-physical rVariable records in this buffer must agree with the rVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical rVariable records. Be careful if using C `struct` objects to store multiple full-physical rVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `sturct` may not be contiguous). See Section 2.3.15 for more details on how to create this buffer.

The required preselected objects/states are the current CDF and its current record number for rVariables.

`zVAR_NAME_` Renames the current zVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *varName`

New name of the zVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATASPEC_` Respecifies the data specification (data type and number of elements) of the current zVariable (in the current CDF). A zVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see Section 2.2.9) and the number of elements are the same. Required arguments are as follows:

in: `long dataType`

New data type. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type at each 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 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.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECVARY_` Respecifies the record variance of the current zVariable (in the current CDF). A zVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `long recVary`

New record variance. Specify one of the variances described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMVARYS_` Respecifies the dimension variances of the current zVariable (in the current CDF). A zVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long dimVarys[]`

New dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_PADVALUE_` Specifies the pad value for the current zVariable (in the current CDF). A zVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). Section 2.3.16 describes variable pad values. Required arguments are as follows:

in: `void *value`

Pad value. The pad value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_INITIALRECS_` Specifies the number of records to initially write to the current zVariable (in the current CDF). This may be specified only once per zVariable and before any other records have been written to that zVariable. If a pad value has not yet been specified, the default is used (see Section 2.3.16). If a pad value has been explicitly specified, that value is written to the records. Section 2.3.7 describes initial records. Required arguments are as follows:

in: `long NinitialRecs`

Number of records to write.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_EXTENDRECS_` Specifies the number of physical records to extend the current zVariable (in the current CDF) whenever that zVariable needs to be extended. Section 2.3.7 describes extend records. **NOTE:** The number of extend records has no effect for NRV zVariables or multi-file CDFs. Required arguments are as follows:

in: `long NextendRecs`

Number of records to extend. A value of zero (0) indicates that the default number of records should be used.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATA_` Writes one value to the current zVariable (in the current CDF). The value is written at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the current dimension indices for the zVariable.

`zVAR_HYPERDATA_` Writes one or more values to the current zVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

in: `void *buffer`

Values. The values starting at memory address `buffer` are written to the CDF.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

`zVAR_SEQDATA_` Writes one value to the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the zVariable, the zVariable is extended as necessary. Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are written.

`zVARs_RECDATA_` Writes full-physical records to one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is written at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of zVariables to which to write. This must be at least one (1).

in: `long varNums[]`

The zVariables to which to write. This array, whose size is determined by the value of `numVars`, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: `void *buffer`

The buffer of full-physical zVariable records to be written. The order of the full-physical zVariable records in this buffer must agree with the zVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical zVariable records. Be careful if using C `struct` objects to store multiple full-physical zVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See Section 2.3.15 for more details on how to create this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`ATTR_SCOPE_` Respecifies the scope for the current attribute (in the current CDF). Required arguments are as follows:

in: `long scope`

New attribute scope. Specify one of the scopes described in Section 4.6.9.

The required preselected objects/states are the current CDF and its current attribute.

`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 as follows:

in: `char *attrName`

New attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

The required preselected objects/states are the current CDF and its current attribute.

`gENTRY_DATASPEC_` Modifies the data specification (data type and number of elements) of the gEntry at the current gEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in Section 2.2.9. **NOTE:** This operation should only be performed on gAttributes. Required arguments are as follows:

in: `long dataType`

New data type of the gEntry. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`rENTRY_DATASPEC_` Modifies the data specification (data type and number of elements) of the rEntry at the current rEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in Section 2.2.9. **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `long dataType`

New data type of the rEntry. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`zENTRY_DATASPEC_` Modifies the data specification (data type and number of elements) of the zEntry at the current zEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in Section 2.2.9. **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `long dataType`

New data type of the zEntry. Specify one of the data types described in Section 4.6.4.

in: `long numElements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`gENTRY_DATA_` Writes a gEntry to the current attribute at the current gEntry number (in the current CDF). An existing gEntry may be overwritten with a new gEntry having the same data specification (data type and number of elements) or a different data specification. **NOTE:** This operation should only be performed on gAttributes. Required arguments are as follows:

in: `long dataType`

Data type of the gEntry. Specify one of the data types described in Section 4.6.4.

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`.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`rENTRY_DATA_` Writes an rEntry to the current attribute at the current rEntry number (in the current CDF). An existing rEntry may be overwritten with a new rEntry having the same data specification (data type and number of elements) or a different data specification. **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `long dataType`

Data type of the rEntry. Specify one of the data types described in Section 4.6.4.

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`.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`zENTRY_DATA_` Writes a zEntry to the current attribute at the current zEntry number (in the current CDF). An existing zEntry may be overwritten with a new zEntry having the same data specification (data type and number of elements) or a different data specification. **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `long dataType`

Data type of the zEntry. Specify one of the data types described in Section 4.6.4.

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`.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`GET_` The following items may be read (gotten):

`rVARs_NUMDIMS_` Inquires the number of dimensions for the rVariables in the current CDF. Required arguments are as follows:

out: `long *numDims`

Number of dimensions.

The only required preselected object/state is the current CDF.

`rVARs_DIMSIZES_` Inquires the size of each dimension for the rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long dimSizes[]`

Dimension sizes. `dimSizes` is a 1-dimensional array containing one element per dimension. Each element of `dimSizes` receives the corresponding dimension size.

The only required preselected object/state is the current CDF.

`zVAR_NUMDIMS_` Inquires the number of dimensions for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *numDims`

Number of dimensions.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMSIZES_` Inquires the size of each dimension for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long dimSizes[]`

Dimension sizes. `dimSizes` is a 1-dimensional array containing one element per dimension. Each element of `dimSizes` receives the corresponding dimension size.

The required preselected objects/states are the current CDF and its current zVariable.

`CDF_ENCODING_` Inquires the data encoding of the current CDF. Required arguments are as follows:

out: `long *encoding`

Data encoding. The encodings are described in Section 4.6.5.

The only required preselected object/state is the current CDF.

`CDF_MAJORITY_` Inquires the variable majority of the current CDF. Required arguments are as follows:

out: `long *majority`

Variable majority. The majorities are described in Section 4.6.7.

The only required preselected object/state is the current CDF.

`CDF_FORMAT_` Inquires the format of the current CDF. Required arguments are as follows:

out: `long *format`

CDF format. The formats are described in Section 4.6.3.

The only required preselected object/state is the current CDF.

`CDF_COPYRIGHT_` Reads the copyright notice for the CDF library that created the current CDF. Required arguments are as follows:

out: `char *copyRight`

CDF copyright text. This character string must be large enough to hold `CDF_COPYRIGHT_LEN + 1` characters (including the `NUL` terminator).

The only required preselected object/state is the current CDF.

LIB_COPYRIGHT_ Reads the copyright notice of the CDF library being used. Required arguments are as follows:

out: `char *copyRight`

CDF library copyright text. This character string must be large enough hold `CDF_COPYRIGHT_LEN + 1` characters (including the `NUL` terminator).

There are no required preselected objects/states.

CDF_NUMrVARS_ Inquires the number of rVariables in the current CDF. Required arguments are as follows:

out: `long *numVars`

Number of rVariables.

The only required preselected object/state is the current CDF.

CDF_NUMzVARS_ Inquires the number of zVariables in the current CDF. Required arguments are as follows:

out: `long *numVars`

Number of zVariables.

The only required preselected object/state is the current CDF.

CDF_NUMATTRS_ Inquires the number of attributes in the current CDF. Required arguments are as follows:

out: `long *numAttrs`

Number of attributes.

The only required preselected object/state is the current CDF.

CDF_NUMgATTRS_ Inquires the number of gAttributes in the current CDF. Required arguments are as follows:

out: `long *numAttrs`

Number of gAttributes.

The only required preselected object/state is the current CDF.

CDF_NUMvATTRS_ Inquires the number of vAttributes in the current CDF. Required arguments are as follows:

out: `long *numAttrs`

Number of vAttributes.

The only required preselected object/state is the current CDF.

rVARs_MAXREC_ Inquires the maximum record number of the rVariables 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 rVariables contain no records. The maximum record number for an individual rVariable may be inquired using <GET_, rVAR_MAXREC_>. Required arguments are as follows:

out: long *maxRec

Maximum record number.

The only required preselected object/state is the current CDF.

zVARs_MAXREC_ Inquires the maximum record number of the zVariables 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 zVariables contain no records. The maximum record number for an individual zVariable may be inquired using <GET_, zVAR_MAXREC_>. Required arguments are as follows:

out: long *maxRec

Maximum record number.

The only required preselected object/state is the current CDF.

CDF_VERSION_ Inquires the version number of the CDF library that created the current CDF. Required arguments are as follows:

out: long *version

Version number.

The only required preselected object/state is the current CDF.

CDF_RELEASE_ Inquires the release number of the CDF library that created the current CDF. Required arguments are as follows:

out: long *release

Release number.

The only required preselected object/state is the current CDF.

CDF_INCREMENT_ Inquires the incremental number of the CDF library that created the current CDF. Required arguments are as follows:

out: long *increment

Incremental number.

The only required preselected object/state is the current CDF.

LIB_VERSION_ Inquires the version number of the CDF library being used. Required arguments are as follows:

out: long *version

Version number.

There are no required preselected objects/states.

`LIB_RELEASE_` Inquires the release number of the CDF library being used. Required arguments are as follows:

out: `long *release`

Release number.

There are no required preselected objects/states.

`LIB_INCREMENT_` Inquires the incremental number of the CDF library being used. Required arguments are as follows:

out: `long *increment`

Incremental number.

There are no required preselected objects/states.

`LIB_subINCREMENT_` Inquires the subincremental character of the CDF library being used. Required arguments are as follows:

out: `char *subincrement`

Subincremental character.

There are no required preselected objects/states.

`rVAR_NAME_` Inquires the name of the current rVariable (in the current CDF). Required arguments are as follows:

out: `char *varName`

Name of the rVariable. This character string must be large enough to hold `CDF_VAR_NAME_LEN + 1` characters (including the `NUL` terminator).

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATATYPE_` Inquires the data type of the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *dataType`

Data type. The data types are described in Section 4.6.4.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_NUMELEMS_` Inquires the number of elements (of the data type) for the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *numElements`

Number of elements of the data type at each value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string. (Each value consists of the entire string.) For all other data types this will always be

one (1) — multiple elements at each value are not allowed for non-character string data types.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_RECVARY_` Inquires the record variance of the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *recVary`

Record variance. The variances are described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DIMVARYS_` Inquires the dimension variances of the current rVariable (in the current CDF). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long dimVarys[]`

Dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` receives the corresponding dimension variance. The variances are described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_NUMBER_` Gets the number of the named rVariable (in the current CDF). Note that this operation does not select the current rVariable. Required arguments are as follows:

in: `char *varName`

The rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

out: `long *varNum`

The rVariable number.

The only required preselected object/state is the current CDF.

`rVAR_PADVALUE_` Inquires the pad value of the current rVariable (in the current CDF). Required arguments are as follows:

out: `void *value`

Pad value. This buffer must be large enough to hold the pad value. The pad value is read from the CDF and placed in memory at address `value`.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_EXTENDRECS_` Inquires the number of extend records for the current rVariable (in the current CDF). Extend records are described in Section 2.3.7. Required

arguments are as follows:

out: `long *NextendRecs`

Number of records to extend. A value of zero (0) indicates that the default number of records is being used.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_MAXREC_` Inquires the maximum record number for the current rVariable (in the current CDF). For rVariables 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. Required arguments are as follows:

out: `long *varMaxRec`

Maximum record number.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_MAXallocREC_` Inquires the maximum record number allocated for the current rVariable (in the current CDF). This only has significance for rVariables that are not NRV and are in single-file CDFs. Required arguments are as follows:

out: `long *varMaxRecAlloc`

Maximum record number allocated.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_nINDEXRECORDS_` Inquires the number of index records for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numRecords`

Number of index records.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_nINDEXENTRIES_` Inquires the number of index entries for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numEntries`

Number of index entries.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATA_`

Reads a value from the current rVariable (in the current CDF). The value is read at the current record number and current dimension indices for the rVariables (in the current CDF). Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

`rVAR_HYPERDATA_`

Reads one or more values from the current rVariable (in the current CDF). The values are read based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

`rVAR_SEQDATA_`

Reads one value from the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the rVariable. Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note that the current sequential value for an rVariable increments automatically as values are read.

`rVARs_RECDATA_`

Reads full-physical records from one or more rVariables (in the current CDF). The full-physical records are read at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of rVariables from which to read. This must be at least one (1).

in: `long varNums[]`

The rVariables from which to read. This array, whose size is determined by the value of `numVars`, contains rVariable numbers. The rVariable numbers can be listed in any order.

in: `void *buffer`

The buffer into which the full-physical rVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. The order of the full-physical rVariable records in this buffer will correspond to the rVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical rVariable records. Be careful if using C `struct` objects to receive multiple full-physical rVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See Section 2.3.15 for more details on how to allocate this buffer.

The required preselected objects/states are the current CDF and its current record number for rVariables.

`zVAR_NAME_` Inquires the name of the current zVariable (in the current CDF). Required arguments are as follows:

out: `char *varName`

Name of the zVariable. This character string must be large enough to hold `CDF_VAR_NAME_LEN + 1` characters (including the `NUL` terminator).

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATATYPE_` Inquires the data type of the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *dataType`

Data type. The data types are described in Section 4.6.4.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_NUMELEMS_` Inquires the number of elements (of the data type) for the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *numElements`

Number of elements of the data type at each value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`) this is the

number of characters in the string. (Each value consists of the entire string.) For all other data types this will always be one (1) — multiple elements at each value are not allowed for non-character string data types.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECVARY_` Inquires the record variance of the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *recVary`

Record variance. The variances are described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMVARYS_` Inquires the dimension variances of the current zVariable (in the current CDF). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long dimVarys[]`

Dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` receives the corresponding dimension variance. The variances are described in Section 4.6.8.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_NUMBER_` Gets the number of the named zVariable (in the current CDF). Note that this operation does not select the current zVariable. Required arguments are as follows:

in: `char *varName`

The zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

out: `long *varNum`

The zVariable number.

The only required preselected object/state is the current CDF.

`zVAR_PADVALUE_` Inquires the pad value of the current zVariable (in the current CDF). Required arguments are as follows:

out: `void *value`

Pad value. This buffer must be large enough to hold the pad value. The pad value is read from the CDF and placed in memory at address `value`.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_EXTENDRECS_` Inquires the number of extend records for the current zVariable (in the current CDF). Extend records are described in Section 2.3.7. Required arguments are as follows:

out: `long *NextendRecs`

Number of records to extend. A value of zero (0) indicates that the default number of records is being used.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_MAXREC_` Inquires the maximum record number for the current zVariable (in the current CDF). For zVariables 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. Required arguments are as follows:

out: `long *varMaxRec`

Maximum record number.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_MAXallocREC_` Inquires the maximum record number allocated for the current zVariable (in the current CDF). This only has significance for zVariables that are not NRV and are in single-file CDFs. Required arguments are as follows:

out: `long *varMaxRecAlloc`

Maximum record number allocated.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_nINDEXRECORDS_` Inquires the number of index records for the current zVariable (in the current CDF). This only has significance for zVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numRecords`

Number of index records.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_nINDEXENTRIES_` Inquires the number of index entries for the current zVariable (in the current CDF). This only has significance for zVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numEntries`

Number of index entries.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATA_` Reads a value from the current zVariable (in the current CDF). The value is read at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the current dimension indices for the zVariable.

`zVAR_HYPERDATA_` Reads one or more values from the current zVariable (in the current CDF). The values are read based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

`zVAR_SEQDATA_` Reads one value from the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the zVariable. Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are read.

`zVARs_RECDATA_` Reads full-physical records from one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is read at

the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of zVariables from which to read. This must be at least one (1).

in: `long varNums[]`

The zVariables from which to read. This array, whose size is determined by the value of `numVars`, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: `void *buffer`

The buffer into which the full-physical zVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. The order of the full-physical zVariable records in this buffer will correspond to the zVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical zVariable records. Be careful if using C `struct` objects to receive multiple full-physical zVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See Section 2.3.15 for more details on how to allocate this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`ATTR_SCOPE_` Inquires the scope of the current attribute (in the current CDF). Required arguments are as follows:

out: `long *scope`

Attribute scope. The scopes are described in Section 4.6.9.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NAME_` Inquires the name of the current attribute (in the current CDF). Required arguments are as follows:

out: `char *attrName`

Attribute name. This character string must be large enough to hold `CDF_ATTR_NAME_LEN + 1` characters (including the `NUL` terminator).

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMBER_` Gets the number of the named attribute (in the current CDF). Note that this operation does not select the current attribute. Required arguments are as follows:

in: `char *attrName`

Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

out: `long *attrNum`

The attribute number.

The only required preselected object/state is the current CDF.

`ATTR_MAXgENTRY_` Inquires the maximum gEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of gEntries for the attribute. **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `long *maxEntry`

The maximum gEntry number for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMgENTRIES_` Inquires the number of gEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum gEntry number used. **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `long *numEntries`

The number of gEntries for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_MAXrENTRY_` Inquires the maximum rEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of rEntries for the attribute. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `long *maxEntry`

The maximum rEntry number for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMrENTRIES_` Inquires the number of rEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum rEntry number used. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `long *numEntries`

The number of rEntries for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_MAXzENTRY_` Inquires the maximum zEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of zEntries for the attribute. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `long *maxEntry`

The maximum zEntry number for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMzENTRIES_` Inquires the number of zEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum zEntry number used. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `long *numEntries`

The number of zEntries for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`gENTRY_DATATYPE_` Inquires the data type of the gEntry at the current gEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `long *dataType`

Data type. The data types are described in Section 4.6.4.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`gENTRY_NUMELEMS_` Inquires the number of elements (of the data type) of the gEntry at the current gEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

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.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`gENTRY_DATA_` Reads the gEntry data value from the current attribute at the current gEntry number (in the current CDF). **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`rENTRY_DATATYPE_` Inquires the data type of the rEntry at the current rEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `long *dataType`

Data type. The data types are described in Section 4.6.4.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`rENTRY_NUMELEMS_` Inquires the number of elements (of the data type) of the rEntry at the current rEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

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.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`rENTRY_DATA_` Reads the rEntry data value from the current attribute at the current rEntry number (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`zENTRY_DATATYPE_` Inquires the data type of the zEntry at the current zEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `long *dataType`

Data type. The data types are described in Section 4.6.4.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`zENTRY_NUMELEMS_` Inquires the number of elements (of the data type) of the zEntry at the current zEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

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.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`zENTRY_DATA_` Reads the zEntry data value from the current attribute at the current zEntry number (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

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`.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`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 as follows:

out: `char *text`

Text explaining the status code. This string must be large enough to hold `CDF_STATUSTEXT_LEN + 1` characters.

The only required preselected object/state is the current status code.

### More Examples

Several more examples of the use of `CDFlib` follow. In each example it is assumed that the current CDF has already been selected (either implicitly by creating/opening the CDF or explicitly with `<SELECT_,CDF_>`).

**rVariable Creation.** In this example an rVariable will be created with a pad value being specified; initial records will be written; and the number of records to extend the rVariable when necessary will be specified.

Note that the pad value was specified before the initial records. This results in the specified pad value being written. Had the pad value not been specified first, the initial records would have been written with the default pad value. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;                     /* Status returned from CDF
                                           library. */
long        dimVarys[2];                /* Dimension variances. */
long        varNum;                     /* rVariable number. */
float       padValue = -999.9;          /* Pad value. */
.
.
dimVarys[0] = VARY;
dimVarys[1] = VARY;
status = CDFlib (CREATE_, rVAR_, "HUMIDITY", CDF_REAL4, 1, VARY,
                                 dimVarys, &varNum,
                 PUT_, rVAR_PADVALUE_, &padValue,
                       rVAR_INITIALRECS_, (long) 500,
                       rVAR_EXTENDRECS_, (long) 50,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

**zVariable Creation (Character String Data Type).** In this example a zVariable with a character string data type will be created with a pad value being specified. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;                     /* Status returned from CDF
                                           library. */
long        dimVarys[1];                /* Dimension variances. */
long        varNum;                     /* zVariable number. */
long        numDims = 1;                /* Number of dimensions. */
static long dimSizes[1] = { 20 };       /* Dimension sizes. */
long        numElems = 10;              /* Number of elements (characters
                                           in this case). */
static char padValue = "**********";    /* Pad value. */
.
.
dimVarys[0] = VARY;
```

```
status = CDFlib (CREATE_, zVAR_, "Station", CDF_CHAR, numElems, numDims,
                                 dimSizes, NOVARY, dimVarys, &varNum,
                 PUT_, zVAR_PADVALUE_, padValue,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

**Hyper Read with Subsampling.** In this example an rVariable will be subsampled in a CDF whose rVariables are 2-dimensional and have dimension sizes `[100,200]`. The CDF is row major, and the data type of the rVariable is `CDF_UINT2`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
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 rVariable record
                                        number in the CDF - this was
                                        determined with a call to
                                        CDFinquire. */
.
.
status = CDFlib (SELECT_, rVAR_NAME_, "BRIGHTNESS",
                          rVARs_RECCOUNT_, recCount,
                          rVARs_RECINTERVAL_, recInterval,
                          rVARs_DIMINDICES_, indices,
                          rVARs_DIMCOUNTS_, counts,
                          rVARs_DIMINTERVALS_, intervals,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
```

```
for (recNum = 0; recNum <= maxRec; recNum++) {
   status = CDFlib (SELECT_, rVARs_RECNUMBER_, recNum,
                    GET_, rVAR_HYPERDATA_, values,
                    NULL_);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
   /* process values */
   .
   .
}
.
.
```

**Attribute Renaming.** In this example the attribute named `Tmp` will be renamed `TMP`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;              /* Status returned from CDF library. */
.
.
status = CDFlib (SELECT_, ATTR_NAME_, "Tmp",
                 PUT_, ATTR_NAME, "TMP",
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

**Sequential Access.** In this example the values for a zVariable will be averaged. The values will be read using the sequential access method (see Section 2.3.14). Each value in each record will be read and averaged. It is assumed that the data type of the zVariable has been determined to be `CDF_REAL4`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;             /* Status returned from CDF library. */
long        varNum;             /* zVariable number. */
long        recNum = 0;         /* Record number, start at first record. */
static long indices[2] = {0,0}; /* Dimension indices. */
float       value;              /* Value read. */
double      sum = 0.0;          /* Sum of all values. */
```

```
long        count = 0;          /* Number of values. */
float       ave;                /* Average value. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "FLUX", &varNum,
                NULL_);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (SELECT_, zVAR_, varNum,
                          zVAR_SEQPOS_, recNum, indices,
                GET_, zVAR_SEQDATA_, &value,
                NULL_);
while (status >= CDF_OK) {
  sum += value;
  count++;
  status = CDFlib (GET_, zVAR_SEQDATA_, &value,
                   NULL_);
}

if (status != END_OF_VAR) UserStatusHandler (status);

ave = sum / count;
.
.
```

**Attribute rEntry Writes.** In this example a set of attribute rEntries for a particular rVariable will be written. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;                  /* Status returned from CDF
                                           library. */
static float   scale[2] = {-90.0,90.0}; /* Scale, minimum/maximum. */
.
.
status = CDFlib (SELECT_, rENTRY_NAME_, "LATITUDE",
                          ATTR_NAME_, "FIELDNAM",
                PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                    "Latitude            ",
                SELECT_, ATTR_NAME_, "SCALE",
                PUT_, rENTRY_DATA_, CDF_REAL4, (long) 2, scale,
                SELECT_, ATTR_NAME_, "UNITS",
                PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                    "Degrees north       ",
                NULL_);
if (status != CDF_OK) UserStatusHandler (status);
```

```
.
.
```

**Multiple zVariable Write.** In this example full-physical records will be written to the zVariables in a CDF. Note the ordering of the zVariables (see Section 2.3.15). It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;             /* Status returned from CDF library. */
short       time;               /* 'Time' value. */
char        vectorA[3];         /* 'vectorA' values. */
double      vectorB[5];         /* 'vectorB' values. */
long        recNumber;          /* Record number. */
char        buffer[45];         /* Buffer of full-physical records. */
long        varNumbers[3];      /* Variable numbers. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "vectorB", &varNumbers[0],
                       zVAR_NUMBER_, "time", &varNumbers[1],
                       zVAR_NUMBER_, "vectorA", &varNumbers[2],
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
for (recNumber = 0; recNumber < 100; recNumber++) {
   .
   /* read values from input file */
   .
   memmove (&buffer[0], vectorB, 40);
   memmove (&buffer[40], &time, 2);
   memmove (&buffer[42], vectorA, 3);
   status = CDFlib (SELECT_, zVARs_RECNUMBER_, recNumber,
                    PUT_, zVARs_RECDATA_, 3L, varNumbers, buffer,
                    NULL_);
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

Note that it would be more efficient to read the values directly into `buffer`. The method shown here was used to illustrate how to create the buffer of full-physical records.

### A Potential Mistake We Don't Want You to Make

The following example illustrates one of the most common mistakes made when using the Internal Interface in a C application. Please don't do something like the following:

```
.
.
#include "cdf.h"
.
.
CDFid          id;                      /* CDF identifier (handle). */
CDFstatus      status;                  /* Status returned from CDF
                                           library. */
long           varNum;                  /* zVariable number. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                 GET_, zVAR_NUMBER_, "EPOCH", &varNum,
                 SELECT_, zVAR_, varNum,                  /* ERROR! */
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

It looks like the current zVariable will be selected based on the zVariable number determined by using the `<GET_,zVAR_NUMBER_>` operation. What actually happens is that the zVariable number passed to the `<SELECT_,zVAR_>` operation is undefined. This is because the C compiler is passing `varNum` by value rather than reference.[11] Since the argument list passed to `CDFlib` is created before `CDFlib` is called, `varNum` does not yet have a value. Only after the `<GET_,zVAR_NUMBER_>` operation is performed does `varNum` have a valid value. But at that point it's too late since the argument list has already been created. Don't do this.

In this type of situation you would have to make two calls to `CDFlib`. The first would inquire the zVariable number and the second would select the current zVariable. Because the `<SELECT_,zVAR_NAME_>` operation allows you to select the current zVariable directly by name, this particular situation should never arise. It was simply used as an example. If you do encounter a situation like this that you think could be avoided with a new operation, contact CDFsupport; we're adding new operations all the time.

## 4.8 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 C explains each status code. When the status code returned from a CDF function is tested, the following rules apply.

| | |
|---|---|
| `status > CDF_OK` | Indicates successful completion but some additional information is provided. These are informational codes. |

[11] Fortran programmers can get away with doing something like this because everything is passed by reference.

| | |
|---|---|
| `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_STATUSTEXT_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.

## 4.9 CDF_EPOCH Values

Several functions exist that compute, break down, parse, and encode `CDF_EPOCH` values. These functions may be called by applications using the `CDF_EPOCH` data type and are included in the CDF library. Section 2.2.9 describes EPOCH values.

### 4.9.1 computeEPOCH

`computeEPOCH` calculates a `CDF_EPOCH` value given the individual components. If an illegal component is detected, the value returned will be `-1.0`.

```
double computeEPOCH(     /* Out -- CDF_EPOCH value returned. */
long year,               /* In  -- Year (AD, e.g., 1994). */
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). */
```

### 4.9.2 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, e.g., 1994). */
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). */
```

### 4.9.3 parseEPOCH

parseEPOCH parses a standard date/time character string and returns a CDF_EPOCH value. The format of the string is defined in Section 2.2.9 (and is also the format produced by the encodeEPOCH function). If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH(                    /* Out -- CDF_EPOCH value
                                                 returned. */
char epString[EPOCH_STRING_LEN+1]);   /* In  -- The standard date/time
                                                 character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

### 4.9.4 encodeEPOCH

encodeEPOCH encodes a CDF_EPOCH value into a standard date/time character string. The format of the string is defined in Section 2.2.9 (and is also the format expected by the parseEPOCH function).

```
void encodeEPOCH(
double epoch;                          /* In  -- the CDF_EPOCH value. */
char epString[EPOCH_STRING_LEN+1]);    /* Out -- The standard date/time
                                                  character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

### 4.9.5 encodeEPOCH1

encodeEPOCH1 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmdd.ttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and ttttttt is the fraction of the day (e.g., 5000000 is 12 o'clock noon).

```
void encodeEPOCH1(
double epoch;                          /* In  -- the CDF_EPOCH value. */
char epString[EPOCH1_STRING_LEN+1]);   /* Out -- The alternate date/time
                                                  character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

### 4.9.6 encodeEPOCH2

encodeEPOCH2 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmddhhmmss where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), and ss is the second (0-59).

```
void encodeEPOCH2(
double epoch;                              /* In  -- the CDF_EPOCH value. */
char epString[EPOCH2_STRING_LEN+1]);       /* Out -- The alternate date/time
                                                  character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

## 4.10 Custom C Functions

Most of the Standard Interface functions callable from C applications are implemented as C macros that call CDFlib (Internal Interface). For example, the CDFcreate function is actually defined as the following C macro:

```
#define CDFcreate(CDFpath,numDims,dimSizes,encoding,majority,id) \
CDFlib (CREATE_, CDF_, CDFpath, 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 (preprocessor) 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 that 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 that calls the CDF library or references CDF parameters must include one or more CDF include files. On VMS systems a logical name, `CDF$INC`, that specifies the location of the CDF include files is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_INC`, that serves the same purpose is defined in the definitions file named `definitions.<shell>` (where `<shell>` is the shell being used: `csh`, `ksh`, `sh`, or `tcsh`). This section assumes that you are using the appropriate definitions file.

On VMS and UNIX systems the following line would be included at/near the top of each routine:

```
INCLUDE '<inc-path>cdf.inc'
```

where `<inc-path>` is the pathname of the directory containing `cdf.inc`. On VMS systems `CDF$INC:` may be used for `<inc-path>`. On UNIX systems `<inc-path>` must be a relative or absolute pathname. (An environment variable may not be used.) Another option would be to create a symbolic link to `cdf.inc` (using `ln -s`) 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. On UNIX systems you will need to know where on your system `cdf.inc` has been installed.

On MS-DOS systems (assuming Microsoft Fortran is being used) the following lines would be included at/near the top of each routine/source file:

```
INCLUDE 'cdfmsf.inc'
.
. (PROGRAM, SUBROUTINE, or FUNCTION statement)
.
INCLUDE 'cdf.inc'
```

The include file `cdfmsf.inc` contains an `INTERFACE` statement for each subroutine/function in the CDF library. Including this file is absolutely essential if you are using the Internal Interface (`CDF_lib`). `cdfmsf.inc` is located in the same directory as `cdf.inc`.

**NOTE:** There are limitations on where `cdfmsf.inc` can be included. It must generally be included before the `PROGRAM`, `SUBROUTINE`, or `FUNCTION` statement of a routine. If a source file contains multiple routines, `cdfmsf.inc` only needs to be included once (at the very top of the source file). `cdf.inc`, however, may need to be included inside each routine.

### 5.2.1 VMS Systems

An example of the command to compile a source file on VMS systems would be as follows:

```
$ FORTRAN <source-name>
```

where `<source-name>` is the name of the source file being compiled. (The `.FOR` extension is not necessary.) The object module created will be named `<source-name>.OBJ`.

**NOTE:** If you are running OpenVMS on a DEC Alpha and are using a CDF distribution built for a default double-precision floating-point representation of `D_FLOAT`, you will also have to specify `/FLOAT=D_FLOAT` on the `CC` command line in order to correctly process double-precision floating-point values.

### 5.2.2 UNIX Systems

An example of the command to compile a source file on UNIX systems would be as follows:[1]

```
% f77 -c <source-name>.f
```

where `<source-file>.f` is the name of the source file being compiled. (The `.f` extension is required.)

The `-c` option specifies that only an object module is to be produced. (The link step is described in Section 4.3.) The object module created will be named `<source-name>.o`.

### 5.2.3 MS-DOS Systems

**NOTE:** Even though your application is written in Fortran and compiled with a Fortran compiler, compatible C runtime system libraries will be necessary to successfully link your application. This is because the

[1] The name of the Fortran compiler may be different depending on the flavor of UNIX being used.

CDF library is written in C and calls C runtime system functions.

An example of the command to compile a source file on MS-DOS systems would be as follows (for Microsoft Fortran):[2]

```
> FL /c /AL /FPi /I<inc-path> <source-name>.for
```

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

The `/c` option specifies that only an object module is to be produced. (The link step is described in Section 5.3.) The object module will be named `<source-name>.obj`.

The `/AL` option specifies that the object module 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. The CDF libraries supplied with the CDF distribution for MS-DOS systems are compiled using the large memory model.

The `/FPi` option specifies how floating-point operations will be handled at runtime. With this option a math coprocessor will be used if it exists; otherwise, the emulation library will be called. Using this option allows your program to run on any MS-DOS system regardless of whether or not a math coprocessor exists. If you know that a math coprocessor exists, you may want to use a floating-point option that provides better performance.

**NOTE:** The CDF library is written in C. The Fortran compiler used to compile your applications must be compatible with the C compiler used to build the CDF library. (Microsoft C and Microsoft Fortran have been shown in the examples in this document.) The linker used must also be configured to allow a Fortran application to call C routines (in the CDF library). Your Fortran applications, however, should not be concerned with calling functions written in C. (They can assume that they are calling Fortran.) The CDF library is written to handle the default Fortran calling conventions.

## 5.3 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).[3] Both the Standard and Internal interfaces for Fortran applications are built into the CDF library. On VMS systems a logical name, `CDF$LIB`, that specifies the location of the CDF library is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_LIB`, that serves the same purpose is defined in the definitions file `definitions.<shell>` (where `<shell>` is the shell being used: `csh`, `ksh`, `sh`, or `tcsh`). This section assumes that you are using the appropriate definitions file.

[2] This example assumes you have properly set the MS-DOS environment variables used by the Microsoft Fortran compiler and linker. It is also assumed that the environment variables are set such that the linker will be able to find both the Fortran and C runtime system libraries that are needed.

[3] A shareable version of the CDF library is also available on VMS and some flavors of UNIX. Its use is described in Section 5.4.

### 5.3.1 VMS Systems

An example of the command used to link an application on a VMS system would be as follows:

```
$ LINK <object-file(s)>, CDF$LIB:LIBCDF/LIBRARY
```

where `<object-file(s)>` is your application's object module(s). (The `.OBJ` extension is not necessary.) The name of the executable created will be the name part of the first object module listed with `.EXE` appended. A different executable name may be specified by using the `/EXECUTABLE` qualifier.

### 5.3.2 OpenVMS Systems

An example of the command used to link an application on an OpenVMS system would be as follows:

```
$ LINK <object-file(s)>, CDF$LIB:LIBCDF/LIBRARY, SYS$LIBRARY:<crtl>/LIBRARY
```

where `<object-file(s)>` is your application's object module(s) (the `.OBJ` extension is not necessary) and `<crtl>` is `VAXCRTL` if your CDF distribution is built for a default double-precision floating-point representation of `G_FLOAT` or `VAXCRTLD` for a default of `D_FLOAT`. (You must specify a VAX C runtime library because the CDF library is written in C.) The name of the executable created will be the name part of the first object file listed with `.EXE` appended. A different executable name may be specified by using the `/EXECUTABLE` qualifier.

### 5.3.3 UNIX Systems

An example of the command used to link an application on a UNIX system would be as follows:

```
% f77 <object-file(s)>.o ${CDF_LIB}/libcdf.a
```

where `<object-file(s)>.o` is your application's object module(s). (The `.o` extension is required.) The name of the executable created will be `a.out` by default. It may also be explicitly specified using the `-o` option. Some UNIX systems may also require that `-lc` (the C runtime library), `-lm` (the math library), and/or `-ldl` (the dynamic linker library) be specified at the end of the command line. This may depend on the particular release of the operating system being used.

#### 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 ${CDF_LIB}/libcdf.a
```

where <source-file(s)>.f is the name of the source file(s) being compiled/linked. (The .f extension is required.) Some UNIX systems may also require that -lc, -lm, and/or -ldl be specified at the end of the command line.

### 5.3.4 MS-DOS Systems

**NOTE:** Even though your application is written in Fortran and compiled with a Fortran compiler, compatible C runtime system libraries will be necessary to successfully link your application. This is because the CDF library is written in C and calls C runtime system functions.

An example of the command used to link an application on MS-DOS systems would be as follows (for Microsoft Fortran):

```
> LINK /NOE <object-file(s)>,<executable>,nul.map,<lib-path>libcdf.lib;
```

where <object-file(s)> is your application's object module(s) (the .obj extension is not necessary); <executable> is the name of the executable file to be created (.exe will be appended); and <lib-path> is the pathname of the directory containing libcdf.lib. You will need to know where on your system libcdf.lib has been installed.

A map file is created by default unless the special name nul.map is used (as shown). If a map file is desired, the map file parameter should be omitted (in which case the name of the map file will be the name part of the executable file with .map appended), or a map file should be explicitly specified.

The /NOE option specifies that the linker should not search extended dictionaries of library symbols. This is necessary to suppress errors that would be generated because of multiply defined symbols between the Microsoft Fortran and Microsoft C system libraries.

**NOTE:** The same memory model must have been used to compile your application's source files and the CDF library. The CDF library can be compiled using either the large or huge memory model. The CDF libraries supplied with the CDF distribution for MS-DOS systems are compiled using the large memory model.

## 5.4 Linking, Shared CDF Library

A shareable version of the CDF library is also available on VMS systems and some flavors of UNIX. The shared version is put in the same directory as the non-shared version and is named as follows:

| Machine/Operating System | Shared CDF Library |
|---|---|
| VAX/VMS & DEC Alpha/OpenVMS | LIBCDF.EXE |
| Sun/SunOS & Sun/SOLARIS | libcdf.so |
| HP 9000/HP-UX | libcdf.sl |
| IBM RS6000/AIX | libcdf.o |

The commands necessary to link to a shareable library vary among operating systems. Examples are shown in the following sections.

### 5.4.1 VAX/VMS

```
$ ASSIGN CDF$LIB:LIBCDF.EXE CDF$LIBCDFEXE
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
  CDF$LIBCDFEXE/SHAREABLE
  SYS$SHARE:VAXCRTL/SHAREABLE
  <Control-Z>
$ DEASSIGN CDF$LIBCDFEXE
```

where `<object-file(s)>` is your application's object module(s). (The `.OBJ` extension is not necessary.) The name of the executable created will be the name part of the first object file listed with `.EXE` appended. A different executable name may be specified by using the `/EXECUTABLE` qualifier.

**NOTE:** On VMS systems the shared CDF library may also be installed in `SYS$SHARE`.

### 5.4.2 DEC Alpha/OpenVMS

```
$ ASSIGN CDF$LIB:LIBCDF.EXE CDF$LIBCDFEXE
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
  CDF$LIBCDFEXE/SHAREABLE
  SYS$LIBRARY:<crtl>/LIBRARY
  <Control-Z>
$ DEASSIGN CDF$LIBCDFEXE
```

where `<object-file(s)>` is your application's object module(s) (the `.OBJ` extension is not necessary) and `<crtl>` is `VAXCRTL` if your CDF distribution is built for a default double-precision floating-point representation of `G_FLOAT` or `VAXCRTLD` for a default of `D_FLOAT`. (You must specify a VAX C runtime library because the CDF library is written in C.) The name of the executable created will be the name part of the first object file listed with `.EXE` appended. A different executable name may be specified by using the `/EXECUTABLE` qualifier.

**NOTE:** On OpenVMS systems the shareable CDF library may also be installed in `SYS$SHARE`.

### 5.4.3 Sun/SunOS

```
% f77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lm -ldl
```

where `<object-file(s)>.o` is your application's object module(s) (the `.o` extension is required), and `<exe-file>` is the name of the executable file created.

### 5.4.4 Sun/SOLARIS

```
% f77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lc -lm
```

where `<object-file(s)>.o` is your application's object module(s) (the `.o` extension is required), and `<exe-file>` is the name of the executable file created.

### 5.4.5 HP 9000/HP-UX

```
% f77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.sl
```

where `<object-file(s)>.o` is your application's object module(s) (the `.o` extension is required), and `<exe-file>` is the name of the executable file created.

### 5.4.6 IBM RS6000/AIX

```
% f77 -o <exe-file> <object-file(s)>.o -L${CDF_LIB} ${CDF_LIB}/libcdf.o -lc -lm
```

where `<object-file(s)>.o` is your application's object module(s) (the `.o` extension is required), and `<exe-file>` is the name of the executable file created.

## 5.5 Argument Passing

The CDF library is written entirely in C. Most computer systems have Fortran and C compilers that allow a Fortran application to call a C function without being concerned that different programming languages are involved. The CDF library takes advantage of the mechanisms provided by these compilers so that your Fortran application can appear to be calling another Fortran subroutine/function (in actuality the CDF library written in C). Pass all arguments exactly as shown in the description of each CDF function. This includes character strings (i.e., `%REF(...)` is not required). Be aware, however, that trailing blanks on variable and attribute names will be considered as part of the name. If the trailing blanks are not desired, pass only the part of the character string containing the name (e.g., `VAR_NAME(1:8)`).

**NOTE:** Unfortunately, the world is not a perfect place. The Microsoft C and Microsoft Fortran compilers on the IBM PC and the C and Fortran compilers on the NeXT computer do not provide the needed mechanism to pass character strings from Fortran to C without explicitly `NUL` terminating the strings. Your Fortran application must place an ASCII `NUL` character after the last character of a CDF, variable, or attribute name. An example of this follows:

```
.
.
CHARACTER ATTR_NAME*9                ! Attribute name
.
.
ATTR_NAME(1:8) = 'VALIDMIN'          ! Actual attribute name
ATTR_NAME(9:9) = CHAR(0)             ! ASCII NUL character
.
.
```

`CHAR(0)` is an intrinsic Fortran function that returns the ASCII character for the numerical value passed in (0 is the numerical value for an ASCII `NUL` character). `ATTR_NAME` could then be passed to one of the CDF library functions.

When the CDF library passes out a character string on an IBM PC (using the Microsoft compilers) or on a NeXT computer, the number of characters written will be exactly as shown in the description of the function called. You must declare your Fortran variable to be exactly that size.

## 5.6 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. Note that both rVariables and zVariables are numbered starting at one (1).

## 5.7 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`. The file `cdf.inc` should be `INCLUDE`ed in all application source files referencing CDF functions/parameters.

### 5.7.1 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 5.9 describes how to use these constants to interpret status codes.

### 5.7.2 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. |

### 5.7.3 CDF Data Types

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).

### 5.7.4 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 passed into the CDF library (to be written to a CDF) should always be in the host machine's native encoding. Attribute entry and variable values read from a CDF by the CDF library and passed out to an application will be in the currently selected decoding for that CDF (see Section 2.2.8).

| | |
|---|---|
| HOST_ENCODING | Indicates host machine data representation (native). This encoding will provide the greatest performance when reading/writing on a machine of the same type. |
| NETWORK_ENCODING | Indicates network transportable data representation (XDR). |
| VAX_ENCODING | Indicates VAX data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation. |
| ALPHAVMSd_ENCODING | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation. |
| ALPHAVMSg_ENCODING | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's G_FLOAT representation. |
| SUN_ENCODING | Indicates SUN data representation. |
| SGi_ENCODING | Indicates Silicon Graphics Iris and Power Series data representation. |

| | |
|---|---|
| DECSTATION_ENCODING | Indicates DECstation data representation. |
| ALPHAOSF1_ENCODING | Indicates DEC Alpha running OSF/1 data representation. |
| IBMRS_ENCODING | Indicates IBMRS data representation (IBM RS6000 series). |
| HP_ENCODING | Indicates HP data representation (HP 9000 series). |
| IBMPC_ENCODING | Indicates IBM PC data representation. |
| NeXT_ENCODING | Indicates NeXT data representation. |

When creating a CDF (via the Standard Interface) or respecifying a CDF's encoding (via the Internal Interface), you must specify NETWORK_ENCODING, HOST_ENCODING, or the specific encoding of the host machine (e.g., 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.)

### 5.7.5 Data Decodings

A CDF's decoding affects how its attribute entry and variable data values are passed out to a calling application. The decoding for a CDF may be selected and reselected any number of times while the CDF is open. Selecting a decoding does not affect how the values are stored in the CDF file(s) — only how the values are decoded by the CDF library. Any decoding may be used with any of the supported encodings. Section 2.2.8 describes a CDF's decoding in more detail.

| | |
|---|---|
| HOST_DECODING | Indicates host machine data representation (native). This is the default decoding. |
| NETWORK_DECODING | Indicates network transportable data representation (XDR). |
| VAX_DECODING | Indicates VAX data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation. |
| ALPHAVMSd_DECODING | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation. |
| ALPHAVMSg_DECODING | Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's G_FLOAT representation. |
| SUN_DECODING | Indicates SUN data representation. |
| SGi_DECODING | Indicates Silicon Graphics Iris and Power Series data representation. |
| DECSTATION_DECODING | Indicates DECstation data representation. |
| ALPHAOSF1_DECODING | Indicates DEC Alpha running OSF/1 data representation. |
| IBMRS_DECODING | Indicates IBMRS data representation (IBM RS6000 series). |
| HP_DECODING | Indicates HP data representation (HP 9000 series). |
| IBMPC_DECODING | Indicates IBM PC data representation. |

| | |
|---|---|
| NeXT_DECODING | Indicates NeXT data representation. |

The default decoding is HOST_DECODING. The other decodings may be selected via the Internal Interface with the <SELECT_,CDF_DECODING_> operation. Section 2.2.8 describes those situations in which a decoding other than HOST_DECODING may be desired.

### 5.7.6 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). The majority is the same for rVariables and zVariables.

| | |
|---|---|
| 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 multidimensional arrays varies the fastest in memory.

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

The majority must also be considered when performing sequential reads and writes. When sequentially reading a variable, the values passed out by the CDF library will be ordered according to the majority. When sequentially writing a variable, the values passed into the CDF library will be assumed (by the CDF library) to be ordered according to the majority.

### 5.7.7 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 of the other records are virtual and contain the same values.)

If a variable has a dimension variance of VARY, then each value/subarray along that dimension is physically stored. If the dimension variance is NOVARY, then only one value/subarray along that dimension is physically stored. (All other values/subarrays along that dimension are virtual and contain the same values.)

### 5.7.8 Attribute Scopes

Attribute scopes are simply a way to explicitly declare the intended use of an attribute by user applications (and the CDF toolkit).

| | |
|---|---|
| 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 rEntry or zEntry corresponds to an rVariable or zVariable, respectively.) |
| 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 2.4.3 explains attribute scopes and how assumed scopes can arise in a CDF.

### 5.7.9 Read-Only Modes

Once a CDF has been opened, it may be placed into a read-only mode to prevent accidental modification (such as when the CDF is simply being browsed). Read-only mode is selected via the Internal Interface using the <SELECT_,CDF_READONLY_MODE_> operation.

| | |
|---|---|
| READONLYon | Turns on read-only mode. |
| READONLYoff | Turns off read-only mode. |

### 5.7.10 zModes

Once a CDF has been opened, it may be placed into one of two variations of zMode. zMode is fully explained in Section 2.1.2. A zMode is selected for a CDF via the Internal Interface using the <SELECT_,CDF_zMODE_> operation.

| | |
|---|---|
| zMODEoff | Turns off zMode. |
| zMODEon1 | Turns on zMode/1. |
| zMODEon2 | Turns on zMode/2. |

### 5.7.11 -0.0 to 0.0 Modes

Once a CDF has been opened, the CDF library may be told to convert -0.0 to 0.0 when read from that CDF. This mode is selected via the Internal Interface using the <SELECT_,CDF_NEGtoPOSfp0_MODE_> operation.

| | |
|---|---|
| NEGtoPOSfp0on | Convert -0.0 to 0.0 when read from the CDF. |
| NEGtoPOSfp0off | Do not convert -0.0 to 0.0 when read from the CDF. |

### 5.7.12 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 for the rVariables or a zVariable. |

The CDF library imposes no limit on the number of variables, attributes, or attribute entries that a CDF may have. On the IBM PC, however, the number of rVariable and zVariables will be limited to 100 of each in a multi-file CDF because of the 8.3 naming convention imposed by MS-DOS.

### 5.7.13 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 file names). 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 CDF copyright text. |
| CDF_STATUSTEXT_LEN | Maximum length of the explanation text for a status code. |

## 5.8 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 5.9). 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. Note that zVariables and vAttribute zEntries are only accessible via the Internal Interface.

### 5.8.1 CDF_create (Standard Interface)

```
SUBROUTINE CDF_create (CDF_path, num_dims, dim_sizes, encoding, majority,
1                      id, status)

CHARACTER CDF_path*(*)        ! in  -- CDF pathname.
```

```
INTEGER*4 num_dims            ! in  -- Number of dimensions, rVariables.
INTEGER*4 dim_sizes(*)        ! in  -- Dimension sizes, rVariables.
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 overwrite 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 and .z0,.z1,... files.

The arguments to CDF_create are defined as follows:

| | |
|---|---|
| CDF_path | 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).<br><br>**NOTE:** On UNIX systems pathnames are case-sensitive. |
| num_dims | Number of dimensions the rVariables in the CDF are 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). If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| encoding | The encoding for variable data and attribute entry data. Specify one of the encodings described in Section 5.7.4. |
| majority | The majority for variable data. Specify one of the majorities described in Section 5.7.6. |
| 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 5.9 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 specified in the configuration file of your CDF distribution. Consult your system manager for this default. The CDF_lib function (Internal Interface) may be used to change a CDF's format to single-file.

**NOTE:** 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 5.8.5).

**Example(s)**

The following example will create a 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, rVariables.
INTEGER*4 dim_sizes(3)    ! Dimension sizes, rVariables.
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`.

### 5.8.2 CDF_open (Standard Interface)

```
SUBROUTINE CDF_open (CDF_path, id, status)

CHARACTER CDF_path*(*)        ! 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_path` 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 5.9 explains how to interpret status codes. |

**NOTE:** 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 5.8.5).

**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_path*(CDF_PATHNAME_LEN)   ! Pathname of CDF.

DATA CDF_path/'NOAA1'/
.
.
CALL CDF_open (CDF_path, id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 5.8.3 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 that which created the CDF is provided (e.g., CDF V2.3 is version 2, release 3) 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 that created the CDF. |
| release | The release number of the CDF library that created the CDF. |
| copyright | The copyright notice of the CDF library that created the CDF. This character string must be large enough to hold CDF_COPYRIGHT_LEN characters and will be blank padded if necessary. 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 5.9 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.
    INTEGER*4 last_char                       ! Last character position
                                              ! actually used in the copyright.
    INTEGER*4 start_char                      ! Starting character position
                                              ! in a line of the copyright.
    CHARACTER lf*1                            ! Linefeed character.
    .
    .
    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)

      last_char = CDF_COPYRIGHT_LEN
      DO WHILE (copyright(last_char:last_char) .EQ. ' ')
        last_char = last_char - 1
      END DO

      lf = CHAR(10)

      start_char = 1
      DO i = 1, last_char
        IF (copyright(i:i) .EQ. lf) THEN
```

```
          WRITE (6,301) copyright(start_char:i-1)
 301      FORMAT (' ',A)
          start_char = i + 1
        END IF
      END DO
    END IF
    .
    .
```

### 5.8.4 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, rVariables.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)    ! out -- Dimension sizes, rVariables.
INTEGER*4 encoding                   ! out -- Data encoding.
INTEGER*4 majority                   ! out -- Variable majority.
INTEGER*4 max_rec                    ! out -- Maximum record number in the CDF,
                                              rVariables.
INTEGER*4 num_vars                   ! out -- Number of rVariables 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 rVariable dimensions and their sizes before it can access rVariable data. Knowing the variable majority can be used to optimize performance and is necessary to properly use the variable hyper functions (for both rVariables and zVariables).

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 for the rVariables. |
| `dim_sizes` | The dimension sizes for the rVariables. `dim_sizes` is a 1-dimensional array containing one element per dimension. Each element of `dim_sizes` receives the corresponding dimension size. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary). |
| `encoding` | The encoding of the variable data and attribute entry data. The encodings are defined in Section 5.7.4. |
| `majority` | The majority of the variable data. The majorities are defined in Section 5.7.6. |
| `max_rec` | The maximum record number written to an rVariable in the CDF. Note that the maximum record number written is also kept separately for each rVariable |

in the CDF. The value of max_rec is the largest of these. Some rVariables may have fewer records actually written. CDF_lib (Internal Interface) may be used to inquire the maximum record written for an individual rVariable (see Section 5.8.25).

| | |
|---|---|
| num_vars | The number of rVariables in the CDF. |
| num_attrs | The number of attributes in the CDF. |
| status | The completion status code. Section 5.9 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, rVariables.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)   ! Dimension sizes, rVariables
                                    ! (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.
INTEGER*4 num_vars                  ! Number of rVariables 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)
.
.
```

### 5.8.5 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 cache buffers are flushed; the CDF's open file is closed (or files in the case of a multi-file CDF); and the CDF identifier is made available for reuse.

**NOTE:** 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 cache buffers are left unflushed.

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 5.9 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)
.
.
```

### 5.8.6 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 and `.z0`,`.z1`,... 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 5.9 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
.
.
```

### 5.8.7 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 5.9 explains how to interpret status codes, and Appendix C 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 and will be blank padded 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.
      INTEGER*4 last_char                   ! Last character position
                                            ! actually used in the copyright.
      .
      .
      CALL CDF_open ('giss_wetl', id, status)
      IF (status .LT. CDF_WARN) THEN         ! INFO and WARNING codes ignored.
        CALL CDF_error (status, text)
        last_char = CDF_STATUSTEXT_LEN
        DO WHILE (text(last_char:last_char) .EQ. ' ')
          last_char = last_char - 1
        END DO
        WRITE (6,101) text(1:last_char)
 101    FORMAT (' ','ERROR> ',A)
      END IF
      .
      .
```

### 5.8.8 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 5.7.8. |
| `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 5.9 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)
.
.
```

### 5.8.9 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 (e.g., 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.)

**NOTE:** Some Fortran compilers give a warning if CDF_attr_num is not used in a routine where cdf.inc has been included. This warning may be ignored. To eliminate the warning message, remove the declaration of CDF_attr_num from cdf.inc, but be sure to declare CDF_attr_num in all routines that use it.

**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 5.8.10.

### 5.8.10 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 5.8.9). |
| 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 5.9 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)
.
.
```

### 5.8.11 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 (see Section 5.8.12).

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 5.8.9). |
| attr_name | The attribute's name. This character string must be large enough to hold CDF_ATTR_NAME_LEN characters and will be blank padded if necessary. |
| attr_scope | The scope of the attribute. Attribute scopes are defined in Section 5.7.8. |
| max_entry | For gAttributes this is the maximum entry number used. For vAttributes this is the maximum rEntry number used. In either case 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 (see Section 5.8.25). |
| status | The completion status code. Section 5.9 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
   .
   .
```

### 5.8.12 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 -- 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 (see Section 5.8.11). 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 for which to inquire an entry. This number may be determined with a call to CDF_attr_num (see Section 5.8.9). |
| 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 rVariable (the rVariable being described in some way by the entry). |
| data_type | The data type of the specified entry. The data types are defined in Section 5.7.3. |
| 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 5.9 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 rEntry numbers are actually rVariable 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 used.
   INTEGER*4 data_type                       ! 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
```

### 5.8.13 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 the 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 5.8.9). |
| 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 rVariable (the rVariable being described in some way by the rEntry). |
| data_type | The data type of the entry. Specify one of the data types defined in Section 5.7.3. |
| 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 value is written to the CDF from memory address value. |
| status | The completion status code. Section 5.9 explains how to interpret status codes. |

#### Example(s)

The following example writes two attribute entries. The first is to gEntry number one (1) of the gAttribute TITLE. The second is to vAttribute VALIDs for the rEntry that corresponds to the rVariable 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, rEntry
                                   ! number 1.
INTEGER*2 TMPvalids(2)             ! Value(s) of VALIDs attribute,
                                   ! rEntry for rVariable TMP

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_num(id,'TMP'),
1                  CDF_INT2, num_elements, TMPvalids, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 5.8.14 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 5.8.9).

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 rVariable (the rVariable being described in some way by the rEntry).

value — The value read. This buffer must be large enough to hold the value. The subroutine CDF_attr_entry_inquire would be used to determine the entry data type and number of elements (of that data type). The 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 rEntry corresponding to the PRES_LVL rVariable (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 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')              ! The rEntry number is
                                                   ! the rVariable number.
IF (entryN .LT. 0) CALL UserStatusHandler (entryN) ! 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
   .
   .
```

### 5.8.15 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 2.4.3 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 5.8.9). |
| `attr_scope` | The new scope for the attribute. Specify one of the definite scopes defined in Section 5.7.8. |
| `status` | The completion status code. Section 5.9 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 2.4.3). 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
.
.
```

### 5.8.16 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 --  rVariable 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 -- rVariable number.
INTEGER*4 status             ! out -- Completion status.
```

CDF_var_create is used to create a new rVariable in a CDF. A variable (rVariable or zVariable) 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 rVariable 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 rVariable. Specify one of the data types defined in Section 5.7.3. |
| num_elements | The number of elements of the data type at each value. For character string data types (CDF_CHAR and CDF_UCHAR) this is the number of characters in the string. (Each value consists of the entire string.) For all other data types this must always be one (1) — multiple elements at each 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 5.7.7. |
| dim_variances | The variable's dimension variances. dim_variances is a 1-dimensional array containing one element per rVariable dimension. Each element of dim_variances specifies the corresponding dimension variance. For each dimension specify one of the variances defined in Section 5.7.7. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary). |
| var_num | The number assigned to the new rVariable. This number must be used in subsequent CDF function calls when referring to this rVariable. An existing rVariable's number may be determined with the CDF_var_num function. |
| status | The completion status code. Section 5.9 explains how to interpret status codes. |

### Example(s)

The following example will create several rVariables in a CDF. In this case EPOCH, LAT, and LON are independent rVariables and TMP is a dependent rVariable.

```
.
.
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 rVariable number.
INTEGER*4 LON_var_num               ! LON rVariable number.
INTEGER*4 TMP_var_num               ! TMP rVariable 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)
.
.
```

### 5.8.17 CDF_var_num (Standard Interface)

```
INTEGER*4 FUNCTION CDF_var_num (id, var_name)

INTEGER*4 id                ! in -- CDF identifier.
CHARACTER var_name*(*)      ! in -- rVariable name.
```

CDF_var_num is used to determine the number associated with a given rVariable name. If the rVariable is found, CDF_var_num returns its number — which will be equal to or greater than one (1). If an error occurs (e.g., the rVariable 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 rVariable 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 an rVariable number is needed. CDF_var_num is declared in cdf.inc. (Fortran functions must be declared so that the returned value is interpreted correctly.)

**NOTE:** Some Fortran compilers give a warning if CDF_var_num is not used in a routine where cdf.inc has been included. This warning may be ignored. To eliminate the warning message, remove the declaration of CDF_var_num from cdf.inc, but be sure to declare CDF_var_num in all routines that use it.

### Example(s)

In the following example CDF_var_num is used as an embedded function call when inquiring about an rVariable.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                            ! CDF identifier.
INTEGER*4 status                        ! Returned status code.
CHARACTER var_name*(CDF_VAR_NAME_LEN)   ! rVariable name.
INTEGER*4 data_type                     ! Data type of the rVariable.
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 rVariable named LATITUDE was inquired. Note that if 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 an rVariable 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 rVariable names

will be unknown — CDF_var_inquire would be used to determine them. CDF_var_inquire is described in Section 5.8.19.

### 5.8.18 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  -- rVariable number.
CHARACTER var_name*(*)     ! in  -- New name.
INTEGER*4 status           ! out -- Completion status.
```

CDF_var_rename is used to rename an existing rVariable. A variable (rVariable or zVariable) with the same 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 rVariable to rename. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| var_name | The new rVariable name. This may be at most CDF_VAR_NAME_LEN characters. Variables names are case-sensitive. |
| status | The completion status code. Section 5.9 explains how to interpret status codes. |

**Example(s)**

In the following example the rVariable named TEMPERATURE is renamed TMP (if it exists). Note that if CDF_var_num returns a value less than one (1), then that value is not an rVariable 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                   ! rVariable 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
.
.
```

### 5.8.19 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  -- rVariable number.
CHARACTER var_name*(CDF_VAR_NAME_LEN)   ! out -- rVariable 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 rVariable. This function would normally be used before reading rVariable 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 rVariable to inquire. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| var_name | The rVariable's name. This character string must be large enough to hold CDF_VAR_NAME_LEN characters and will be blank padded if necessary. |
| data_type | The data type of the rVariable. The data types are defined in Section 5.7.3. |
| num_elements | The number of elements of the data type at each value. For character string data types (CDF_CHAR and CDF_UCHAR) this is the number of characters in the string. (Each value consists of the entire string.) For all other data types this will always be one (1) — multiple elements at each value are not allowed for non-character string data types. |
| rec_variance | The record variance. The record variances are defined in Section 5.7.7. |
| dim_variances | The dimension variances. dim_variances is a 1-dimensional array containing one element per rVariable dimension. Each element of dim_variances receives the corresponding dimension variance. The dimension variances are defined in |

Section 5.7.7. For 0-dimensional rVariable this argument is ignored (but a place holder is necessary).

`status` The completion status code. Section 5.9 explains how to interpret status codes.

**Example(s)**

The following example inquires about an rVariable named `HEAT_FLUX` in a CDF. Note that the rVariable 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)   ! rVariable 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)
.
.
```

### 5.8.20 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 -- rVariable 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 rVariable).
INTEGER*4 status         ! out -- Completion status.
```

`CDF_var_put` is used to write a single value to an rVariable. `CDF_var_hyper_put` may be used to write more than one rVariable 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 rVariable to which to write. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| 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 rVariable dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariables 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 5.9 explains how to interpret status codes. |

**Example(s)**

The following example writes values to the rVariable LATITUDE of a CDF whose rVariables are 2-dimensional and have 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                     ! rVariable 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 an rVariable
                                                  ! 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 are "virtually" the same at each index of the first dimension.)

### 5.8.21 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  -- rVariable 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 rVariable).
INTEGER*4 status         ! out -- Completion status.
```

CDF_var_get is used to read a single value from an rVariable. CDF_var_hyper_get may be used to read more than one rVariable 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 rVariable from which to read. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| 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 rVariable dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary). |
| value | The value read. This buffer must be large enough to hold the value. CDF_var_inquire would be used to determine the rVariable's data type and number of elements (of that data type) at each value. The value is read from the CDF and placed into memory at address value. |
| status | The completion status code. Section 5.9 explains how to interpret status codes. |

#### Example(s)

The following example will read and hold an entire record of data from an rVariable. The CDF's rVariables are 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                     ! rVariable 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
.
.
```

### 5.8.22 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 -- rVariable 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 rVariable.)
INTEGER*4 status          ! out -- Completion status.
```

CDF_var_hyper_put is used to write a buffer of one or more values to an rVariable. 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_inquire is used to determine the variable majority of a CDF. Section 2.3.11 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 rVariable to which to write. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| 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[4] (e.g., 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 rVariable dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariables 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 rVariable dimension. Each element of counts specifies the corresponding dimension count. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary). |
| intervals | For each dimension the interval between values for subsampling[5] (e.g., an interval of 2 means write to every other value). intervals is a 1-dimensional array containing one element per rVariable dimension. Each element of intervals specifies the corresponding dimension interval. For 0-dimensional rVariables 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 5.9 explains how to interpret status codes. |

### Example(s)

The following example writes values to the rVariable LATITUDE in a CDF whose rVariables are 2-dimensional and have dimension sizes [360,181]. For LATITUDE the record variance is NOVARY; the dimension variances

[4] "Subsampling" is not the best term to use when writing data, but you should know what we mean.
[5] Again, not the best term.

are [NOVARY,VARY]; and the data type is CDF_INT2. This example is similar to the example in Section 5.8.20 except that it 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                     ! rVariable 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 an rVariable
                                                  ! number but rather a
                                                  ! warning/error code
DO lat = -90, 90
  lats(91+lat) = lat
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)
.
.
```

### 5.8.23 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 -- rVariable 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 rVariable).
INTEGER*4 status            ! out -- Completion status.
```

CDF_var_hyper_get is used to read a buffer of one or more values from an rVariable. 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 2.3.11 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 rVariable from which to read. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| 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 (e.g., 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 rVariable dimension. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariables 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 rVariable dimension. Each element of counts specifies the corresponding dimension count. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary). |
| intervals | For each dimension the interval between values for subsampling (e.g., an interval of 2 means read every other value). intervals is a 1-dimensional array containing one element per rVariable dimension. Each element of intervals specifies the corresponding dimension interval. For 0-dimensional rVariables 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 enough 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 5.9 explains how to interpret status codes. |

**Example(s)**

The following example will read an entire record of data from an rVariable. The CDF's rVariables are 3-dimensional with sizes `[180,91,10]`, and 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 5.8.21 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                     ! rVariable 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)` for proper indexing.

### 5.8.24 CDF_var_close (Standard Interface)

```
SUBROUTINE CDF_var_close (id, var_num, status)

INTEGER*4 id                ! in -- CDF identifier.
INTEGER*4 var_num           ! in -- rVariable number.
```

```
INTEGER*4 status            ! out -- Completion status.
```

CDF_var_close is used to close an rVariable in a multi-file CDF. This function is not applicable to single-file CDFs. The use of CDF_var_close is not required since the CDF library automatically closes the rVariable files when a multi-file CDF is closed or when there are insufficient file pointers available (because of an open file quota) to keep all of the rVariable files open. CDF_var_close would be used by an application since it knows best how its rVariables are going to be accessed. Closing an rVariable would also free the cache buffers that are associated with the rVariable's file. This could be important in those situations where memory is limited (e.g., the IBM PC). The caching scheme used by the CDF library is described in Section 2.1.4. Note that there is not a function that opens an rVariable. The CDF library automatically opens an rVariable when it is accessed by an application (unless it is already open).

The arguments to CDF_var_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. |
| var_num | The number of the rVariable to close. This number may be determined with a call to CDF_var_num (see Section 5.8.17). |
| status | The completion status code. Section 5.9 explains how to interpret status codes. |

**Example(s)**

The following example will close an rVariable in a multi-file CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
.
.
CALL CDF_var_close (id, CDF_var_num(id,'Flux'), status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 5.8.25 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 (e.g., specifying a single-file format for a CDF, accessing zVariables, or specifying a pad value for an rVariable or zVariable). 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 that specify one or more operations to be performed (e.g., opening a CDF, creating an attribute, or writing a variable value). 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 (e.g., a CDF, variable, or attribute) or a state (e.g., a CDF's format, a variable's data specification, or a CDF's current attribute). 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 (assuming multi-file is the default):

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
CHARACTER CDF_path*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_path/'test1'/, num_dims/2/, dim_sizes/100,200/,
1    encoding/HOST_ENCODING/, majority/ROW_MAJOR/, format/SINGLE_FILE/
.
.
CALL CDF_create (CDF_path, num_dims, dim_sizes, encoding, majority,
1                id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_lib (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 with a format of multi-file. 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:

| | |
|---|---|
| `PUT_` | The first operation to be performed. In this case an item is going to be put to the "current" CDF (a new format). `PUT_` is defined in `cdf.inc` (as are all CDF constants). It was not necessary to select a current CDF since the call to `CDF_create` implicitly selected the CDF created as the current CDF.[6] This is the case since all of the Standard Interface functions actually call the Internal Interface to perform their operations. |
| `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. |
| `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 5.9 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_path, 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. |

[6] In previous releases of CDF it was required that the current CDF be selected in each call to `CDF_lib`. That requirement has been eliminated. The CDF library now maintains the current CDF from one call to the next of `CDF_lib`.

| | |
|---|---|
| CDF_ | The item to be created — a CDF in this case. There are four required argument that must follow. When a CDF is created (with CDF_lib), the format, encoding, and majority default to values specified when your CDF distribution was built and installed. Consult your system manager for these defaults. |
| CDF_path | The pathname 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 into the CDF. |
| 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 implicitly 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 5.9 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 current objects/states are used by the CDF library when performing operations:

CDF (object)

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_>[7] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.[8]

rVariable (object)

An rVariable operation is always performed on the current rVariable in the current CDF. For each open CDF a current rVariable is maintained. This current rVariable is implicitly selected when an rVariable is created (in the current CDF) or it may be explicitly selected with the <SELECT_,rVAR_> or <SELECT_,rVAR_NAME_> operations. There is no current rVariable in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

zVariable (object)

A zVariable operation is always performed on the current zVariable in the current CDF. For each open CDF a current zVariable is maintained. This current zVariable is implicitly selected when a zVariable is created (in the current CDF) or it may be explicitly selected with the <SELECT_,zVAR_> or <SELECT_,zVAR_NAME_> operations. There is no current zVariable in a CDF until one is created (which implicitly selects it) or until one is explicitly selected.

attribute (object)

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.

gEntry number (state)

A gAttribute gEntry operation is always performed on the current gEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current gEntry number is maintained. This current gEntry number must be explicitly selected with the <SELECT_,gENTRY_> operation. (There is no implicit or default selection of the current gEntry number for a CDF.) Note that the current gEntry number is maintained for the CDF (not each attribute) — it applies to all of the attributes in that CDF.

rEntry number (state)

A vAttribute rEntry operation is always performed on the current rEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current rEntry number is maintained. This current rEntry number must be explicitly selected with the <SELECT_,rENTRY_> operation. (There is no implicit or default selection of the current rEntry number for a CDF.) Note that the current rEntry number is maintained for the CDF (not each attribute) — it applies to all of the attributes in that CDF.

zEntry number (state)

A vAttribute zEntry operation is always performed on the current zEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current zEntry number is maintained. This current zEntry number must be explicitly selected with the <SELECT_,zENTRY_> operation. (There is no implicit or default selection of the current zEntry number for a CDF.) Note that the current zEntry number is maintained for the CDF (not each attribute) — it applies to all of the attributes in that CDF.

[7] This notation is used to specify an operation to be performed on an item. The syntax is <operation,item>.

[8] In previous releases of CDF it was required that the current CDF be selected in each call to CDF_lib. That requirement no longer exists. The CDF library now maintains the current CDF from one call to the next of CDF_lib.

record number, rVariables (state)

An rVariable read or write operation is always performed at (for single and multiple variable reads and writes) or starting at (for hyper reads and writes) the current record number for the rVariables in the current CDF. When a CDF is opened or created, the current record number for its rVariables is initialized to zero (0). It may then be explicitly selected using the <SELECT_,rVARs_RECNUMBER_> operation. Note that the current record number for rVariables is maintained for a CDF (not each rVariable) — it applies to all of the rVariables in that CDF.

record count, rVariables (state)

An rVariable hyper read or write operation is always performed using the current record count for the rVariables in the current CDF. When a CDF is opened or created, the current record count for its rVariables is initialized to one (1). It may then be explicitly selected using the <SELECT_,rVARs_RECCOUNT_> operation. Note that the current record count for rVariables is maintained for a CDF (not each rVariable) — it applies to all of the rVariables in that CDF.

record interval, rVariables (state)

An rVariable hyper read or write operation is always performed using the current record interval for the rVariables in the current CDF. When a CDF is opened or created, the current record interval for its rVariables is initialized to one (1). It may then be explicitly selected using the <SELECT_,rVARs_RECINTERVAL_> operation. Note that the current record interval for rVariables is maintained for a CDF (not each rVariable) — it applies to all of the rVariables in that CDF.

dimension indices, rVariables (state)

An rVariable read or write operation is always performed at (for single reads and writes) or starting at (for hyper reads and writes) the current dimension indices for the rVariables in the current CDF. When a CDF is opened or created, the current dimension indices for its rVariables are initialized to zeroes (0,0,...). They may then be explicitly selected using the <SELECT_,rVARs_DIMINDICES_> operation. Note that the current dimension indices for rVariables are maintained for a CDF (not each rVariable) — they apply to all of the rVariables in that CDF. For 0-dimensional rVariables the current dimension indices are not applicable.

dimension counts, rVariables (state)

An rVariable hyper read or write operation is always performed using the current dimension counts for the rVariables in the current CDF. When a CDF is opened or created, the current dimension counts for its rVariables are initialized to the dimension sizes of the rVariables (which specifies the entire array). They may then be explicitly selected using the <SELECT_,rVARs_DIMCOUNTS_> operation. Note that the current dimension counts for rVariables are maintained for a CDF (not each rVariable) — they apply to all of the rVariables in that CDF. For 0-dimensional rVariables the current dimension counts are not applicable.

dimension intervals, rVariables (state)

An rVariable hyper read or write operation is always performed using the current dimension intervals for the rVariables in the current CDF. When a CDF is opened or created, the current dimension intervals for its rVariables are initialized to ones (1,1,...). They may then be explicitly selected using the <SELECT_,rVARs_DIMINTERVALS_> operation. Note that the current dimension intervals for rVariables are maintained for a CDF (not each rVariable) — they apply to all of the rVariables in that CDF. For 0-dimensional rVariables the current dimension intervals are not applicable.

sequential value, rVariable (state)

An rVariable sequential read or write operation is always performed at the current sequential value

for that rVariable. When an rVariable is created (or for each rVariable in a CDF being opened), the current sequential value is set to the first physical value (even if no physical values exist yet). It may then be explicitly selected using the <SELECT_,rVAR_SEQPOS_> operation. Note that a current sequential value is maintained for each rVariable in a CDF.

record number, zVariable (state)

A zVariable read or write operation is always performed at (for single reads and writes) or starting at (for hyper reads and writes) the current record number for the current zVariable in the current CDF. A multiple variable read or write operation is performed at the current record number of each of the zVariables involved. (The record numbers do not have to be the same.) When a zVariable is created (or for each zVariable in a CDF being opened) the current record number for that zVariable is initialized to zero (0). It may then be explicitly selected using the <SELECT_,zVAR_RECNUMBER_> operation (which only affects the current zVariable in the current CDF). Note that a current record number is maintained for each zVariable in a CDF.

record count, zVariable (state)

A zVariable hyper read or write operation is always performed using the current record count for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current record count for that zVariable is initialized to one (1). It may then be explicitly selected using the <SELECT_,zVAR_RECCOUNT_> operation (which only affects the current zVariable in the current CDF). Note that a current record count is maintained for each zVariable in a CDF.

record interval, zVariable (state)

A zVariable hyper read or write operation is always performed using the current record interval for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened) the current record interval for that zVariable is initialized to one (1). It may then be explicitly selected using the <SELECT_,zVAR_RECINTERVAL_> operation (which only affects the current zVariable in the current CDF). Note that a current record interval is maintained for each zVariable in a CDF.

dimension indices, zVariable (state)

A zVariable read or write operation is always performed at (for single reads and writes) or starting at (for hyper reads and writes) the current dimension indices for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current dimension indices for that zVariable are initialized to zeroes (0,0,...). They may then be explicitly selected using the <SELECT_,zVAR_DIMINDICES_> operation (which only affects the current zVariable in the current CDF). Note that current dimension indices are maintained for each zVariable in a CDF. For 0-dimensional zVariables the current dimension indices are not applicable.

dimension counts, zVariable (state)

A zVariable hyper read or write operation is always performed using the current dimension counts for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current dimension counts for that zVariable are initialized to the dimension sizes of that zVariable (which specifies the entire array). They may then be explicitly selected using the <SELECT_,zVAR_DIMCOUNTS_> operation (which only affects the current zVariable in the current CDF). Note that current dimension counts are maintained for each zVariable in a CDF. For 0-dimensional zVariables the current dimension counts are not applicable.

dimension intervals, zVariable (state)

A zVariable hyper read or write operation is always performed using the current dimension intervals

for the current zVariable in the current CDF. When a zVariable is created (or for each zVariable in a CDF being opened), the current dimension intervals for that zVariable are initialized to ones (1,1,...). They may then be explicitly selected using the <SELECT_,zVAR_DIMINTERVALS_> operation (which only affects the current zVariable in the current CDF). Note that current dimension intervals are maintained for each zVariable in a CDF. For 0-dimensional zVariables the current dimension intervals are not applicable.

sequential value, zVariable (state)

A zVariable sequential read or write operation is always performed at the current sequential value for that zVariable. When a zVariable is created (or for each zVariable in a CDF being opened) the current sequential value is set to the first physical value (even if no physical values exist yet). It may then be explicitly selected using the <SELECT_,zVAR_SEQPOS_> operation. Note that a current sequential value is maintained for each zVariable in a CDF.

status code (state)

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.[9]

### 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 5.9 explains how to interpret status codes. Appendix C lists the possible status codes and the type of each: error, warning, or informational.

### 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_path, num_dims, dim_sizes, id,
1             PUT_, CDF_FORMAT_, format,
2                   CDF_MAJORITY_, majority,
3             CREATE_, ATTR_, attr_name, scope, attr_num,
```

[9] The CDF library now maintains the current status code from one call to the next of CDF_lib.

```
4                       rVAR_, 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 rVAR_) 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_path, num_dims, dim_sizes, id, PUT_,
1             CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
2             ATTR_, attr_name, scope, attr_num, rVAR_, 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.[10]

### Syntax

CDF_lib takes 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.

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)
```

[10] If it isn't, call us — we can help you.

where opx is an operation to perform, itemx is the item on which to perform the operation, and argx is a required argument for the operation. The NULL_ operation must be used to end the call to CDF_lib. The completion status is placed into status.

### Operations/Items/Arguments...

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, required arguments, and required preselected objects/states are listed in the following sections. The required preselected objects/states are those objects/states that must be selected (typically with the SELECT_ operation) before a particular operation may be performed. Note that some of the required preselected objects/states have default values as described beginning on page 258.

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 as follows:

in: CHARACTER CDF_path*(*)

Pathname of the CDF to be created. (Do not append 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_path MUST be of type CHARACTER.

in: INTEGER*4 num_dims

Number of dimensions for the rVariables. This can be as few as zero (0) and at most CDF_MAX_DIMS. Note that this must be specified even if the CDF will contain only zVariables.

in: INTEGER*4 dim_sizes(*)

Dimension sizes for the rVariables. 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 rVariables this argument is ignored (but a place holder is necessary). Note that this must be specified even if the CDF will contain only zVariables.

out: `INTEGER*4 id`

CDF identifier to be used in subsequent operations on the CDF.

A CDF is created with the default format, encoding, and variable majority as specified in the configuration file of your CDF distribution. Consult your system manager to determine these defaults. These defaults can be changed with the corresponding `PUT_` operations of `CDF_FORMAT_`, `CDF_ENCODING_`, and `CDF_MAJORITY_`.

`<CLOSE_,CDF_>` MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk.

There are no required preselected objects/states.

`rVAR_` A new rVariable will be created in the current CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF. The created rVariable implicitly becomes the current rVariable (in the current CDF). Required arguments are as follows:

in: `CHARACTER var_name*(*)`

Name of the rVariable 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 rVariable. Specify one of the data types described in Section 5.7.3.

in: `INTEGER*4 num_elements`

Number of elements of the data type at each 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 value of the variable. 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 5.7.7.

in: `INTEGER*4 dim_varys(*)`

Dimension variances. `dim_varys` is a 1-dimensional array containing one element per rVariable dimension. Each element of `dim_varys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 5.7.7. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

out: INTEGER*4 var_num

rVariable. This number must be used in subsequent CDF function calls when referring to this rVariable. An existing rVariable's number may be determined with the <GET_,rVAR_NUMBER_> operation.

The only required preselected object/state is the current CDF.

zVAR_ A new zVariable will be created in the current CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF. The created zVariable implicitly becomes the current zVariable (in the current CDF). Required arguments are as follows:

in: CHARACTER var_name*(*)

Name of the zVariable 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 zVariable. Specify one of the data types described in Section 5.7.3.

in: INTEGER*4 num_elements

Number of elements of the data type at each 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 value of the variable. 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 num_dims

Number of dimensions for the zVariable. This may be as few as zero and at most CDF_MAX_DIMS.

in: INTEGER*4 dim_sizes(*)

The dimension sizes. dim_sizes is a 1-dimensional array containing num_dims elements. Each element of dim_sizes specifies the corresponding dimension size. Each dimension size must be greater than zero (0). For a 0-dimensional zVariable this argument is ignored (but a place holder is necessary).

in: INTEGER*4 rec_vary

Record variance. Specify one of the variances described in Section 5.7.7.

in: INTEGER*4 dim_varys(*)

Dimension variances. dim_varys is a 1-dimensional array containing one element per dimension. Each element of dim_varys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 5.7.7. For a 0-dimensional zVariable this argument is ignored (but a place holder is necessary).

out: INTEGER*4 var_num

Number assigned to the new zVariable. This number must be used in subsequent CDF function calls when referring to this zVariable. An existing zVariable's number may be determined with the <GET_,zVAR_NUMBER_> operation.

The only required preselected object/state is the current CDF.

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 as follows:

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 5.7.8.

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.

The only required preselected object/state is the current CDF.

OPEN_ The following item(s) may be opened:

CDF_ Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

in: CHARACTER CDF_path*(*)

Pathname of the CDF to be opened. (Do not append 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_path MUST be of type CHARACTER.

out: INTEGER*4 id

CDF identifier to be used in subsequent operations of the CDF.

There are no required preselected objects/states.

`DELETE_` The following item(s) may be deleted:

`CDF_` Deletes the current CDF. A CDF must be opened before it can be deleted. When the CDF is deleted, there is no longer a current CDF.

There are no required arguments.

The only required preselected object/state is the current CDF.

`CLOSE_` The following items may be closed:

`CDF_` Closes the current CDF. When the 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 required arguments.

The only required preselected object/state is the current CDF.

`rVAR_` Closes the current rVariable (in the current CDF). This operation in only applicable to multi-file CDFs.

There are no required arguments.

The required preselected objects/states are the current CDF and its current rVariable.

`zVAR_` Closes the current zVariable (in the current CDF). This operation is only applicable to multi-file CDFs.

There are no required arguments.

The required preselected objects/states are the current CDF and its current zVariable.

`SELECT_` The following items may be selected:

`CDF_` Explicitly selects the current CDF. Required arguments are as follows:

in: `INTEGER*4 id`

Identifier of the CDF. This identifier must have been initialized by a successful `<CREATE_,CDF_>` or `<OPEN_,CDF_>` operation.

There are no required preselected objects/states.

`CDF_STATUS_` Selects the current status code. Required arguments are as follows:

in: `INTEGER*4 status`

CDF status code.

There are no required preselected objects/states.

`CDF_READONLY_MODE_` Selects a read-only mode (for the current CDF). Required arguments are as follows:

in: `INTEGER*4 mode`

The read-only mode. Specify one of the read-only modes described in Section 5.7.9.

The only required preselected object/state is the current CDF.

`CDF_zMODE_` Selects a zMode (for the current CDF). Required arguments are as follows:

in: `INTEGER*4 mode`

The zMode. Specify one of the zModes described in Section 5.7.10.

The only required preselected object/state is the current CDF.

`CDF_NEGtoPOSfp0_MODE_`

Selects a `-0.0` to `0.0` mode (for the current CDF). Required arguments are as follows:

in: `INTEGER*4 mode`

The `-0.0` to `0.0` mode. Specify one of the `-0.0` to `0.0` modes described in Section 5.7.11.

The only required preselected object/state is the current CDF.

`CDF_DECODING_` Selects a decoding (for the current CDF). Required arguments are as follows:

in: `INTEGER*4 decoding`

The decoding. Specify one of the decodings described in Section 5.7.5.

The only required preselected object/state is the current CDF.

`CDF_CACHESIZE_` Selects the number of cache buffers to be used for the "`.cdf`" file (for the current CDF). Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`rVAR_` Explicitly selects the current rVariable (in the current CDF) by number. Required arguments are as follows:

in: `INTEGER*4 var_num`

rVariable number.

The only required preselected object/state is the current CDF.

`rVAR_NAME_` Explicitly selects the current rVariable (in the current CDF) by name. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

rVariable 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`.

The only required preselected object/state is the current CDF.

`rVARs_RECNUMBER_` Selects the current record number for all rVariables in the current CDF. Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

The only required preselected object/state is the current CDF.

`rVARs_RECCOUNT_` Selects the current record count for all rVariables in the current CDF. Required arguments are as follows:

in: `INTEGER*4 rec_count`

Record count.

The only required preselected object/state is the current CDF.

`rVARs_RECINTERVAL_` Selects the current record interval for all rVariables in the current CDF. Required arguments are as follows:

in: `INTEGER*4 rec_interval`

Record interval.

The only required preselected object/state is the current CDF.

`rVARs_DIMINDICES_` Selects the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index.

The only required preselected object/state is the current CDF.

`rVARs_DIMCOUNTS_` Selects the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 counts(*)`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` specifies the corresponding dimension count.

The only required preselected object/state is the current CDF.

`rVARs_DIMINTERVALS_` Selects the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 intervals(*)`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` specifies the corresponding dimension interval.

The only required preselected object/state is the current CDF.

`rVAR_SEQPOS_` Selects the current sequential value for sequential access for the current rVariable (in the current CDF). Note that the current sequential value is maintained for each rVariable individually. Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

in: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per rVariable dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_CACHESIZE_` Selects the number of cache buffers to be used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

The required preselected objects/states are the current CDF and its current rVariable.

`rVARs_CACHESIZE_` Selects the number of cache buffers to be used for all of the rVariable files (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`zVAR_` Explicitly selects the current zVariable (in the current CDF) by number. Required arguments are as follows:

in: `INTEGER*4 var_num`

zVariable number.

The only required preselected object/state is the current CDF.

`zVAR_NAME_` Explicitly selects the current zVariable (in the current CDF) by name. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

zVariable 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`.

The only required preselected object/state is the current CDF.

`zVARs_RECNUMBER_` Selects the current record number for each zVariable in the current CDF. This operation is provided to simplify the selection of the current record numbers for the zVariables involved in a multiple variable access operation (see Section 2.3.15). Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

The only required preselected object/state is the current CDF.

`zVAR_RECNUMBER_` Selects the current record number for the current zVariable in the current CDF. Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECCOUNT_` Selects the current record count for the current zVariable in the current CDF. Required arguments are as follows:

in: `INTEGER*4 rec_count`

Record count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECINTERVAL_` Selects the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

in: `INTEGER*4 rec_interval`

Record interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINDICES_` Selects the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMCOUNTS_` Selects the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 counts(*)`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` specifies the corresponding dimension count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINTERVALS_` Selects the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 intervals(*)`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` specifies the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_SEQPOS_` Selects the current sequential value for sequential access for the current zVariable (in the current CDF). Note that the current sequential value is maintained for each zVariable individually. Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

in: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_CACHESIZE_` Selects the number of cache buffers to be used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

The required preselected objects/states are the current CDF and its current zVariable.

`zVARs_CACHESIZE_` Selects the number of cache buffers to be used for all of the zVariable files (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`ATTR_` Explicitly selects the current attribute (in the current CDF). Required arguments are as follows:

in: `INTEGER*4 attr_num`

Attribute number.

The only required preselected object/state is the current CDF.

`ATTR_NAME_` Explicitly selects the current attribute (in the current CDF) by name. Required arguments are as follows:

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`.

The only required preselected object/state is the current CDF.

`gENTRY_` Selects the current gEntry number for all gAttributes in the current CDF. **NOTE:** Only use this operation when dealing with gAttributes. Required arguments are as follows:

in: `INTEGER*4 entry_num`

gEntry number.

The only required preselected object/state is the current CDF.

`rENTRY_` Selects the current rEntry number for all vAttributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes.

Required arguments are as follows:

in: `INTEGER*4 entry_num`

rEntry number.

The only required preselected object/state is the current CDF.

`rENTRY_NAME_` Selects the current rEntry number for all vAttributes (in the current CDF) by rVariable name. The number of the named rVariable becomes the current rEntry number. (The current rVariable is not changed.) **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

rVariable 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`.

The only required preselected object/state is the current CDF.

`zENTRY_` Selects the current zEntry number for all vAttributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `INTEGER*4 entry_num`

zEntry number.

The only required preselected object/state is the current CDF.

`zENTRY_NAME_` Selects the current zEntry number for all vAttributes (in the current CDF) by zVariable name. The number of the named zVariable becomes the current zEntry number. (The current zVariable is not changed.) **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

zVariable 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`.

The only required preselected object/state is the current CDF.

`CONFIRM_` The following items may be confirmed:

`CDF_NAME_` Confirms the pathname of the current CDF. Required arguments are as follows:

out: `CHARACTER CDF_path*(CDF_PATHNAME_LEN)`

Pathname of the CDF. This character string must be large enough to hold `CDF_PATHNAME_LEN` characters and will be blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `CDF_path` MUST be of type `CHARACTER`.

The only required preselected object/state is the current CDF.

CDF_READONLY_MODE_ Confirms the read-only mode for the current CDF. Required arguments are as follows:

out: INTEGER*4 mode

The read-only mode. The read-only modes are described in Section 5.7.9.

The only required preselected object/state is the current CDF.

CDF_zMODE_ Confirms the zMode for the current CDF. Required arguments are as follows:

in: INTEGER*4 mode

The zMode. The zModes are described in Section 5.7.10.

The only required preselected object/state is the current CDF.

CDF_NEGtoPOSfp0_MODE_

Confirms the -0.0 to 0.0 mode for the current CDF. Required arguments are as follows:

in: INTEGER*4 mode

The -0.0 to 0.0 mode. The -0.0 to 0.0 modes are described in Section 5.7.11.

The only required preselected object/state is the current CDF.

CDF_DECODING_ Confirms the decoding for the current CDF. Required arguments are as follows:

in: INTEGER*4 decoding

The decoding. The decodings are described in Section 5.7.5.

The only required preselected object/state is the current CDF.

CDF_CACHESIZE_ Confirms the number of cache buffers being used for the ".cdf" file (for the current CDF). Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: INTEGER*4 num_buffers

The number of cache buffers being used.

The only required preselected object/state is the current CDF.

rVAR_ Confirms the current rVariable (in the current CDF). Required arguments are as follows:

out: INTEGER*4 var_num

rVariable number.

The only required preselected object/state is the current CDF.

`rVAR_EXISTENCE_` Confirms the existence of the named rVariable (in the current CDF). If the rVariable does not exist, an error code will be returned. In any case the current rVariable is not affected. Required arguments are as follows:

out: `CHARACTER var_name*(*)`

The rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

The only required preselected object/state is the current CDF.

`rVARs_RECNUMBER_` Confirms the current record number for all rVariables in the current CDF. Required arguments are as follows:

out: `INTEGER*4 rec_num`

Record number.

The only required preselected object/state is the current CDF.

`rVARs_RECCOUNT_` Confirms the current record count for all rVariables in the current CDF. Required arguments are as follows:

out: `INTEGER*4 rec_count`

Record count.

The only required preselected object/state is the current CDF.

`rVARs_RECINTERVAL_` Confirms the current record interval for all rVariables in the current CDF. Required arguments are as follows:

out: `INTEGER*4 rec_interval`

Record interval.

The only required preselected object/state is the current CDF.

`rVARs_DIMINDICES_` Confirms the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index.

The only required preselected object/state is the current CDF.

`rVARs_DIMCOUNTS_` Confirms the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 counts(*)`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` receives the corresponding dimension count.

The only required preselected object/state is the current CDF.

`rVARs_DIMINTERVALS_` Confirms the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 intervals(*)`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` receives the corresponding dimension interval.

The only required preselected object/state is the current CDF.

`rVAR_SEQPOS_` Confirms the current sequential value for sequential access for the current rVariable (in the current CDF). Note that a current sequential value is maintained for each rVariable individually. Required arguments are as follows:

out: `INTEGER*4 rec_num`

Record number.

out: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per rVariable dimension. Each element of `indices` receives the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_CACHESIZE_` Confirms the number of cache buffers being used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers being used.

The required preselected objects/states are the current CDF and its current rVariable.

`zVAR_` Confirms the current zVariable (in the current CDF). Required arguments are as follows:

out: `INTEGER*4 var_num`

zVariable number.

The only required preselected object/state is the current CDF.

`zVAR_EXISTENCE_` Confirms the existence of the named zVariable (in the current CDF). If the zVariable does not exist, an error code will be returned. In any case the current zVariable is not affected. Required arguments are as

follows:

out: `CHARACTER var_name*(*)`

The zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

The only required preselected object/state is the current CDF.

`zVAR_RECNUMBER_` Confirms the current record number for the current zVariable in the current CDF. Required arguments are as follows:

out: `INTEGER*4 rec_num`

Record number.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECCOUNT_` Confirms the current record count for the current zVariable in the current CDF. Required arguments are as follows:

out: `INTEGER*4 rec_count`

Record count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECINTERVAL_` Confirms the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

out: `INTEGER*4 rec_interval`

Record interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINDICES_` Confirms the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMCOUNTS_` Confirms the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 counts(*)`

Dimension counts. `counts` is a 1-dimensional array containing

one element per dimension. Each element of `counts` receives the corresponding dimension count.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMINTERVALS_` Confirms the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 intervals(*)`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` receives the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_SEQPOS_` Confirms the current sequential value for sequential access for the current zVariable (in the current CDF). Note that a current sequential value is maintained for each zVariable individually. Required arguments are as follows:

out: `INTEGER*4 rec_num`

Record number.

out: `INTEGER*4 indices(*)`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` receives the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_CACHESIZE_` Confirms the number of cache buffers being used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. Section 2.1.4 describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `INTEGER*4 num_buffers`

The number of cache buffers being used.

The required preselected objects/states are the current CDF and its current zVariable.

`ATTR_` Confirms the current attribute (in the current CDF). Required arguments are as follows:

out: `INTEGER*4 attr_num`

Attribute number.

The only required preselected object/state is the current CDF.

ATTR_EXISTENCE_ Confirms the existence of the named attribute (in the current CDF). If the attribute does not exist, an error code will be returned. In any case the current attribute is not affected. Required arguments are as follows:

out: `CHARACTER attr_name*(*)`

The attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters.

The only required preselected object/state is the current CDF.

gENTRY_ Confirms the current gEntry number for all attributes in the current CDF. **NOTE:** Only use this operation when dealing with gAttributes. Required arguments are as follows:

out: `INTEGER*4 entry_num`

gEntry number.

The only required preselected object/state is the current CDF.

rENTRY_ Confirms the current rEntry number for all attributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

out: `INTEGER*4 entry_num`

rEntry number.

The only required preselected object/state is the current CDF.

zENTRY_ Confirms the current zEntry number for all attributes in the current CDF. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

out: `INTEGER*4 entry_num`

zEntry number.

The only required preselected object/state is the current CDF.

gENTRY_EXISTENCE_ Confirms the existence of the specified gEntry (for the current attribute in the current CDF). If the gEntry does not exist, an error code will be returned. In any case the current gEntry number is not affected. **NOTE:** Only use this operation when dealing with gAttributes. Required arguments are as follows:

in: `INTEGER*4 entry_num`

The gEntry number.

The required preselected objects/states are the current CDF and its current attribute.

rENTRY_EXISTENCE_ Confirms the existence of the specified rEntry (for the current attribute in the current CDF). If the rEntry does not exist, an error

code will be returned. In any case the current rEntry number is not affected. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `INTEGER*4 entry_num`

The rEntry number.

The required preselected objects/states are the current CDF and its current attribute.

`zENTRY_EXISTENCE_` Confirms the existence of the specified zEntry (for the current attribute in the current CDF). If the zEntry does not exist, an error code will be returned. In any case the current zEntry number is not affected. **NOTE:** Only use this operation when dealing with vAttributes. Required arguments are as follows:

in: `INTEGER*4 entry_num`

The zEntry number.

The required preselected objects/states are the current CDF and its current attribute.

`PUT_` The following items may be written (put):

`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 pad value) or attribute entries have been written. Required arguments are as follows:

in: `INTEGER*4 encoding`

New data encoding. Specify one of the encodings described in Section 5.7.4.

The only required preselected object/state is the current CDF.

`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 as follows:

in: `INTEGER*4 majority`

New variable majority. Specify one of the majorities described in Section 5.7.6.

The only required preselected object/state is the current CDF.

`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 as follows:

in: `INTEGER*4 format`

New CDF format. Specify one of the formats described in Section 5.7.2.

The only required preselected object/state is the current CDF.

`rVAR_NAME_` Renames the current rVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

New name of the rVariable. 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`.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATASPEC_` Respecifies the data specification (data type and number of elements) of the current rVariable (in the current CDF). An rVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see Section 2.2.9) and the number of elements are the same. Required arguments are as follows:

in: `INTEGER*4 data_type`

New data type. Specify one of the data types described in Section 5.7.3.

in: `INTEGER*4 num_elements`

Number of elements of the data type at each 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 value.) For non-character string data types this must be one (1) — multiple elements are not allowed for non-character string data types.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_RECVARY_` Respecifies the record variance of the current rVariable (in the current CDF). An rVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `INTEGER*4 rec_vary`

New record variance. Specify one of the variances described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DIMVARYS_` Respecifies the dimension variances of the current rVariable (in the current CDF). An rVariable's dimension variances may not be changed

if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 dim_varys(*)`

New dimension variances. `dim_varys` is a 1-dimensional array containing one element per dimension. Each element of `dim_varys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_PADVALUE_` Specifies the pad value for the current rVariable (in the current CDF). An rVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). Section 2.3.16 describes variable pad values. Required arguments are as follows:

in: `<type> value`

Pad value. `<type>` depends on the data type of the rVariable. The pad value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_INITIALRECS_` Specifies the number of records to initially write to the current rVariable (in the current CDF). This may be specified only once per rVariable and before any other records have been written to that rVariable. If a pad value has not yet been specified, the default is used (see Section 2.3.16). If a pad value has been explicitly specified, that value is written to the records. Section 2.3.7 describes initial records. Required arguments are as follows:

in: `INTEGER*4 n_initial_recs`

Number of records to write.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_EXTENDRECS_` Specifies the number of physical records to extend the current rVariable (in the current CDF) whenever that rVariable needs to be extended. Section 2.3.7 describes extend records. **NOTE:** The number of extend records has no effect for NRV rVariables or multi-file CDFs. Required arguments are as follows:

in: `INTEGER*4 n_extend_recs`

Number of records to extend. A value of zero (0) indicates that the default number of records should be used.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATA_` Writes one value to the current rVariable (in the current CDF). The value is written at the current record number and current dimension indices for the rVariables (in the current CDF). Required arguments are as follows:

in: `<type> value`

Value. `<type>` depends on the data type of the rVariable. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

`rVAR_HYPERDATA_` Writes one or more values to the current rVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

in: `<type> buffer`

Values. `<type>` depends on the data type of the rVariable. The values starting at memory address `buffer` are written to the CDF. **NOTE:** On UNIX systems, if the rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

`rVAR_SEQDATA_` Writes one value to the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the rVariable, the rVariable is extended as necessary. Required arguments are as follows:

in: `<type> value`

Value. `<type>` depends on the data type of the rVariable. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note that the current sequential value for an rVariable increments automatically as values are written.

`rVARs_RECDATA_` Writes full-physical records to one or more rVariables (in the current CDF). The full-physical records are written at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `INTEGER*4 num_vars`

The number of rVariables to which to write. This must be at least one (1).

in: `INTEGER*4 var_nums(*)`

The rVariables to which to write. This array, whose size is determined by the value of `num_vars`, contains rVariable numbers. The rVariable numbers can be listed in any order.

in: `<type> buffer`

The buffer of full-physical rVariable records to be written. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to get the full-physical records being written.) The order of the full-physical rVariable records in this buffer must agree with the rVariable numbers listed in `varNums` and this buffer must be contiguous — there can be no spacing between full-physical rVariable records. Be careful if using Fortran STRUCTUREs to store multiple full-physical rVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See Section 2.3.15 for more details on how to create this buffer.

The required preselected objects/states are the current CDF and its current record number for rVariables.

`zVAR_NAME_` Renames the current zVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

New name of the zVariable. 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`.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATASPEC_` Respecifies the data specification (data type and number of elements) of the current zVariable (in the current CDF). A zVariable's data spec-

ification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see Section 2.2.9) and the number of elements are the same. Required arguments are as follows:

in: `INTEGER*4 data_type`

New data type. Specify one of the data types described in Section 5.7.3.

in: `INTEGER*4 num_elements`

Number of elements of the data type at each 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 value.) For non-character string data types this must be one (1) — multiple elements are not allowed for non-character string data types.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_RECVARY_` Respecifies the record variance of the current zVariable (in the current CDF). A zVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `INTEGER*4 rec_vary`

New record variance. Specify one of the variances described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMVARYS_` Respecifies the dimension variances of the current zVariable (in the current CDF). A zVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `INTEGER*4 dim_varys(*)`

New dimension variances. `dim_varys` is a 1-dimensional array containing one element per dimension. Each element of `dim_varys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_PADVALUE_` Specifies the pad value for the current zVariable (in the current CDF). A zVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). Section 2.3.16 describes variable pad values. Required

arguments are as follows:

in: `<type> value`

Pad value. `<type>` depends on the data type of the zVariable. The pad value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_INITIALRECS_` Specifies the number of records to initially write to the current zVariable (in the current CDF). This may be specified only once per zVariable and before any other records have been written to that zVariable. If a pad value has not yet been specified, the default is used (see Section 2.3.16). If a pad value has been explicitly specified, that value is written to the records. Section 2.3.7 describes initial records. Required arguments are as follows:

in: `INTEGER*4 n_initial_recs`

Number of records to write.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_EXTENDRECS_` Specifies the number of physical records to extend the current zVariable (in the current CDF) whenever that zVariable needs to be extended. Section 2.3.7 describes extend records. **NOTE:** The number of extend records has no effect for NRV zVariables or multi-file CDFs. Required arguments are as follows:

in: `INTEGER*4 n_extend_recs`

Number of records to extend. A value of zero (0) indicates that the default number of records should be used.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATA_` Writes one value to the current zVariable (in the current CDF). The value is written at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

in: `<type> value`

Value. `<type>` depends on the data type of the zVariable. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the

current dimension indices for the zVariable.

`zVAR_HYPERDATA_` Writes one or more values to the current zVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

in: `<type> buffer`

Values. `<type>` depends on the data type of the zVariable. The values starting at memory address `buffer` are written to the CDF. **NOTE:** On UNIX systems, if the zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

`zVAR_SEQDATA_` Writes one value to the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the zVariable, the zVariable is extended as necessary. Required arguments are as follows:

in: `<type> value`

Value. `<type>` depends on the data type of the zVariable. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are written.

`zVARs_RECDATA_` Writes full-physical records to one or more zVariables (in the current CDF). The full physical record for a particular zVariable is written at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `INTEGER*4 num_vars`

The number of zVariables to which to write. This must be at least one (1).

in: `INTEGER*4 var_nums(*)`

The zVariables to which to write. This array, whose size is

determined by the value of num_vars, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: <type> buffer

The buffer of full-physical zVariable records to be written. <type> must be a Fortran variable that will be passed by reference and cannot be of type CHARACTER. (The CDF library is expecting an address at which to get the full-physical records being written.) The order of the full-physical zVariable records in this buffer must agree with the zVariable numbers listed in varNums, and this buffer must be contiguous — there can be no spacing between full-physical zVariable records. Be careful if using Fortran STRUCTUREs to store multiple full-physical zVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See Section 2.3.15 for more details on how to create this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, <SELECT_,zVARs_RECNUMBER_>, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using <SELECT_,zVAR_RECNUMBER_>).

ATTR_SCOPE_ Respecifies the scope for the current attribute (in the current CDF). Required arguments are as follows:

in: INTEGER*4 scope

New attribute scope. Specify one of the scopes described in Section 5.7.8.

The required preselected objects/states are the current CDF and its current attribute.

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 as follows:

in: CHARACTER attr_name*(*)

New 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.

The required preselected objects/states are the current CDF and its current attribute.

gENTRY_DATASPEC_ Modifies the data specification (data type and number of elements) of the gEntry at the current gEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent and the number of elements must not be changed. (Equivalent data types are described in Section 2.2.9.) **NOTE:** This operation should only be

performed on gAttributes. Required arguments are as follows:

in: `INTEGER*4 data_type`

New data type of the gEntry. Specify one of the data types described in Section 5.7.3.

in: `INTEGER*4 num_elements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`rENTRY_DATASPEC_` Modifies the data specification (data type and number of elements) of the rEntry at the current rEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent and the number of elements must not be changed. (Equivalent data types are described in Section 2.2.9.) **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `INTEGER*4 data_type`

New data type of the rEntry. Specify one of the data types described in Section 5.7.3.

in: `INTEGER*4 num_elements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`zENTRY_DATASPEC_` Modifies the data specification (data type and number of elements) of the zEntry at the current zEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent and the number of elements must not be changed. (Equivalent data types are described in Section 2.2.9.) **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `INTEGER*4 data_type`

New data type of the zEntry. Specify one of the data types described in Section 5.7.3.

in: `INTEGER*4 num_elements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`gENTRY_DATA_` Writes a gEntry to the current attribute at the current gEntry number (in the current CDF). An existing gEntry may be overwritten with a new gEntry having the same data specification (data type and number of elements) or a different data specification. **NOTE:** This operation should only be performed on gAttributes. Required arguments are as follows:

in: `INTEGER*4 data_type`

Data type of the gEntry. Specify one of the data types described in Section 5.7.3.

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 gEntry. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the gEntry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`rENTRY_DATA_` Writes an rEntry to the current attribute at the current rEntry number (in the current CDF). An existing rEntry may be overwritten with a new rEntry having the same data specification (data type and number of elements) or a different data specification. **NOTE:** This operation should only be performed on vAttributes. Required arguments are as follows:

in: `INTEGER*4 data_type`

Data type of the rEntry. Specify one of the data types described in Section 5.7.3.

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 rEntry. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the rEntry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`zENTRY_DATA_` Writes a zEntry to the current attribute at the current zEntry number (in the current CDF). An existing zEntry may be overwritten with a new zEntry having the same data specification (data type and number of elements) or a different data specification. **NOTE:** This operation

should only be performed on vAttributes. Required arguments are as follows:

in: `INTEGER*4 data_type`

Data type of the zEntry. Specify one of the data types described in Section 5.7.3.

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 zEntry. The value is written to the CDF from memory address `value`. **NOTE:** On UNIX systems, if the zEntry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`GET_` The following items may be read (gotten):

`rVARs_NUMDIMS_` Inquires the number of dimensions for the rVariables in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_dims`

Number of dimensions.

The only required preselected object/state is the current CDF.

`rVARs_DIMSIZES_` Inquires the size of each dimension for the rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 dim_sizes(*)`

Dimension sizes. `dim_sizes` is a 1-dimensional array containing one element per rVariable dimension. Each element of `dim_sizes` receives the corresponding dimension size.

The only required preselected object/state is the current CDF.

`zVAR_NUMDIMS_` Inquires the number of dimensions for the current zVariable in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_dims`

Number of dimensions.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DIMSIZES_` Inquires the size of each dimension for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 dim_sizes(*)`

Dimension sizes. `dim_sizes` is a 1-dimensional array containing one element per dimension. Each element of `dim_sizes` receives the corresponding dimension size.

The required preselected objects/states are the current CDF and its current zVariable.

`CDF_ENCODING_` Inquires the data encoding of the current CDF. Required arguments are as follows:

out: `INTEGER*4 encoding`

Data encoding. The encodings are described in Section 5.7.4.

The only required preselected object/state is the current CDF.

`CDF_MAJORITY_` Inquires the variable majority of the current CDF. Required arguments are as follows:

out: `INTEGER*4 majority`

Variable majority. The majorities are described in Section 5.7.6.

The only required preselected object/state is the current CDF.

`CDF_FORMAT_` Inquires the format of the current CDF. Required arguments are as follows:

out: `INTEGER*4 format`

CDF format. The formats are described in Section 5.7.2.

The only required preselected object/state is the current CDF.

`CDF_COPYRIGHT_` Reads the copyright notice for the CDF library that created the current CDF. Required arguments are as follows:

out: `CHARACTER copy_right*(CDF_COPYRIGHT_LEN)`

CDF copyright text. This character string must be large enough hold `CDF_COPYRIGHT_LEN` characters and will be blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `copy_right` MUST be of type `CHARACTER`.

The only required preselected object/state is the current CDF.

`LIB_COPYRIGHT_` Reads the copyright notice for the CDF library being used. Required arguments are as follows:

out: `CHARACTER copy_right*(CDF_COPYRIGHT_LEN)`

CDF library copyright text. This character string must be large enough hold `CDF_COPYRIGHT_LEN` characters and will be

blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `copy_right` MUST be of type `CHARACTER`.

There are no required preselected objects/states.

`CDF_NUMrVARS_` Inquires the number of rVariables in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_vars`

Number of rVariables.

The only required preselected object/state is the current CDF.

`CDF_NUMzVARS_` Inquires the number of zVariables in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_vars`

Number of zVariables.

The only required preselected object/state is the current CDF.

`CDF_NUMATTRS_` Inquires the number of attributes in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_attrs`

Number of attributes.

The only required preselected object/state is the current CDF.

`CDF_NUMgATTRS_` Inquires the number of gAttributes in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_attrs`

Number of gAttributes.

The only required preselected object/state is the current CDF.

`CDF_NUMvATTRS_` Inquires the number of vAttributes in the current CDF. Required arguments are as follows:

out: `INTEGER*4 num_attrs`

Number of vAttributes.

The only required preselected object/state is the current CDF.

`rVARs_MAXREC_` Inquires the maximum record number of the rVariables in the current CDF. A value of zero (0) indicates that no records have been written to the rVariables in the CDF. The maximum record number for an individual rVariable may be inquired using `<GET_,rVAR_MAXREC_>`. Required arguments are as follows:

out: `INTEGER*4 max_rec`

Maximum record number.

The only required preselected object/state is the current CDF.

zVARs_MAXREC_ Inquires the maximum record number of the zVariables in the current CDF. A value of zero (0) indicates that no records have been written to the zVariables in the CDF. The maximum record number for an individual zVariable may be inquired using <GET_,zVAR_MAXREC_>. Required arguments are as follows:

out: INTEGER*4 max_rec

Maximum record number.

The only required preselected object/state is the current CDF.

CDF_VERSION_ Inquires the version number of the CDF library that created the current CDF. Required arguments are as follows:

out: INTEGER*4 version

Version number.

The only required preselected object/state is the current CDF.

CDF_RELEASE_ Inquires the release number of the CDF library that created the current CDF. Required arguments are as follows:

out: INTEGER*4 release

Release number.

The only required preselected object/state is the current CDF.

CDF_INCREMENT_ Inquires the incremental number of the CDF library that created the current CDF. Required arguments are as follows:

out: INTEGER*4 increment

Incremental number.

The only required preselected object/state is the current CDF.

LIB_VERSION_ Inquires the version number of the CDF library being used. Required arguments are as follows:

out: INTEGER*4 version

Version number.

There are no required preselected objects/states.

LIB_RELEASE_ Inquires the release number of the CDF library being used. Required arguments are as follows:

out: INTEGER*4 release

Release number.

There are no required preselected objects/states.

LIB_INCREMENT_ Inquires the incremental number of the CDF library being used. Required arguments are as follows:

out: `INTEGER*4 increment`

Incremental number.

There are no required preselected objects/states.

`LIB_subINCREMENT_` Inquires the subincremental character of the CDF library being used. Required arguments are as follows:

out: `CHARACTER*1 subincrement`

Subincremental character. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `subincrement` MUST be of type `CHARACTER`.

There are no required preselected objects/states.

`rVAR_NAME_` Inquires the name of the current rVariable (in the current CDF). Required arguments are as follows:

out: `CHARACTER var_name*(CDF_VAR_NAME_LEN)`

Name of the rVariable. This character string must be large enough to hold `CDF_VAR_NAME_LEN` characters and will be blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `var_name` MUST be of type `CHARACTER`.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATATYPE_` Inquires the data type of the current rVariable (in the current CDF). Required arguments are as follows:

out: `INTEGER*4 data_type`

Data type. The data types are described in Section 5.7.3.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_NUMELEMS_` Inquires the number of elements (of the data type) for the current rVariable (in the current CDF). Required arguments are as follows:

out: `INTEGER*4 num_elements`

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

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_RECVARY_` Inquires the record variance of the current rVariable (in the current CDF). Required arguments are as follows:

out: `INTEGER*4 rec_vary`

Record variance. The variances are described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DIMVARYS_` Inquires the dimension variances of the current rVariable (in the current CDF). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `INTEGER*4 dim_varys(*)`

Dimension variances. `dim_varys` is a 1-dimensional array containing one element per rVariable dimension. Each element of `dim_varys` receives the corresponding dimension variance. The variances are described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_NUMBER_` Gets the number of the named rVariable (in the current CDF). Note that this operation does not select the current rVariable. Required arguments are as follows:

in: `CHARACTER var_name*(*)`

The rVariable 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 rVariable number.

The only required preselected object/state is the current CDF.

`rVAR_PADVALUE_` Inquires the pad value of the current rVariable (in the current CDF). Required arguments are as follows:

out: `<type> value`

Pad value. `<type>` depends on the data type of the rVariable. The pad value is read from the CDF and placed into memory at address `value`. This buffer must be large enough to hold the pad value. **NOTE:** On UNIX systems, if the rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_EXTENDRECS_` Inquires the number of extend records for the current rVariable (in the current CDF). Extend records are described in Section 2.3.7. Required arguments are as follows:

out: `INTEGER*4 n_extend_recs`

Number of records to extend. A value of zero (0) indicates that the default number of records is being used.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_MAXREC_` Inquires the maximum record number for the current rVariable (in the current CDF). For rVariables 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. Required arguments are as follows:

out: `INTEGER*4 var_max_rec`

Maximum record number.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_MAXallocREC_` Inquires the maximum record number allocated for the current rVariable (in the current CDF). This only has significance for rVariables that are not NRV and are in single-file CDFs. Required arguments are as follows:

out: `INTEGER*4 var_max_rec_alloc`

Maximum record number allocated.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_nINDEXRECORDS_` Inquires the number of index records for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *num_records`

Number of index records.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_nINDEXENTRIES_` Inquires the number of index entries for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *num_entries`

Number of index entries.

The required preselected objects/states are the current CDF and its current rVariable.

`rVAR_DATA_` Reads a value from the current rVariable (in the current CDF). The value is read at the current record number and current dimension

indices for the rVariables (in the current CDF). Required arguments are as follows:

out: `<type> value`

Value. `<type>` depends on the data type of the rVariable. 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 rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

`rVAR_HYPERDATA_` Reads one or more values from the current rVariable (in the current CDF). The values are read based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

out: `<type> buffer`

Value(s). `<type>` depends on the data type of the rVariable. 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 rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

`rVAR_SEQDATA_` Reads one value from the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the rVariable. Required arguments are as follows:

out: `<type> value`

Value. `<type>` depends on the data type of the rVariable. 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 rVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note

that the current sequential value for an rVariable increments automatically as values are read.

`rVARs_RECDATA_` Reads full-physical records from one or more rVariables (in the current CDF). The full-physical records are read at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `INTEGER*4 num_vars`

The number of rVariables from which to read. This must be at least one (1).

in: `INTEGER*4 var_nums(*)`

The rVariables from which to read. This array, whose size is determined by the value of `num_vars`, contains rVariable numbers. The rVariable numbers can be listed in any order.

in: `<type> buffer`

The buffer into which the full-physical rVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to place the full-physical records being read.) The order of the full-physical rVariable records in this buffer will correspond to the rVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical rVariable records. Be careful if using Fortran STRUCTUREs to receive multiple full-physical rVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See Section 2.3.15 for more details on how to create this buffer.

The required preselected objects/states are the current CDF and its current record number for rVariables.

`zVAR_NAME_` Inquires the name of the current zVariable (in the current CDF). Required arguments are as follows:

out: `CHARACTER var_name*(CDF_VAR_NAME_LEN)`

Name of the zVariable. This character string must be large enough to hold `CDF_VAR_NAME_LEN` characters and will be blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `var_name` MUST be of type `CHARACTER`.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATATYPE_` Inquires the data type of the current zVariable (in the current CDF). Required arguments are as follows:

out: INTEGER*4 data_type

Data type. The data types are described in Section 5.7.3.

The required preselected objects/states are the current CDF and its current zVariable.

zVAR_NUMELEMS_ Inquires the number of elements (of the data type) for the current zVariable (in the current CDF). Required arguments are as follows:

out: INTEGER*4 num_elements

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

The required preselected objects/states are the current CDF and its current zVariable.

zVAR_RECVARY_ Inquires the record variance of the current zVariable (in the current CDF). Required arguments are as follows:

out: INTEGER*4 rec_vary

Record variance. The variances are described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current zVariable.

zVAR_DIMVARYS_ Inquires the dimension variances of the current zVariable (in the current CDF). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: INTEGER*4 dim_varys(*)

Dimension variances. dim_varys is a 1-dimensional array containing one element per dimension. Each element of dim_varys receives the corresponding dimension variance. The variances are described in Section 5.7.7.

The required preselected objects/states are the current CDF and its current zVariable.

zVAR_NUMBER_ Gets the number of the named zVariable (in the current CDF). Note that this operation does not select the current zVariable. Required arguments are as follows:

in: CHARACTER var_name*(*)

The zVariable 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 rVariable number.

The only required preselected object/state is the current CDF.

`zVAR_PADVALUE_` Inquires the pad value of the current zVariable (in the current CDF). Required arguments are as follows:

out: `<type> value`

Pad value. `<type>` depends on the data type of the zVariable. The pad value is read from the CDF and placed into memory at address `value`. This buffer must be large enough to hold the pad value. **NOTE:** On UNIX systems, if the zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_EXTENDRECS_` Inquires the number of extend records for the current zVariable (in the current CDF). Extend records are described in Section 2.3.7. Required arguments are as follows:

out: `INTEGER*4 n_extend_recs`

Number of records to extend. A value of zero (0) indicates that the default number of records is being used.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_MAXREC_` Inquires the maximum record number for the current zVariable (in the current CDF). For zVariables 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. Required arguments are as follows:

out: `INTEGER*4 var_max_rec`

Maximum record number.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_MAXallocREC_` Inquires the maximum record number allocated for the current zVariable (in the current CDF). This has significance only for zVariables that are not NRV and are in single-file CDFs. Required arguments are as follows:

out: `INTEGER*4 var_max_rec_alloc`

Maximum record number allocated.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_nINDEXRECORDS_` Inquires the number of index records for the current zVariable (in the current CDF). This has significance only for zVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used

for variable records in a single-file CDF. Required arguments are as follows:

out: `long *num_records`

Number of index records.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_nINDEXENTRIES_` Inquires the number of index entries for the current zVariable (in the current CDF). This has significance only for zVariables that are in single-file CDFs. Section 2.2.6 describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *num_entries`

Number of index entries.

The required preselected objects/states are the current CDF and its current zVariable.

`zVAR_DATA_` Reads a value from the current zVariable (in the current CDF). The value is read at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

out: `<type> value`

Value. `<type>` depends on the data type of the zVariable. 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 zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the current dimension indices for the zVariable.

`zVAR_HYPERDATA_` Reads one or more values from the current zVariable (in the current CDF). The values are read based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

out: `<type> buffer`

Value(s). `<type>` depends on the data type of the zVariable. 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 zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

`zVAR_SEQDATA_` Reads one value from the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the zVariable. Required arguments are as follows:

out: `<type> value`

Value. `<type>` depends on the data type of the zVariable. 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 zVariable has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are read.

`zVARs_RECDATA_` Reads full-physical records from one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is read at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `INTEGER*4 num_vars`

The number of zVariables from which to read. This must be at least one (1).

in: `INTEGER*4 var_nums(*)`

The zVariables from which to read. This array, whose size is determined by the value of `num_vars`, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: `<type> buffer`

The buffer into which the full-physical zVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to place the full-physical records being read.) The order of the full-physical zVariable records in this buffer will correspond to the zVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical zVariable records. Be careful if using For-

tran STRUCTUREs to receive multiple full-physical zVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See Section 2.3.15 for more details on how to create this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`ATTR_SCOPE_` Inquires the scope of the current attribute (in the current CDF). Required arguments are as follows:

out: `INTEGER*4 scope`

Attribute scope. The scopes are described in Section 5.7.8.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NAME_` Inquires the name of the current attribute (in the current CDF). Required arguments are as follows:

out: `CHARACTER attr_name*(CDF_ATTR_NAME_LEN)`

Attribute name. This character string must be large enough to hold `CDF_ATTR_NAME_LEN` characters and will be blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of `CDF_lib`, `attr_name` MUST be of type `CHARACTER`.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMBER_` Gets the number of the named attribute (in the current CDF). Note that this operation does not select the current attribute. Required arguments are as follows:

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.

The only required preselected object/state is the current CDF.

`ATTR_MAXgENTRY_` Inquires the maximum gEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of gEntries for the attribute. **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `INTEGER*4 max_entry`

The maximum gEntry number for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMgENTRIES_` Inquires the number of gEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum gEntry number used. **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `INTEGER*4 num_entries`

The number of gEntries for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_MAXrENTRY_` Inquires the maximum rEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of rEntries for the attribute. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `INTEGER*4 max_entry`

The maximum rEntry number for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMrENTRIES_` Inquires the number of rEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum rEntry number used. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `INTEGER*4 num_entries`

The number of rEntries for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_MAXzENTRY_` Inquires the maximum zEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of zEntries for the attribute. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `INTEGER*4 max_entry`

The maximum zEntry number for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`ATTR_NUMzENTRIES_` Inquires the number of zEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum zEntry number used. **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `INTEGER*4 num_entries`

The number of zEntries for the attribute.

The required preselected objects/states are the current CDF and its current attribute.

`gENTRY_DATATYPE_` Inquires the data type of the gEntry at the current gEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `INTEGER*4 data_type`

Data type. The data types are described in Section 5.7.3.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`gENTRY_NUMELEMS_` Inquires the number of elements (of the data type) of the gEntry at the current gEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

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.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`gENTRY_DATA_` Reads the gEntry data value from the current attribute at the current gEntry number (in the current CDF). **NOTE:** Use this operation with gAttributes only. Required arguments are as follows:

out: `<type> value`

Value. `<type>` depends on the data type of the gEntry. 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 gEntry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

`rENTRY_DATATYPE_` Inquires the data type of the rEntry at the current rEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `INTEGER*4 data_type`

Data type. The data types are described in Section 5.7.3.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`rENTRY_NUMELEMS_` Inquires the number of elements (of the data type) of the rEntry at the current rEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

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.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`rENTRY_DATA_` Reads the rEntry data value from the current attribute at the current rEntry number (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `<type> value`

Value. `<type>` depends on the data type of the rEntry. 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 rEntry has a character string data type, then `<type>` MUST be `CHARACTER` to ensure the proper operation of `CDF_lib`.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

`zENTRY_DATATYPE_` Inquires the data type of the zEntry at the current zEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: `INTEGER*4 data_type`

Data type. The data types are described in Section 5.7.3.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

`zENTRY_NUMELEMS_` Inquires the number of elements (of the data type) of the zEntry at the current zEntry number for the current attribute (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

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.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

zENTRY_DATA_ Reads the zEntry data value from the current attribute at the current zEntry number (in the current CDF). **NOTE:** Use this operation with vAttributes only. Required arguments are as follows:

out: <type> value

Value. <type> depends on the data type of the zEntry. 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 zEntry has a character string data type, then <type> MUST be CHARACTER to ensure the proper operation of CDF_lib.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

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 as follows:

out: CHARACTER text*(CDF_STATUSTEXT_LEN)

Text explaining the status code. This string must be large enough to hold CDF_STATUSTEXT_LEN characters and will be blank padded if necessary. **NOTE:** On UNIX systems, for the proper operation of CDF_lib, text MUST be of type CHARACTER.

The only required preselected object/state is the current status code.

### More Examples

Several more examples of the use of CDF_lib follow. In each example it is assumed that the current CDF has already been selected (either implicitly by creating/opening the CDF or explicitly with <SELECT_,CDF_>).

**rVariable Creation.** In this example an rVariable will be created with a pad value being specified; initial records will be written; and the number of records to extend the rVariable when necessary will be specified. Note that the pad value was specified before the initial records. This results in the specified pad value being written. Had the pad value not been specified first, the initial records would have been written with the default pad value. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>cdf.h'
.
.
INTEGER*4 status                    ! Status returned from CDF library.
INTEGER*4 dim_varys(2)              ! Dimension variances.
```

```
INTEGER*4 var_num                    ! rVariable number.
REAL*4    pad_value                  ! Pad value.

DATA pad_value/-999.9/
.
.
dim_varys(1) = VARY
dim_varys(2) = VARY
CALL CDF_lib (CREATE_, rVAR_, 'HUMIDITY', CDF_REAL4, 1, VARY,
1                             dim_varys, var_num,
2             PUT_, rVAR_PADVALUE_, pad_value,
3                   rVAR_INITIALRECS_, 500,
4                   rVAR_EXTENDRECS_, 50,
5             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

**zVariable Creation (Character String Data Type).** In this example a zVariable with a character string data type will be created with a pad value being specified. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 status                     ! Status returned from CDF library.
INTEGER*4 dim_varys(1)               ! Dimension variances.
INTEGER*4 var_num                    ! zVariable number.
INTEGER*4 num_dims                   ! Number of dimension.
INTEGER*4 dim_sizes(1)               ! Dimension sizes.
INTEGER*4 num_elems                  ! Number of elements (of the data
                                     ! type).
CHARACTER pad_value                  ! Pad value.

DATA pad_value/'**********'/,
0    num_dims/1/,
1    dim_sizes/20/,
2    num_elems/10/
.
.
dim_varys(0) = VARY
CALL CDF_lib (CREATE_, zVAR_, 'Station', CDF_CHAR, num_elems, num_dims,
1                      dim_sizes, NOVARY, dim_varys, var_num,
2             PUT_, zVAR_PADVALUE_, pad_value,
3             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

**Hyper Read with Subsampling.** In this example an rVariable will be subsampled in a CDF whose rVariables are 2-dimensional and have dimension sizes `[100,200]`. The CDF is column major, and the data type of the rVariable is `CDF_INT2`. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
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 rVariable 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_, rVAR_NAME_, 'BRIGHTNESS',
1                      rVARs_RECCOUNT_, rec_count,
2                      rVARs_RECINTERVAL_, rec_interval,
3                      rVARs_DIMINDICES_, indices,
4                      rVARs_DIMCOUNTS_, counts,
5                      rVARs_DIMINTERVALS_, intervals,
6              NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

DO rec_num = 1, max_rec
  CALL CDF_lib (SELECT_, rVARs_RECNUMBER_, rec_num,
  1              GET_, rVAR_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`. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 status               ! Status returned from CDF library.
.
.
CALL CDF_lib (SELECT_, 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 zVariable will be averaged. The values will be read using the sequential access method (see Section 2.3.14). Each value in each record will be read and averaged. It is assumed that the data type of the zVariable has been determined to be `CDF_REAL4`. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 status               ! Status returned from CDF library.
INTEGER*4 var_num              ! zVariable number.
INTEGER*4 rec_num              ! Record number, start at first record.
INTEGER*4 indices(2)           ! Dimension indices.
REAL*4    value                ! Value read.
REAL*8    sum                  ! Sum of all values.
INTEGER*4 count                ! Number of values.
REAL*4    ave                  ! Average value.

DATA indices/1,1/, sum/0.0/, count/0/, rec_num/1/
.
.
CALL CDF_lib (GET_, zVAR_NUMBER_, 'FLUX', var_num,
1             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_lib (SELECT_, zVAR_, var_num,
1                      zVAR_SEQPOS_, rec_num, indices,
```

```
2             GET_, zVAR_SEQDATA_, value,
3             NULL_, status)

DO WHILE (status .GE. CDF_OK)
  sum = sum + value
  count = count + 1
  CALL CDF_lib (GET_, zVAR_SEQDATA_, value,
  1             NULL_, status)
END DO

IF (status .NE. END_OF_VAR) CALL UserStatusHandler (status)

ave = sum / count
.
.
```

**Attribute rEntry Writes.** In this example a set of attribute rEntries for a particular rVariable will be written. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
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_, rENTRY_NAME_, 'LATITUDE',
1                      ATTR_NAME_, 'FIELDNAM',
2             PUT_, rENTRY_DATA_, CDF_CHAR, 20, 'Latitude           ',
3             SELECT_, ATTR_NAME_, 'SCALE',
4             PUT_, rENTRY_DATA_, CDF_REAL4, 2, scale,
5             SELECT_, ATTR_NAME_, 'UNITS',
6             PUT_, rENTRY_DATA_, CDF_CHAR, 20, 'Degrees north      ',
7             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

**Multiple zVariable Write.** In this example full-physical records will be written to the zVariables in a CDF. Note the ordering of the zVariables (see Section 2.3.15). It is assumed that the current CDF has already been selected.

```
.
.
```

```
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 status                  ! Status returned from CDF library.
INTEGER*2 time                    ! 'Time' value.
BYTE      vector_a(3)             ! 'vectorA' values.
REAL*8    vector_b(5)             ! 'vectorB' values.
INTEGER*4 rec_number              ! Record number.
BYTE      buffer(45)              ! Buffer of full-physical records.
INTEGER*4 var_numbers(3)          ! Variable numbers.

EQUIVALENCE (vector_b, buffer(1))
EQUIVALENCE (time, buffer(41))
EQUIVALENCE (vector_a, buffer(43))
.
.
CALL CDF_lib (GET_, zVAR_NUMBER_, 'vectorB', var_numbers(0),
1                  zVAR_NUMBER_, 'time', var_numbers(1),
2                  zVAR_NUMBER_, 'vectorA', var_numbers(2),
3             NULL_, status);
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
DO rec_number = 1, 100
  .
  /* read values from input file */
  .
  CALL CDFlib (SELECT_, zVARs_RECNUMBER_, rec_number,
  1            PUT_, zVARs_RECDATA_, 3L, var_numbers, buffer,
  2            NULL_, status);
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END DO
.
.
```

## 5.9 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 C explains each status code. When the status code returned from a CDF function is tested, 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_STATUSTEXT_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.

## 5.10 CDF_EPOCH Values

Several subroutines exist that compute, break down, parse, and encode CDF_EPOCH values. These subroutines may be called by applications using the CDF_EPOCH data type and are included in the CDF library. Section 2.2.9 describes EPOCH values.

### 5.10.1 compute_EPOCH

compute_EPOCH calculates a CDF_EPOCH value given the individual components. If an illegal component is detected, the value returned will be -1.0.

```
SUBROUTINE compute_EPOCH (year, month, day, hour, minute, second, msec,
.                         EPOCH)
INTEGER*4 year              ! In  -- Year (AD, e.g., 1994).
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.
```

### 5.10.2 EPOCH_breakdown

EPOCH_breakdown breaks down a CDF_EPOCH value into the individual components.

```
SUBROUTINE EPOCH_breakdown (EPOCH, year, month, day, hour, minute, second,
.                           msec)

REAL*8    EPOCH           ! In  -- The CDF_EPOCH value.
INTEGER*4 year            ! Out -- Year (AD, e.g., 1994).
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).
```

### 5.10.3 parse_EPOCH

parse_EPOCH parses a standard date/time character string and returns a CDF_EPOCH value. The format of the string is defined in Section 2.2.9 (and is also the format produced by the encode_EPOCH function). If an illegal field is detected in the string, the value returned will be -1.0.

```
SUBROUTINE parse_EPOCH (ep_string, EPOCH)
CHARACTER ep_string*(EPOCH_STRING_LEN)    ! In  -- The standard date/time
                                          !        character string.
REAL*8    EPOCH                           ! Out -- CDF_EPOCH value.
```

EPOCH_STRING_LEN is defined in cdf.inc.

### 5.10.4 encode_EPOCH

encode_EPOCH encodes a CDF_EPOCH value into a standard date/time character string. The format of the string is defined in Section 2.2.9 (and is also the format expected by the parse_EPOCH function).

```
SUBROUTINE encode_EPOCH (EPOCH, ep_string)
REAL*8    EPOCH                           ! In  -- CDF_EPOCH value.
CHARACTER ep_string*(EPOCH_STRING_LEN)    ! Out -- The standard date/time
                                          !        character string.
```

EPOCH_STRING_LEN is defined in cdf.inc.

### 5.10.5 encode_EPOCH1

encode_EPOCH1 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmdd.ttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and ttttttt is the fraction of the day (e.g., 5000000 is 12 o'clock noon).

```
SUBROUTINE encode_EPOCH1 (EPOCH, ep_string)
REAL*8    EPOCH                           ! In  -- CDF_EPOCH value.
CHARACTER ep_string*(EPOCH1_STRING_LEN)   ! Out -- The alternate date/time
                                          !        character string.
```

EPOCH1_STRING_LEN is defined in cdf.inc.

### 5.10.6 encode_EPOCH2

encode_EPOCH2 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmddhhmmss where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), and ss is the second (0-59).

```
SUBROUTINE encode_EPOCH2 (EPOCH, ep_string)
REAL*8    EPOCH                          ! In  -- CDF_EPOCH value.
CHARACTER ep_string*(EPOCH2_STRING_LEN)  ! Out -- The alternate date/time
                                         !        character string.
```

`EPOCH2_STRING_LEN` is defined in `cdf.inc`.

# Appendix A

# Skeleton Table Format

## A.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. Lines are limited to 132 characters.
2. Keywords for the header section, gAttributes section, vAttributes section, rVariables section, and end section must always be specified (in that order). The zVariables section is optional — its keyword may be omitted.
3. 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.
4. Attribute and variable names must be delimited. Any character not in the name may be used as the delimiter with the following exceptions:
   (a) Do not use an exclamation point (`!`) to delimit an attribute or variable name.
   (b) Do not use a period (`.`) to delimit an attribute name in the variables section.
   (c) Do not use a left square bracket (`[`) to delimit a variable name.
5. When specifying a character string attribute entry value, do not use a hyphen (`-`) to delimit the string or strings (if the string is split across one or more lines).
6. All items are referenced from one (1). These include gAttribute gEntry numbers and NRV variable index values.

In the descriptions that follow, optional fields are shown in brackets ([...]).

## A.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
!     ---------   ------------ ------------   -------     ----    -----------
   <rVars>/<zVars>   <gAttrs>    <vAttrs>   <n-recs>/z  <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 the `CDFskeleton` program, the following encodings are valid: `HOST`, `NETWORK`, `VAX`, `ALPHAVMSd`, `ALPHAVMSg`, `SUN`, `SGi`, `DECSTATION`, `ALPHAOSF1`, `IBMRS`, `HP`, `IBMPC`, and `NeXT`. 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 2.2.7.

`<variable-majority>` The variable majority of the CDF. This may be either `ROW` or `COLUMN`. Variable majority is described in Section 2.3.11.

`<cdf-format>` The format of the CDF. This may be either `SINGLE` or `MULTI`. CDF formats are described in Section 2.2.6. 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 create a multi-file CDF by default. `SkeletonTable` (in CDF V2.1 and beyond) always generates this line regardless of the version of the CDF being read.

`<rVars>` The number of rVariables in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table,

this value is ignored (but a place holder is necessary). The number of rVariables created is determined by the number of rVariable definitions in the rVariable definitions section.

`<zVars>` The number of zVariables in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored (but a place holder is necessary). The number of zVariables created is determined by the number of zVariable definitions in the zVariable definitions section.

`<gAttrs>` The number of gAttributes in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored (but a place holder is necessary). The number of gAttributes created is determined by the number of definitions in the gAttributes section.

`<vAttrs>` The number of vAttributes in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored (but a place holder is necessary). The number of vAttributes created is determined by the number of definitions in the vAttributes section.

`<n-recs>` The (maximum) number of rVariable records in the CDF. `SkeletonTable` always places the correct number here. However, when `CDFskeleton` reads a skeleton table, this value is ignored (but a place holder is necessary). The number of 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 2.3.9.

`<n-dims>` The number of dimensions for the rVariables in the CDF.

`<dim-sizes>` The dimension sizes for the rVariables in the CDF — one value per dimension. If the rVariables have zero (0) dimensions, this field would be left blank.

An example header section for a CDF with 2-dimensional rVariables follows:

```
#header

                CDF NAME: sample2
           DATA ENCODING: NETWORK
                MAJORITY: ROW
                  FORMAT: SINGLE

! Variables G.Attributes V.Attributes Records  Dims   Sizes
! --------- ------------ ------------ -------  ----  -------
    14/0          18             4        1/z      2   180 360
```

If the rVariables 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
! --------- ------------ ------------ -------  ----  -----
    14/0          18             4        1/z      0
```

## A.3 gAttributes Section

The gAttributes section contains the definition of each gAttribute as well as any gEntries for those gAttributes. The format of the gAttributes 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 gAttribute definition.

Zero or more gAttribute definitions are allowed. (There is no limit on the number of attributes that a CDF may have.) The format of each gAttribute 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 gAttribute. The name must be delimited with a character not appearing in the name itself (e.g., `"TITLE"` or `'History'`). The delimiting characters are not part of the gAttribute name in the CDF.

`<entry-n>` The gEntry number. Zero or more gEntries may be specified for a gAttribute, and there are no restrictions on the gEntry numbers that may be used (except that they must be greater than zero).

`<data-type>` The data type for the gEntry. 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 gEntry specified. If omitted, the data type of the previous gEntry is assumed.

`<value>` The value(s) for the gEntry. A period (`.`) follows the value(s) of the last gEntry for a gAttribute.

**Attribute Entry Values**

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 (e.g., `{ "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 (e.g., `{ 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 2.2.9. A `CDF_EPOCH` value may not be split across two lines.

Several example gAttribute 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"               1:    CDF_EPOCH    { 04-Jul-1976 12:00:00.000,
                                               31-Oct-1976 00:00:00.000 }
                        2:                 { 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.4 vAttributes Section

The vAttributes section contains the names of the vAttributes in the CDF. Any rEntries or zEntries for these vAttributes are defined in the rVariables/zVariables sections (following the definition of the corresponding variable). The format of the vAttributes section is as follows:

```
#VARIABLEattributes

[<attribute-name>
 <attribute-name>
 <attribute-name>
        .
        .
        .
 <attribute-name>]
```

Where `<attribute-name>` is a vAttribute name delimited with a character not appearing in the name itself (e.g., `"VALIDMIN"` or `'Units'`). The delimiting characters are not part of the vAttribute name in the CDF. There may be zero or more vAttribute names. (There is no limit on the number of attributes that a CDF may have.)

An example vAttributes section follows:

```
#VARIABLEattributes

  "FIELDNAM"
  "VALIDMIN"
  "Units"
```

## A.5 rVariables Section

The rVariables section contains the definition of each rVariable in the CDF, the values for any vAttribute rEntries associated with each rVariable, and (optionally) data values for those rVariables that are non-record-variant (NRV). The format of the rVariables section is as follows:

```
#variables

[<variable-definition>
 <variable-definition>
 <variable-definition>
           .
           .
           .
 <variable-definition>]
```

Where `<variable-definition>` is an rVariable definition. The format of each rVariable 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 rVariable. The name must be delimited with a character

not appearing in the name itself (e.g., `"EPOCH"` or `'Temperature'`). The delimiting characters are not part of the rVariable name in the CDF.

`<var-data-type>` The data type for the rVariable. 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 rVariable. 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 rVariable. For each dimension 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 rVariables have zero dimensions, this field would be left blank.

`<attr-name>` The name of the vAttribute for which to specify an rEntry for this rVariable. The vAttribute must have been specified in the vAttributes section. The name must be delimited with a character not appearing in the name itself (e.g., `"SCALEMAX"` or `'range'`). The delimiting characters are not part of the vAttribute name in the CDF.

`<entry-data-type>` The data type for the vAttribute rEntry. 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`.

`<entry-value>` The value(s) for the vAttribute rEntry. The format of attribute entry values is described in Section A.3.

**NOTE:** The last rEntry MUST be followed by a period (`.`). If no rEntries are specified for an rVariable, the period must still be present.

`<indices>` The indices of an NRV value. The indices are enclosed in brackets and separated by commas (e.g., `[ 23, 1 ]` or `[ 1, 80 ]`). If the rVariables have 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 2.2.9.

NRV values may only be specified for NRV variables (described in Section 2.3.9).

The vAttribute rEntries are optional. If omitted, the terminating period is still required. The NRV values are also optional. (NRV variables are described in Section 2.3.9.)

A sample rVariable definition for a CDF with 2-dimensional rVariables follows:

```
! 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 definitions for a CDF with 0-dimensional rVariables follow:

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

  "Latitude"        CDF_REAL4       1          F

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

    "VALIDMIN"      CDF_REAL4    { -90.0 }
    "VALIDMAX"      CDF_REAL4    { 90.0 }.

    [ ] = -12.3
```

## A.6 zVariables Section

The optional zVariables section contains the definition of each zVariable in the CDF, the values for any vAttribute zEntries associated with each zVariable, and (optionally) data values for those zVariables that are non-record-variant (NRV). The format of the zVariables section is as follows:

```
#zVariables

[<variable-definition>
 <variable-definition>
 <variable-definition>
           .
           .
           .
```

```
 <variable-definition>]
```

Where <variable-definition> is a zVariable definition. The format of each zVariable definition is as follows:

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

  <var-name> <var-data-type> <n-elems> <dims> <sizes> <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 zVariable. The name must be delimited with a character not appearing in the name itself (e.g., "EPOCH" or 'Temperature'). The delimiting characters are not part of the zVariable name in the CDF. |
| <var-data-type> | The data type for the zVariable. 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). |
| <dims> | The number of dimensions for the zVariable. |
| <sizes> | The dimension sizes — one value per dimension. If the zVariable has zero (0) dimensions, this field would be left blank. |

`<rec-vary>` The record variance of the zVariable. 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 zVariable. For each dimension 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 zVariable has zero dimensions, this field would be left blank.

`<attr-name>` The name of the vAttribute for which to specify a zEntry for this zVariable. The vAttribute must have been specified in the vAttributes section. The name must be delimited with a character not appearing in the name itself (e.g., `"SCALEMAX"` or `'range'`). The delimiting characters are not part of the vAttribute name in the CDF.

`<entry-data-type>` The data type for the vAttribute zEntry. 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`.

`<entry-value>` The value(s) for the vAttribute zEntry. The format of attribute entry values is described in Section A.3.

**NOTE:** The last zEntry MUST be followed by a period (`.`). If no zEntries are specified for a zVariable, the period must still be present.

`<indices>` The indices of an NRV value. The indices are enclosed in brackets and separated by commas (e.g., `[ 23, 1 ]` or `[ 1, 80 ]`). If the zVariable 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 2.2.9.

NRV values may only be specified for NRV variables (described in Section 2.3.9).

The vAttribute zEntries are optional. If omitted, the terminating period is still required. The NRV values are also optional (NRV variables are described in Section 2.3.9).

Several sample zVariable definitions follow:

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

  "Instrument"    CDF_CHAR      10       0              F
```

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

    "FIELDNAM"    CDF_CHAR     { "Measuring instrument" }.

  [ ] = { "Gonkulator" }


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

  "WIND VELOCITY"  CDF_REAL4      1       3   360 180 10      T       T T T

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

    "FIELDNAM"    CDF_CHAR     { "Wind velocity." }
    "VALIDMIN"    CDF_REAL4    { 0.0 }
    "VALIDMAX"    CDF_REAL4    { 300.0 }
    "UNITS   "    CDF_CHAR     { "Knots" }
    "FORMAT  "    CDF_CHAR     { "F9.1    " }.
```

## A.7 End Section

This section simply consists of the keyword `#end`. This section is required.

## A.8 Example Skeleton Table

An example skeleton table containing rVariables and zVariables 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      " } .


#zVariables

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

  "BIAS"          CDF_INT4       1       0              T

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

    "FIELDNAM"    CDF_CHAR     { "Correction bias for temperature" }
    "VALIDMIN"    CDF_INT4     { -5 }
    "VALIDMAX"    CDF_INT4     { 5 }
    "UNITS   "    CDF_CHAR     { "deg C               " }
    "FORMAT  "    CDF_CHAR     { "I2      " } .


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

  "Coefficients"  CDF_REAL4      1       1     3        F         T

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

    "FIELDNAM"    CDF_CHAR     { "Temperature model coefficients." }
    "FORMAT  "    CDF_CHAR     { "F9.1    " } .

  [ 1 ] = -0.0254
  [ 2 ] = 14.2338
  [ 3 ] = -9.9444
```

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

  "TMP-model"     CDF_REAL4      1       2  360 180     T        T T

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

    "FIELDNAM"    CDF_CHAR     { "Temperature model." }
    "VALIDMIN"    CDF_REAL4    { -20.0 }
    "VALIDMAX"    CDF_REAL4    { 50.0 }
    "SCALEMIN"    CDF_REAL4    { 0.0 }
    "SCALEMAX"    CDF_REAL4    { 30.0 }
    "UNITS   "    CDF_CHAR     { "deg C               " }
    "FORMAT  "    CDF_CHAR     { "F9.6    " } .


#end
```

# Appendix B

# CDF's IDL Support

## B.1 Introduction

The CDF distribution includes a set of IDL functions/procedures that allow access to the CDF library (hereafter referred to as the CDF/IDL interface). The functions/procedures are used in the same manner as the Standard and Internal Interfaces are used in programs written in C. CDFs can be both read and written/modified.

**NOTE:** There is, however, one difference when using the Internal Interface (`CDFlib`). The current CDF must be selected each time `CDFlib` is called. This is because of how IDL does the dynamic mapping to the CDF library at runtime. Failure to select the current CDF may result in IDL crashing.

Research Systems, Inc. (the developers of IDL) have also implemented an interface to CDF that is part of the IDL product. It differs from the interface provided with the CDF distribution in that it is intended more for the non-programmer (and is functionally similar to other interfaces they provide). Which interface you choose to use will depend on your personal preferences.

CDF's interface relies on IDL's support of dynamic linking. IDL in turn relies on the operating system being used to support dynamic linking in a reasonable way. Currently, only VAX/VMS, DEC Alpha/OpenVMS, Sun/SunOS, Sun/SOLARIS, HP 9000/HP-UX, and IBM RS6000/AIX support dynamic linking in a way that makes it possible to use CDF's interface to IDL. Because IDL's CDF interface is built into the IDL executable, it does not rely on dynamic linking and is therefore available on every machine on which IDL is supported.

## B.2 Using CDF's IDL Interface

This section assumes that you are using the "definitions" file provided with the CDF distribution (and that it has been properly configured and named by your system manager). The "definitions" file is a command file named `DEFINITIONS.COM` on VMS systems and a script file named `definitions.<shell>` on UNIX systems (where `<shell>` is the shell being used). The "definitions" file defines the logical name `IDL_STARTUP` on VMS systems and the environment variable `IDL_STARTUP` on UNIX systems to the full directory path of an

IDL startup file provided with the CDF distribution. When IDL is started, this startup file is automatically executed (because `IDL_STARTUP` is defined). The startup file performs the following functions...

1. Makes known to IDL the functions/procedures that are used to call the CDF library (and perform other utility operations).
2. Sets IDL's internal path, which is used when searching for a batch file to be executed.
3. Executes a batch file (`cdf.pro`) that creates a number of local variables necessary for calling CDF functions/procedures and then interpreting the results.

A list of messages should be displayed indicating the CDF functions/procedures that have been compiled.

If you are already using your own startup file (pointed to by your definition of `IDL_STARTUP`), you should remove the definition of `IDL_STARTUP` from the CDF "definitions" file being used and do one of the following...

1. Add the contents of the CDF/IDL startup file to your own startup file.
2. Have your startup file execute the CDF/IDL startup file.

The definition of `IDL_STARTUP` in the CDF "definitions" file being used points to the location of the CDF/IDL startup file.

## B.3 CDF "Include" Files

A number of parameters (macro constants) are necessary when calling functions in the CDF library from a C program. These parameters are defined in `cdf.h`. When calling the CDF functions/procedures in IDL, local variables are used to accomplish the same thing. An IDL batch[1] file, `cdf.pro`, is provided that creates/initializes the necessary local variables. `cdf.pro` is executed automatically by the startup file when you run IDL. All necessary CDF parameters will be available including the status codes commonly used (`CDF_OK` and `CDF_WARN`). Most users will not need the other status codes since the `CDFerror` function can be used to determine the explanation of a status code. If you do require the other status codes, execute the batch file `cdf1.pro` (the IDL internal path will know where to find it). If you are using the Internal Interface (`CDFlib`), you will also need the parameters (local variables) defined in `cdf2.pro`.

IDL limits the number of local variables that may be defined in any one function/procedure. This limit might be reached if you have to include one or more of the CDF "include" files. If that happens you will have to use one or more of the "mapping" functions provided with the CDF/IDL interface. The use of these "mapping" functions eliminates the need to include the corresponding 'include' file.

| Include File | Mapping Function | Purpose |
|---|---|---|
| cdf.pro | mcp.pro | Map CDF parameters |
| cdf1.pro | msc.pro | Map status codes |
| cdf2.pro | mii.pro | Map Internal Interface |

Each function returns the longword value that corresponds to the parameter passed in as a character string. For example, the status code value for `NO_SUCH_CDF` would be retrieved as follows...

[1] These are also known as include files.

```
IDL> status = CDFopen ('test1', id)
IDL> if (status eq MSC('NO_SUCH_CDF')) ...
```

In this case the batch file `cdf1.pro` would not have to be included.

## B.4 On-Line Help

IDL provides an on-line help facility to which help for user written procedures/functions can be added. A specially formatted file is provided with the CDF distribution that contains on-line help for the procedures/functions in the CDF/IDL interface. This file is named `cdf24cdfif.help` and is located in the "help" directory of the CDF distribution. Your system administrator must install this file in the "help" directory of your IDL distribution in order for the on-line help to be available.[2]

## B.5 Available Functions

The CDF functions/procedures available at the IDL command line exactly mirror the Standard and Internal Interface functions available for a program written in C.[3] Chapter 4 describes the functionality of each in detail. The syntax required when calling each function is as follows...

```
IDL> status = CDFlib (op1, ...)
IDL> status = CDFcreate (CDFpath, numDims, dimSizes, encoding, majority, $
IDL>                     id)
IDL> status = CDFopen (CDFpath, id)
IDL> status = CDFdoc (id, version, release, copyright)
IDL> status = CDFinquire (id, numDims, dimSizes, encoding, majority, $
IDL>                      maxRec, numVars, numAttrs)
IDL> status = CDFclose (id)
IDL> status = CDFdelete (id)
IDL> status = CDFerror (status, text)
IDL> status = CDFattrCreate (id, attrName, attrScope, attrNum)
IDL> status = CDFattrRename (id, attrNum, attrName)
IDL> status = CDFattrInquire (id, attrNum, attrName, attrScope, maxEntry)
IDL> status = CDFattrEntryInquire (id, attrNum, entryNum, dataType, $
IDL>                               numElems)
IDL> status = CDFattrPut (id, attrNum, entryNum, dataType, numElems, value)
IDL> status = CDFattrGet (id, attrNum, entryNum, value)
IDL> status = CDFattrCorrectScope (id, attrNum, attrScope)
IDL> status = CDFvarCreate (id, varName, dataType, numElems, recVary, $
IDL>                        dimVarys, varNum)
```

[2] A future release of IDL may allow to be defined either a logical name, environment variable, and/or IDL system variable that would point to the location(s) of on-line help files.

[3] With the exception of the multiple variable access operations available via the Internal Interface (`<GET_,rVARs_RECDATA_>`, `<PUT_,rVARs_RECDATA_>`, `<GET_,zVARs_RECDATA_>`, and `<PUT_,zVARs_RECDATA_>`). Because these operations deal with variables having different data types, the use of structures to store the values would be the logical method to use. Unfortunately, IDL does not currently support the manipulation of structures from within a C application (which is how CDF's IDL interface is implemented). Support of these operations will hopefully be added in the future.

```
IDL> status = CDFvarRename (id, varNum, varName)
IDL> status = CDFvarInquire (id, varNum, varName, dataType, numElems, $
IDL>                         recVary, dimVarys)
IDL> status = CDFvarPut (id, varNum, recNum, indices, value)
IDL> status = CDFvarGet (id, varNum, recNum, indices, value)
IDL> status = CDFvarHyperPut (id, varNum, recStart, recCount, recInterval, $
IDL>                          indices, counts, intervals, buffer)
IDL> status = CDFvarHyperGet (id, varNum, recStart, recCount, recInterval, $
IDL>                          indices, counts, intervals, buffer)
IDL> status = CDFvarClose (id, varNum)
IDL> attrNum = CDFattrNum (id, attrName)
IDL> varNum = CDFvarNum (id, varName)
IDL> status = CDFlib (op1, ...)
IDL> epoch = computeEPOCH (year, month, day, hour, minute, second, $
IDL>                       millisecond)
IDL> EPOCHbreakdown, epoch, year, month, day, hour, minute, second, $
IDL>                 millisecond
IDL> epoch = parseEPOCH (epochString)
IDL> encodeEPOCH, epoch, epochString
IDL> encodeEPOCH1, epoch, epochString
IDL> encodeEPOCH2, epoch, epochString
```

The IDL data types to use for integer arguments are always longwords or arrays of longwords. Attribute entries and variable values should have the data types corresponding their CDF data types. CDF pathnames and variable/attribute names are IDL character strings. `CDFid` and `CDFstatus` are also longwords. Chapter 4 may be used as a guide (especially for `CDFlib`). Arguments passed to the CDF functions/procedures must obviously be created and initialized before the call. Arrays are necessary for dimension sizes, variances, etc. Arguments returned by the CDF functions will be created to be of the proper data type and size (they do not have to be created before calling the CDF function).

**NOTE:** There is one difference between how the Internal Interface (`CDFlib`) is used in IDL and in a C/Fortran application. In IDL the current CDF must be selected each time `CDFlib` is called. Failure to select the current CDF may result in IDL crashing.

Also provided is a function, `row_to_col`, that may be used to change the majority of an array (or arrays) of values from row-major to column-major. This function is necessary because CDF can store variable values in either row or column major order. The syntax of `row_to_col` is as follows...

```
IDL> status = row_to_col (inArray, outArray, numDims, dimSizes, numBytes, $
                          arrayCount
```

where...

| | |
|---|---|
| `inArray` | The array (or arrays) of values to convert — row major ordering. Any valid IDL data type is allowed. |
| `outArray` | The converted array (or arrays) of values — column major ordering. This array will be created by `row_to_col`. |
| `numDims` | The number of dimensions in the array (or arrays). This value must be a longword. |

| | |
|---|---|
| `dimSizes` | The size of each dimension. This must be an array of longwords — one longword per dimension. |
| `numBytes` | The number of bytes in each value. This value must be a longword. |
| `arrayCount` | The number of arrays to convert. The order of the arrays will not be affected. This value must be a longword. |
| `status` | Completion status code. This value will be a longword. The possible values are defined in `cdf1.pro`. |

Multiple arrays (records) of CDF variable values should always be stored with the last dimension of the IDL array corresponding to the record number. For example, assume a CDF variable with two dimensions whose sizes are `[180,360]`. If `50` records are to be stored, an IDL array with three dimensions whose sizes are `[180,360,50]` should be used. If the CDF variables have row major ordering, `row_to_col` would be called with `numDims` set to `2`, `dimSizes` set to `[180,360]`, and `arrayCount` set to `50`.

## B.6 Example IDL Session

The following IDL session will open a CDF (unsuccessfully at first — typo), inquire about a variable named `LATITUDE`, and then close the CDF.

```
.
.
IDL> status=CDFopen('yrdy3',id)
IDL> status=CDFerror(status,text)
IDL> print,text
NO_SUCH_CDF: The specified CDF does not exist.
IDL> status=CDFopen('test2',id)
IDL> status=CDFerror(status,text)
IDL> print,text
CDF_OK: Function completed successfully.
IDL> status=CDFvarInquire(id,CDFvarNum(id,'LATITUDE'),varName,dataType,$
IDL> numElems,recVary,dimVarys)
IDL> status=CDFerror(status,text)
IDL> print,text
CDF_OK: Function completed successfully.
IDL> print,dataType,numElems
          21           1
IDL> status=CDFclose(id)
IDL> status=CDFerror(status,text)
IDL> print,text
CDF_OK: Function completed successfully.
.
.
```

As you can see, checking the return status from each call to the CDF library can be fairly tedious. A procedure such as the following could be used to ease status code checking...

```
pro checkstatus, status
@cdf
if (status lt CDF_OK) then begin
  statusT = CDFerror (status, text)
  print, text
endif
return
end
```

The sample IDL session could then be as follows...

```
.
.
IDL> status=CDFopen('yrdy3',id)
IDL> checkstatus,status
NO_SUCH_CDF: The specified CDF does not exist.
IDL> checkstatus,CDFopen('test2',id)
CDF_OK: Function completed successfully.
IDL> status=CDFvarInquire(id,CDFvarNum(id,'LATITUDE'),varName,dataType,$
IDL> numElems,recVary,dimVarys)
IDL> checkstatus,status
CDF_OK: Function completed successfully.
IDL> print,dataType,numElems
          21           1
IDL> checkstatus,CDFclose(id)
CDF_OK: Function completed successfully.
.
.
```

As you can see, a call to a CDF function can be embedded in a call to `checkstatus`.

## B.7 Example IDL Function

The following example function is also available in the "samples" subdirectory of the CDF distribution in a file named `idlload1.pro`.

```
;
;
; This function will load all data for a specified rVariable from a CDF into
; an IDL variable.  All of the IDL/CDF interface routines may be entered at
; the IDL command line but for demonstration purposes they will be used in
; an IDL function.
;
; Argument descriptions:                          data type
; CDF_name      input CDF name                    string
; var_name      variable name in CDF              string
```

```
; data          array to be returned to IDL         (see creation below)
;
; Executing
; status = get_var('SAMPLE_CDF','VARNAME',data)
;
function get_var, CDF_name, var_name, data
;
; include all CDF definitions
;
@cdf
;
; Open the CDF, note that CDF_id will be returned as the proper data type
;
status = CDFopen(CDF_name,CDF_id)
;
; check the status of the open, if a error is detected print a error and
; return
;
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
endif
;
; Inquire about the structure of the CDF
;
status = CDFinquire(CDF_id,num_dims,dim_sizes,encoding,majority,$
                    max_record_num,num_vars,num_attrs)
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
endif
;
; Get the number in the CDF of the variable
;
var_num = CDFvarNum(CDF_id, var_name)
if(var_num lt CDF_OK)then begin
   status = CDFerror(var_num, error_mes)
   print,error_mes
   return,status
endif
;
; Inquire about the variable
;
status = CDFvarInquire(CDF_id, var_num, out_var_name, data_type, $
                       num_bytes, record_variance, dim_variances)
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
```

```
endif
;
; We will retrieve some of the metadata from this variable,
; since CDF attributes can possibly be arrays of attributes we
; will need to test for this and create a output array if necessary
; scalers attributes will be handled by the software.
; FIELDNAM is a CDF standard attribute and is a description of the variable
;
attr_num = CDFattrNum(CDF_id, "FIELDNAM")
if(attr_num < CDF_OK)then begin
   status = CDFerror(attr_num, error_mes)
   print,error_mes
   return,status
endif
status = CDFattrInquire(CDF_id, attr_num, out_attr_name, $
                        attr_scope, max_entry)
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
endif
;
; Need the data type and size of this attribute
;
status = CDFattrEntryInquire(CDF_id, attr_num, var_num, $
                             attr_data_type, num_elements)
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
endif
;
; check to see if a array needs to be created to store the value[s].
;
if((num_elements gt 1) and $
   (data_type ne CDF_CHAR and data_type ne CDF_UCHAR))then begin
;
; This attribute contains a array of values this attribute we will
; use the idl routine make_array to this, we must first create a 1
; dimensional array with the proper number of elements and data type
; of the attribute.  The make_array function uses as input a size
; vector array, see IDL documentation for the size function to describe
; the contents of the array.
;
    size_array=lonarr(4)
    size_array(0) = 1
    size_array(1) = num_elements
;
; this function will map the CDF data type to a IDL data type (see below)
;
    convert_data_type,attr_data_type,idl_data_type
```

```
    size_array(2) = idl_data_type
;
; We are now ready to make the attribute value array
;
    attr_value = make_array(size=size_array)
endif
status = CDFattrGet(CDF_id, attr_num, var_num, attr_value)
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
endif
print,'Loading variable ',var_name,': ',attr_value
;
; We will now get ready to load all data for this variable,
; use the CDF structure to initialize all parameters to perform this
; operation
;
recStart = 0l
recCount = max_record_num+1
recInterval=1l
indices = lonarr(CDF_MAX_DIMS)
counts = lonarr(CDF_MAX_DIMS)
intervals = lonarr(CDF_MAX_DIMS)
for dim_num=0, CDF_MAX_DIMS-1 do begin
    indices(dim_num) = 0;
    counts(dim_num) = dim_sizes(dim_num)
    intervals(dim_num) = 1
endfor
;
; Before retrieving the data we must create a array large enough to hold all
; the data for this variable and it must be the proper data type, we will
; again use the idl function make_array to this, as above first create the
; size vector with proper dimensions of the IDL array.
;
size_array=lonarr(num_dims+4)
size_array(0) = num_dims + 1
for i = 1, num_dims do size_array(i) = dim_sizes(i-1)
size_array(num_dims+1) = recCount
;
; this function will map the CDF data type to a IDL data type (see below)
;
convert_data_type,data_type,idl_data_type
;
; Now load the data type into the size array
;
size_array(num_dims+2) = idl_data_type
;
; We are now ready to make the data input array, the array returned will
; be of the same data type and size.
;
```

```
inbuf = make_array(size=size_array)
;
; Retrieve the data
;
status = CDFvarHyperGet(CDF_id,var_num, recStart,recCount,recInterval, $
                        indices,counts,intervals,inbuf)
if(status lt CDF_OK)then begin
   st = CDFerror(status, error_mes)
   print,error_mes
   return,status
endif
;
; IDL needs to have the CDF in column majority in order to plot properly,
; we will supply a routine that will convert from ROW to COLUMN major.
; Converting character data is not supported in this release
;
if((majority eq ROW_MAJOR) and $
   (data_type ne CDF_CHAR and data_type ne CDF_UCHAR))then begin
   print,'Converting from ROW to COLUMN major'
;
; Need to know the number of bytes of this data type (e.g. 1,2,4,8)
;
   nbytes,data_type, num_bytes_dt
;
; Now convert
;
   status = row_to_col (inbuf, data, num_dims, dim_sizes, num_bytes_dt, $
                        recCount)
endif else begin
;
; No conversion necessary, load input buffer into return data buffer
;
   data = inbuf
endelse
status=CDFclose(CDF_id)
return,CDF_OK
end
pro convert_data_type,data_type,idl_data_type
@cdf
Case 1 of
 (data_type eq CDF_INT1)   : idl_data_type = 1l
 (data_type eq CDF_INT2)   : idl_data_type = 2l
 (data_type eq CDF_INT4)   : idl_data_type = 3l
 (data_type eq CDF_UINT1)  : idl_data_type = 1l
 (data_type eq CDF_UINT2)  : idl_data_type = 2l
 (data_type eq CDF_UINT4)  : idl_data_type = 3l
 (data_type eq CDF_REAL4)  : idl_data_type = 4l
 (data_type eq CDF_REAL8)  : idl_data_type = 5l
 (data_type eq CDF_EPOCH)  : idl_data_type = 5l
 (data_type eq CDF_BYTE)   : idl_data_type = 1l
 (data_type eq CDF_FLOAT)  : idl_data_type = 4l
```

```
 (data_type eq CDF_DOUBLE) : idl_data_type = 5l
 (data_type eq CDF_CHAR)   : idl_data_type = 7l
 (data_type eq CDF_UCHAR)  : idl_data_type = 7l
endcase
return
end
pro nbytes,data_type,num_bytes
@cdf
num_bytes = 0l
Case 1 of
 (data_type eq CDF_INT1)   : num_bytes = 1l
 (data_type eq CDF_INT2)   : num_bytes = 2l
 (data_type eq CDF_INT4)   : num_bytes = 4l
 (data_type eq CDF_UINT1)  : num_bytes = 1l
 (data_type eq CDF_UINT2)  : num_bytes = 2l
 (data_type eq CDF_UINT4)  : num_bytes = 4l
 (data_type eq CDF_REAL4)  : num_bytes = 4l
 (data_type eq CDF_REAL8)  : num_bytes = 8l
 (data_type eq CDF_EPOCH)  : num_bytes = 8l
 (data_type eq CDF_BYTE)   : num_bytes = 1l
 (data_type eq CDF_FLOAT)  : num_bytes = 4l
 (data_type eq CDF_DOUBLE) : num_bytes = 8l
 (data_type eq CDF_CHAR)   : num_bytes = 1l
 (data_type eq CDF_UCHAR)  : num_bytes = 1l
endcase
return
end
```

# Appendix C

# CDF Status Codes

## C.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 (constants) for each of the status codes (and for any other constants referred to in the explanations). The CDF library Standard Interface 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. The Internal Interface can also be used to inquire explanation text.

There are three classes of status codes: informational, warning, and error. The purpose of each is as follows:

| | |
|---|---|
| Informational | Indicates success but provides some additional information that may be of interest to an application. |
| Warning | Indicates that the function completed but possibly not as expected. |
| Error | Indicates that a fatal error occurred and the function aborted. |

Status codes fall into classes as follows:

Error codes < `CDF_WARN` < Warning codes < `CDF_OK` < Informational codes

`CDF_OK` indicates an unqualified success (it should be the most commonly returned status code). `CDF_WARN` is simply used to distinguish between warning and error status codes.

## C.2 Status Codes and Messages

The following list contains an explanation for each possible status code. Whether a particular status code is considered informational, a warning, or an error is also indicated.

| | |
|---|---|
| ALPHAOSF1_ENCODING_UNSUPPORTED | DEC Alpha/OSF1 encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| ALPHAVMSd_ENCODING_UNSUPPORTED | DEC Alpha/OpenVMS/D_FLOAT encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| ALPHAVMSg_ENCODING_UNSUPPORTED | DEC Alpha/OpenVMS/G_FLOAT encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| ATTR_EXISTS | Named attribute already exists — cannot create or rename. Each attribute in a CDF must have a unique name. Note that trailing blanks are ignored by the CDF library when comparing attribute names. [Error] |
| ATTR_NAME_TRUNC | Attribute name truncated to CDF_ATTR_NAME_LEN characters. The attribute was created but with a truncated name. [Warning] |
| BAD_ARGUMENT | An illegal/undefined argument was passed. Check that all arguments are properly declared and initialized. [Error] |
| BAD_ATTR_NAME | Illegal attribute name specified. Attribute names must contain at least one character, and each character must be printable. [Error] |
| BAD_ATTR_NUM | Illegal attribute number specified. Attribute numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error] |
| BAD_CACHESIZE | An illegal number of cache buffers was specified. The value must be at least one (1). [Error] |
| BAD_CDF_ID | CDF identifier is unknown or invalid. The CDF identifier specified is not for a currently open CDF. [Error] |
| BAD_CDF_NAME | Illegal CDF name specified. CDF names must contain at least one character, and each character must be printable. Trailing blanks are allowed but will be ignored. [Error] |
| BAD_CDFSTATUS | Unknown CDF status code received. The status code specified is not used by the CDF library. [Error] |
| BAD_DATA_TYPE | Unknown data type specified. The CDF data types are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error] |
| BAD_DECODING | Unknown decoding specified. The CDF decodings are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error] |
| BAD_DIM_COUNT | Illegal dimension count specified. A dimension count must be at least one (1) and not greater than the size of the dimension. [Error] |

| | |
|---|---|
| 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. [Error] |
| BAD_DIM_INTERVAL | Illegal dimension interval specified. Dimension intervals must be at least one (1). [Error] |
| BAD_DIM_SIZE | Illegal dimension size specified. A dimension size must be at least one (1). [Error] |
| BAD_ENCODING | Unknown data encoding specified. The CDF encodings are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error] |
| BAD_ENTRY_NUM | Illegal attribute entry number specified. Entry numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. [Error] |
| BAD_EXTEND_RECS | An illegal number of extend records has been specified. The number of extend records must be at least zero (0). A value of zero indicates that the default should be used. Note that extend records are only significant for variables that are not NRV and are in single-file CDFs. [Error] |
| 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. [Error] |
| BAD_FORMAT | Unknown format specified. The CDF formats are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error] |
| 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). [Error] |
| BAD_MAJORITY | Unknown variable majority specified. The CDF variable majorities are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error] |
| BAD_MALLOC | Unable to allocate dynamic memory — system limit reached. Contact CDF User Support if this error occurs. [Error] |
| BAD_NEGtoPOSfp0_MODE | An illegal `-0.0` to `0.0` mode was specified. The `-0.0` to `0.0` modes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error] |
| 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. [Error] |
| 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). [Error]

`BAD_READONLY_MODE` Illegal read-only mode specified. The CDF read-only modes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_REC_COUNT` Illegal record count specified. A record count must be at least one (1). [Error]

`BAD_REC_INTERVAL` Illegal record interval specified. A record interval must be at least one (1). [Error]

`BAD_REC_NUM` Record number is out of range. Record numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. Note that a valid value must be specified regardless of the record variance. [Error]

`BAD_SCOPE` Unknown attribute scope specified. The attribute scopes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_VAR_NAME` Illegal variable name specified. Variable names must contain at least one character and each character must be printable. [Error]

`BAD_VAR_NUM` Illegal variable number specified. Variable numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]

`BAD_zMODE` Illegal zMode specified. The CDF zModes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`CANNOT_CHANGE` Because of dependencies on the value, it cannot be changed. Some possible causes of this error follow:

1. Changing a CDF's data encoding after a variable value (including a pad value) or an attribute entry has been written.
2. Changing a CDF's format after a variable has been created.
3. Changing a CDF's variable majority after a variable value (excluding a pad value) has been written.
4. Changing a variable's data specification after a value (including the pad value) has been written to that variable.
5. Changing a variable's record variance after a value (excluding the pad value) has been written to that variable.

6. Changing a variable's dimension variances after a value (excluding the pad value) has been written to that variable.
7. Writing initial records to a variable after a value (excluding the pad value) has been written to that variable.
8. Changing an attribute entry's data specification where the new specification is not equivalent to the old specification.

[Error]

`CDF_CLOSE_ERROR` Error detected while trying to close CDF. Check that sufficient disk space exists for the `.cdf` file and that it has not been corrupted. [Error]

`CDF_CREATE_ERROR` Cannot create CDF specified — error from file system. Make sure sure that sufficient privilege exists to create the `.cdf` file in the disk/directory location specified and that an open file quota has not already been reached. [Error]

`CDF_EXISTS` The CDF named already exists — cannot create it. The CDF library will not overwrite an existing CDF. [Error]

`CDF_INCOMPATIBLE` Certain operations on Version 2 CDFs are incompatible with the Obsolete Interface (VMS only). These deal mainly with attributes having entries whose data specifications differ. The incompatibilities are described in Section 2.1.1. [Error]

`CDF_NAME_TRUNC` CDF pathname truncated to `CDF_PATHNAME_LEN` characters. The CDF was created but with a truncated name. [Warning]

`CDF_OK` Function completed successfully.

`CDF_OPEN_ERROR` Cannot open CDF specified — error from file system. Check that the `.cdf` file is not corrupted and that sufficient privilege exists to open it. Also check that an open file quota has not already been reached. [Error]

`CDF_READ_ERROR` Failed to read CDF file — error from file system. Check that the `.cdf` file is not corrupted. [Error]

`CDF_WRITE_ERROR` Failed to write CDF file — error from file system. Check that the `.cdf` file is not corrupted. [Error]

`CDH_CLOSE_ERROR` An error occurred while closing the `.cdh` file (V1 CDF). [Error]

`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. [Error]

`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. [Error]

| | |
|---|---|
| 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. [Error] |
| DECSTATION_ENCODING_UNSUPPORTED | DECstation encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| END_OF_VAR | The sequential access current value is at the end of the variable. Reading beyond the end of the last physical value for a variable is not allowed (when performing sequential access). [Error] |
| HP_ENCODING_UNSUPPORTED | HP encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| IBM_PC_OVERFLOW | An operation involving a buffer greater than 64k bytes in size has been specified. [Error] |
| IBMPC_ENCODING_UNSUPPORTED | IBM PC encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| IBMRS_ENCODING_UNSUPPORTED | IBM RS encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| ILLEGAL_FOR_SCOPE | The operation is illegal for the attribute's scope. For example, only gEntries may be written for gAttributes — not rEntries or zEntries. [Error] |
| ILLEGAL_IN_zMODE | The attempted operation is illegal while in zMode. Most operations involving rVariables or rEntries will be illegal. [Error] |
| ILLEGAL_ON_V1_CDF | The specified operation is not allowed on Version 1 CDFs. Section 2.2.1 describes the restrictions when accessing Version 1 CDFs. [Error] |
| MULTI_FILE_FORMAT | The specified operation is not applicable to CDFs with the multi-file format. For example, it does not make sense to inquire indexing statistics for a variable in a multi-file CDF (indexing is only used in single-file CDFs). [Informational] |
| NEGATIVE_FP_ZERO | One or more of the values read are -0.0 (an illegal value on VAXes). [Warning] |
| NeXT_ENCODING_UNSUPPORTED | NeXT encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| NO_ATTR_SELECTED | An attribute has not yet been selected. First select the attribute on which to perform the operation. [Error] |
| NO_CDF_SELECTED | A CDF has not yet been selected. First select the CDF on which to perform the operation. [Error] |

| | |
|---|---|
| NO_DELETE_ACCESS | Deleting is not allowed (read-only access). Make sure that delete access is allowed on the CDF file(s). [Error] |
| NO_ENTRY_SELECTED | An attribute entry has not yet been selected. First select the entry number on which to perform the operation. [Error] |
| NO_MORE_ACCESS | Further access to the CDF is not allowed because of a severe error. If the CDF was being modified, closing the CDF may save some of the modifications made since the CDF was opened. [Error] |
| NO_PADVALUE_SPECIFIED | A pad value has not yet been specified. The default pad value is currently being used for the variable. The default pad value was returned. [Informational] |
| NO_STATUS_SELECTED | A CDF status code has not yet been selected. First select the status code on which to perform the operation. [Error] |
| NO_SUCH_ATTR | The named attribute was not found. Note that attribute names are case-sensitive. [Error] |
| NO_SUCH_CDF | The specified CDF does not exist. Check that the pathname specified is correct. [Error] |
| NO_SUCH_ENTRY | No such entry for specified attribute. [Error] |
| NO_SUCH_VAR | The named variable was not found. Note that variable names are case-sensitive. [Error] |
| NO_VAR_SELECTED | A variable has not yet been selected. First select the variable on which to perform the operation. [Error] |
| NO_VARS_IN_CDF | This CDF contains no rVariables. The operation performed is not applicable to a CDF with no rVariables. [Informational] |
| NO_WRITE_ACCESS | Write access is not allowed on the CDF file(s). Make sure that the CDF file(s) have the proper file system privileges and ownership. [Error] |
| NOT_A_CDF | Named CDF is corrupted or not actually a CDF. Contact CDF User Support if you are sure that the specified file is a CDF. [Error] |
| READ_ONLY_DISTRIBUTION | Your CDF distribution has been built to allow only read access to CDFs. Check with your system manager if you require write access. [Error] |
| READ_ONLY_MODE | The CDF is in read-only mode — modifications are not allowed. [Error] |
| SGi_ENCODING_UNSUPPORTED | SGi encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| SINGLE_FILE_FORMAT | The specified operation is not applicable to CDFs with the single-file format. For example, it does not make sense to close a variable in a single-file CDF. [Informational] |

| | |
|---|---|
| SUN_ENCODING_UNSUPPORTED | Sun encoding is not supported on the host machine. When a CDF is created/modified, only the host and network encodings are supported. [Error] |
| TOO_MANY_VARS | A multi-file CDF on an IBM PC may contain only a limited number of variables because of the 8.3 file naming convention of MS-DOS. This consists of 100 rVariables and 100 zVariables. [Error] |
| UNSUPPORTED_OPERATION | The attempted operation is not supported at this time. [Error] |
| VAR_ALREADY_CLOSED | The specified variable is already closed. [Informational] |
| VAR_CLOSE_ERROR | Error detected while trying to close variable file. Check that sufficient disk space exists for the variable file and that it has not been corrupted. [Error] |
| VAR_CREATE_ERROR | An error occurred while creating a variable file in a multi-file CDF. Check that a file quota has not been reached. [Error] |
| VAR_EXISTS | Named variable already exists - cannot create or rename. Each variable in a CDF must have a unique name (rVariables and zVariables can not share names). Note that trailing blanks are ignored by the CDF library when comparing variable names. [Error] |
| VAR_NAME_TRUNC | Variable name truncated to CDF_VAR_NAME_LEN characters. The variable was created but with a truncated name. [Warning] |
| VAR_OPEN_ERROR | An error occurred while opening variable file. Check that sufficient privilege exists to open the variable file. Also make sure that the associated variable file exists. [Error] |
| VAR_READ_ERROR | Failed to read variable as requested — error from file system. Check that the associated file is not corrupted. [Error] |
| VAR_WRITE_ERROR | Failed to write variable as requested — error from file system. Check that the associated file is not corrupted. [Error] |
| VAX_ENCODING_UNSUPPORTED | VAX encoding is not supported on the host machine. When a CDF is created/modified only the host and network encodings are supported. [Error] |
| VERSION_1_CDF | This is a Version 1 CDF — some restrictions may apply. The restrictions are described in Section 2.2.1. [Informational] |
| VIRTUAL_RECORD_DATA | One or more of the records are virtual (never actually written 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 2.3.7. [Informational] |

# Appendix D

# C Programming Summary

## D.1 Standard Interface

```
CDFstatus CDFcreate (CDFpath, numDims, dimSizes, encoding, majority, id)
char  *CDFpath;                                                    /* in  */
long  numDims;                                                     /* in  */
long  dimSizes[];                                                  /* in  */
long  encoding;                                                    /* in  */
long  majority;                                                    /* in  */
CDFid *id;                                                         /* out */

CDFstatus CDFopen (CDFpath, id)
char  *CDFpath;                                                    /* in  */
CDFid *id;                                                         /* out */

CDFstatus CDFdoc (id, version, release, text)
CDFid id;                                                          /* in  */
long  *version;                                                    /* out */
long  *release;                                                    /* out */
char  text[CDF_DOCUMENT_LEN+1];                                    /* out */

CDFstatus CDFinquire (id, numDims, dimSizes, encoding, majority, maxRec,
                      numVars, numAttrs)
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 (id)
```

```
CDFid id;                                                       /* in  */

CDFstatus CDFdelete (id)
CDFid id;                                                       /* in  */

CDFstatus CDFerror (status, message)
CDFstatus status;                                               /* in  */
char message[CDF_STATUSTEXT_LEN+1];                             /* out */

CDFstatus CDFattrCreate (id, attrName, attrScope, attrNum)
CDFid id;                                                       /* in  */
char  *attrName;                                                /* in  */
long  attrScope;                                                /* in  */
long  *attrNum;                                                 /* out */

long CDFattrNum (id, attrName)
CDFid id;                                                       /* in  */
char  *attrName;                                                /* in  */

CDFstatus CDFattrRename (id, attrNum, attrName)
CDFid id;                                                       /* in  */
long  attrNum;                                                  /* in  */
char  *attrName;                                                /* in  */

CDFstatus CDFattrInquire (id, attrNum, attrName, attrScope, maxEntry)
CDFid id;                                                       /* in  */
long  attrNum;                                                  /* in  */
char  *attrName;                                                /* out */
long  *attrScope;                                               /* out */
long  *maxEntry;                                                /* out */

CDFstatus CDFattrEntryInquire (id, attrNum, entryNum, dataType,
                               numElements)
CDFid id;                                                       /* in  */
long  attrNum;                                                  /* in  */
long  entryNum;                                                 /* in  */
long  *dataType;                                                /* out */
long  *numElements;                                             /* out */

CDFstatus CDFattrPut (id, attrNum, entryNum, dataType, numElements,
                      value)
CDFid id;                                                       /* in  */
long  attrNum;                                                  /* in  */
long  entryNum;                                                 /* in  */
long  dataType;                                                 /* in  */
long  numElements;                                              /* in  */
void  *value;                                                   /* in  */

CDFstatus CDFattrGet (id, attrNum, entryNum, value)
CDFid id;                                                       /* in  */
long  attrNum;                                                  /* in  */
```

```
long  entryNum;                                                 /* in  */
void  *value;                                                   /* out */

CDFstatus CDFattrCorrectScope (id, attrNum, attrScope)
CDFid id;                                                       /* in  */
long  attrNum;                                                  /* in  */
long  attrScope)                                                /* in  */

CDFstatus CDFvarCreate (id, varName, dataType, numElements,
                        recVariances, dimVariances, varNum)
CDFid id;                                                       /* in  */
char  *varName;                                                 /* in  */
long  dataType;                                                 /* in  */
long  numElements;                                              /* in  */
long  recVariance;                                              /* in  */
long  dimVariances[];                                           /* in  */
long  *varNum;                                                  /* out */

long CDFvarNum (id, varName)
CDFid id;                                                       /* in  */
char  *varName;                                                 /* in  */

CDFstatus CDFvarRename (id, varNum, varName)
CDFid id;                                                       /* in  */
long  varNum;                                                   /* in  */
char  *varName;                                                 /* in  */

CDFstatus CDFvarInquire (id, varNum, varName, dataType, numElements,
                         recVariance, dimVariances)
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 (id, varNum, recNum, indices, value)
CDFid id;                                                       /* in  */
long  varNum;                                                   /* in  */
long  recNum;                                                   /* in  */
long  indices[];                                                /* in  */
void  *value;                                                   /* in  */

CDFstatus CDFvarGet (id, varNum, recNum, indices, value)
CDFid id;                                                       /* in  */
long  varNum;                                                   /* in  */
long  recNum;                                                   /* in  */
long  indices[];                                                /* in  */
void  *value;                                                   /* out */
```

```
CDFstatus CDFvarHyperPut (id, varNum, recStart, recCount, recInterval,
                          indices, counts, intervals, buffer)
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 (id, varNum, recStart, recCount, recInterval,
                          indices, counts, intervals, buffer)
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 */

CDFstatus CDFvarClose (id, varNum)
CDFid id;                                                       /* in  */
long  varNum;                                                   /* in  */
```

## D.2 Internal Interface

```
CDFstatus CDFlib (op, ...)
long  op;                                                       /* in  */

   CREATE_    CDF_                    char *CDFpath             /* in  */
                                      long numDims              /* in  */
                                      long dimSizes[]           /* in  */
                                      CDFid *id                 /* out */
              rVAR_                   char *varName             /* in  */
                                      long dataType             /* in  */
                                      long numElements          /* in  */
                                      long recVary              /* in  */
                                      long dimVarys             /* in  */
                                      long *varNum              /* out */
              zVAR_                   char *varName             /* in  */
                                      long dataType             /* in  */
                                      long numElements          /* in  */
                                      long numDims              /* in  */
                                      long dimSizes[]           /* in  */
                                      long recVary              /* in  */
```

```
                                long dimVarys                   /* in  */
                                long *varNum                    /* out */
            ATTR_               char *attrName                  /* in  */
                                long scope                      /* in  */
                                long *attrNum                   /* out */

OPEN_       CDF_                char *CDFpath                   /* in  */
                                CDFid *id                       /* out */

DELETE_     CDF_

CLOSE_      CDF_
            rVAR_
            zVAR_

SELECT_     CDF_                CDFid id                        /* in  */
            CDF_STATUS_         CDFstatus status                /* in  */
            CDF_READONLY_MODE_  long mode                       /* in  */
            CDF_zMODE_          long mode                       /* in  */
            CDF_NEGtoPOSfp0_MODE_ long mode                     /* in  */
            CDF_DECODING_       long decoding                   /* in  */
            CDF_CACHESIZE_      long numBuffers                 /* in  */
            rVAR_               long varNum                     /* in  */
            rVAR_NAME_          char *varName                   /* in  */
            rVARs_RECNUMBER_    long recNum                     /* in  */
            rVARs_RECCOUNT_     long recCount                   /* in  */
            rVARs_RECINTERVAL_  long recInterval                /* in  */
            rVARs_DIMINDICES_   long indices[]                  /* in  */
            rVARs_DIMCOUNTS_    long counts[]                   /* in  */
            rVARs_DIMINTERVALS_ long intervals[]                /* in  */
            rVAR_SEQPOS_        long recNum                     /* in  */
                                long indices[]                  /* in  */
            rVAR_CACHESIZE_     long numBuffers                 /* in  */
            rVARs_CACHESIZE_    long numBuffers                 /* in  */
            zVAR_               long varNum                     /* in  */
            zVAR_NAME_          char *varName                   /* in  */
            zVAR_RECNUMBER_     long recNum                     /* in  */
            zVAR_RECCOUNT_      long recCount                   /* in  */
            zVAR_RECINTERVAL_   long recInterval                /* in  */
            zVAR_DIMINDICES_    long indices[]                  /* in  */
            zVAR_DIMCOUNTS_     long counts[]                   /* in  */
            zVAR_DIMINTERVALS_  long intervals[]                /* in  */
            zVAR_SEQPOS_        long recNum                     /* in  */
                                long indices[]                  /* in  */
            zVAR_CACHESIZE_     long numBuffers                 /* in  */
            zVARs_CACHESIZE_    long numBuffers                 /* in  */
            ATTR_               long attrNum                    /* in  */
            ATTR_NAME_          char *attrName                  /* in  */
            gENTRY_             long entryNum                   /* in  */
            rENTRY_             long entryNum                   /* in  */
            rENTRY_NAME_        char *varName                   /* in  */
```

```
            zENTRY_                 long entryNum               /* in  */
            zENTRY_NAME_            char *varName               /* in  */

CONFIRM_    CDF_NAME_               char *CDFpath               /* out */
            CDF_READONLY_MODE_      long *mode                  /* out */
            CDF_zMODE_              long *mode                  /* out */
            CDF_NEGtoPOSfp0_MODE_   long *mode                  /* out */
            CDF_DECODING_           long *decoding              /* out */
            CDF_CACHESIZE_          long *numBuffers            /* out */
            rVAR_                   long *varNum                /* out */
            rVAR_EXISTENCE_         char *varName               /* in  */
            rVARs_RECNUMBER_        long *recNum                /* out */
            rVARs_RECCOUNT_         long *recCount              /* out */
            rVARs_RECINTERVAL_      long *recInterval           /* out */
            rVARs_DIMINDICES_       long indices[]              /* out */
            rVARs_DIMCOUNTS_        long counts[]               /* out */
            rVARs_DIMINTERVALS_     long intervals[]            /* out */
            rVAR_SEQPOS_            long *recNum                /* out */
                                    long indices[]              /* out */
            rVAR_CACHESIZE_         long *numBuffers            /* out */
            zVAR_                   long *varNum                /* out */
            zVAR_EXISTENCE_         char *varName               /* in  */
            zVAR_RECNUMBER_         long *recNum                /* out */
            zVAR_RECCOUNT_          long *recCount              /* out */
            zVAR_RECINTERVAL_       long *recInterval           /* out */
            zVAR_DIMINDICES_        long indices[]              /* out */
            zVAR_DIMCOUNTS_         long counts[]               /* out */
            zVAR_DIMINTERVALS_      long intervals[]            /* out */
            zVAR_SEQPOS_            long *recNum                /* out */
                                    long indices[]              /* out */
            zVAR_CACHESIZE_         long *numBuffers            /* out */
            ATTR_                   long *attrNum               /* out */
            ATTR_EXISTENCE_         char *attrName              /* in  */
            gENTRY_                 long *entryNum              /* out */
            rENTRY_                 long *entryNum              /* out */
            zENTRY_                 long *entryNum              /* out */
            gENTRY_EXISTENCE_       long entryNum               /* in  */
            rENTRY_EXISTENCE_       long entryNum               /* in  */
            zENTRY_EXISTENCE_       long entryNum               /* in  */

PUT_        CDF_ENCODING_           long encoding               /* in  */
            CDF_MAJORITY_           long majority               /* in  */
            CDF_FORMAT_             long format                 /* in  */
            rVAR_NAME_              char *varName               /* in  */
            rVAR_DATASPEC_          long dataType               /* in  */
                                    long numElements            /* in  */
            rVAR_RECVARY_           long recVary                /* in  */
            rVAR_DIMVARYS_          long dimVarys[]             /* in  */
            rVAR_PADVALUE_          void *value                 /* in  */
            rVAR_INITIALRECS_       long NinitialRecs           /* in  */
            rVAR_EXTENDRECS_        long NextendRecs            /* in  */
```

```
            rVAR_DATA_              void *value                 /* in  */
            rVAR_HYPERDATA_         void *buffer                /* in  */
            rVAR_SEQDATA_           void *value                 /* in  */
            rVARs_RECDATA_          void *buffer                /* in  */
            zVAR_NAME_              char *varName               /* in  */
            zVAR_DATASPEC_          long dataType               /* in  */
                                    long numElements            /* in  */
            zVAR_RECVARY_           long recVary                /* in  */
            zVAR_DIMVARYS_          long dimVarys[]             /* in  */
            zVAR_PADVALUE_          void *value                 /* in  */
            zVAR_INITIALRECS_       long NinitialRecs           /* in  */
            zVAR_EXTENDRECS_        long NextendRecs            /* in  */
            zVAR_DATA_              void *value                 /* in  */
            zVAR_HYPERDATA_         void *buffer                /* in  */
            zVAR_SEQDATA_           void *value                 /* in  */
            zVARs_RECDATA_          void *buffer                /* in  */
            ATTR_SCOPE_             long scope                  /* in  */
            ATTR_NAME_              char *attrName              /* in  */
            gENTRY_DATASPEC_        long dataType               /* in  */
                                    long numElements            /* in  */
            rENTRY_DATASPEC_        long dataType               /* in  */
                                    long numElements            /* in  */
            zENTRY_DATASPEC_        long dataType               /* in  */
                                    long numElements            /* in  */
            gENTRY_DATA_            long dataType               /* in  */
                                    long numElements            /* in  */
                                    void *value                 /* in  */
            rENTRY_DATA_            long dataType               /* in  */
                                    long numElements            /* in  */
                                    void *value                 /* in  */
            zENTRY_DATA_            long dataType               /* in  */
                                    long numElements            /* in  */
                                    void *value                 /* in  */

GET_        rVARs_NUMDIMS_          long *numDims               /* out */
            rVARs_DIMSIZES_         long dimSizes[]             /* out */
            zVAR_NUMDIMS_           long *numDims               /* out */
            zVAR_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_NUMrVARS_           long *numVars               /* out */
            CDF_NUMzVARS_           long *numVars               /* out */
            CDF_NUMATTRS_           long *numAttrs              /* out */
            CDF_NUMgATTRS_          long *numAttrs              /* out */
            CDF_NUMvATTRS_          long *numAttrs              /* out */
            rVARs_MAXREC_           long *maxRec                /* out */
            zVARs_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 */
rVAR_NAME_              char *varName               /* out */
rVAR_DATATYPE_          long *dataType              /* out */
rVAR_NUMELEMS_          long *numElements           /* out */
rVAR_RECVARY_           long *recVary               /* out */
rVAR_DIMVARYS_          long dimVarys[]             /* out */
rVAR_NUMBER_            char *varName               /* in  */
                        long *varNum                /* out */
rVAR_PADVALUE_          void *value                 /* out */
rVAR_EXTENDRECS_        long *NextendRecs           /* out */
rVAR_MAXREC_            long *maxRec                /* out */
rVAR_MAXallocREC_       long *maxRec                /* out */
rVAR_nINDEXRECORDS_     long *numRecords            /* out */
rVAR_nINDEXENTRIES_     long *numEntries            /* out */
rVAR_DATA_              void *value                 /* out */
rVAR_HYPERDATA_         void *buffer                /* out */
rVAR_SEQDATA_           void *value                 /* out */
rVARs_RECDATA_          void *buffer                /* out */
zVAR_NAME_              char *varName               /* out */
zVAR_DATATYPE_          long *dataType              /* out */
zVAR_NUMELEMS_          long *numElements           /* out */
zVAR_RECVARY_           long *recVary               /* out */
zVAR_DIMVARYS_          long dimVarys[]             /* out */
zVAR_NUMBER_            char *varName               /* in  */
                        long *varNum                /* out */
zVAR_PADVALUE_          void *value                 /* out */
zVAR_EXTENDRECS_        long *NextendRecs           /* out */
zVAR_MAXREC_            long *maxRec                /* out */
zVAR_MAXallocREC_       long *maxRec                /* out */
zVAR_nINDEXRECORDS_     long *numRecords            /* out */
zVAR_nINDEXENTRIES_     long *numEntries            /* out */
zVAR_DATA_              void *value                 /* out */
zVAR_HYPERDATA_         void *buffer                /* out */
zVAR_SEQDATA_           void *value                 /* out */
zVARs_RECDATA_          void *buffer                /* out */
ATTR_SCOPE_             long *scope                 /* out */
ATTR_NAME_              char *attrName              /* out */
ATTR_NUMBER_            char *attrName              /* in  */
                        long *attrNum               /* out */
ATTR_MAXgENTRY_         long *maxEntry              /* out */
ATTR_NUMgENTRIES_       long *numEntries            /* out */
ATTR_MAXrENTRY_         long *maxEntry              /* out */
ATTR_NUMrENTRIES_       long *numEntries            /* out */
ATTR_MAXzENTRY_         long *maxEntry              /* out */
ATTR_NUMzENTRIES_       long *numEntries            /* out */
gENTRY_DATATYPE_        long *dataType              /* out */
```

```
gENTRY_NUMELEMS_        long *numElements           /* out */
gENTRY_DATA_            void *value                 /* out */
rENTRY_DATATYPE_        long *dataType              /* out */
rENTRY_NUMELEMS_        long *numElements           /* out */
rENTRY_DATA_            void *value                 /* out */
zENTRY_DATATYPE_        long *dataType              /* out */
zENTRY_NUMELEMS_        long *numElements           /* out */
zENTRY_DATA_            void *value                 /* out */
STATUS_TEXT_            char *text                  /* out */
```

## D.3 EPOCH Utility Routines

```
double computeEPOCH (year, month, day, hour, minute, second, msec)
long year;                                                  /* in  */
long month;                                                 /* in  */
long day;                                                   /* in  */
long hour;                                                  /* in  */
long minute;                                                /* in  */
long second;                                                /* in  */
long msec;                                                  /* in  */

void EPOCHbreakdown (epoch, year, month, day, hour, minute, second, msec)
double epoch;                                               /* in  */
long *year;                                                 /* out */
long *month;                                                /* out */
long *day;                                                  /* out */
long *hour;                                                 /* out */
long *minute;                                               /* out */
long *second;                                               /* out */
long *msec;                                                 /* out */

double parseEPOCH (epString)
char epString[EPOCH_STRING_LEN+1];                          /* in  */

void encodeEPOCH (epoch, epString)
double epoch;                                               /* in  */
char epString[EPOCH_STRING_LEN+1];                          /* out */

void encodeEPOCH1 (epoch, epString)
double epoch;                                               /* in  */
char epString[EPOCH1_STRING_LEN+1];                         /* out */

void encodeEPOCH2 (epoch, epString)
double epoch;                                               /* in  */
char epString[EPOCH2_STRING_LEN+1];                         /* out */
```

# Appendix E

# Fortran Programming Summary

## E.1 Standard Interface

```
SUBROUTINE CDF_create (CDF_path, num_dims, dim_sizes, encoding, majority,
1                      id, status)
CHARACTER CDF_path                                                  ! 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_path, id, status)
CHARACTER CDF_path*(*)                                              ! 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

SUBROUTINE CDF_var_close (id, var_num)
INTEGER*4 id                                                    ! in
INTEGER*4 var_num                                               ! in
```

## E.2 Internal Interface

```
INTEGER*4 FUNCTION CDF_lib (op, ..., status)
INTEGER*4 op                                                        ! in
.
.
.
INTEGER*4 status                                                    ! out

   CREATE_   CDF_                    CHARACTER CDF_path*(*)          ! in
                                     INTEGER*4 num_dims              ! in
                                     INTEGER*4 dim_sizes(*)          ! in
                                     INTEGER*4 id                    ! out
             rVAR_                   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
             zVAR_                   CHARACTER var_name*(*)          ! in
                                     INTEGER*4 data_type             ! in
                                     INTEGER*4 num_elements          ! in
                                     INTEGER*4 num_dims              ! in
                                     INTEGER*4 dim_sizes(*)          ! 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_path*(*)          ! in
                                     INTEGER*4 id                    ! out

   DELETE_   CDF_

   CLOSE_    CDF_
             rVAR_
             zVAR_

   SELECT_   CDF_                    INTEGER*4 id                    ! in
             CDF_STATUS_             INTEGER*4 status                ! in
             CDF_READONLY_MODE_      INTEGER*4 mode                  ! in
             CDF_zMODE_              INTEGER*4 mode                  ! in
             CDF_NEGtoPOSfp0_MODE_   INTEGER*4 mode                  ! in
             CDF_DECODING_           INTEGER*4 decoding              ! in
             CDF_CACHESIZE_          INTEGER*4 num_buffers           ! in
             rVAR_                   INTEGER*4 var_num               ! in
             rVAR_NAME_              CHARACTER var_name*(*)          ! in
             rVARs_RECNUMBER_        INTEGER*4 rec_num               ! in
```

```
            rVARs_RECCOUNT_          INTEGER*4 rec_count              ! in
            rVARs_RECINTERVAL_       INTEGER*4 rec_interval           ! in
            rVARs_DIMINDICES_        INTEGER*4 indices(*)             ! in
            rVARs_DIMCOUNTS_         INTEGER*4 counts(*)              ! in
            rVARs_DIMINTERVALS_      INTEGER*4 intervals(*)           ! in
            rVAR_SEQPOS_             INTEGER*4 rec_num                ! in
                                     INTEGER*4 indices(*)             ! in
            rVAR_CACHESIZE_          INTEGER*4 num_buffers            ! in
            rVARs_CACHESIZE_         INTEGER*4 num_buffers            ! in
            zVAR_                    INTEGER*4 var_num                ! in
            zVAR_NAME_               CHARACTER var_name*(*)           ! in
            zVAR_RECNUMBER_          INTEGER*4 rec_num                ! in
            zVAR_RECCOUNT_           INTEGER*4 rec_count              ! in
            zVAR_RECINTERVAL_        INTEGER*4 rec_interval           ! in
            zVAR_DIMINDICES_         INTEGER*4 indices(*)             ! in
            zVAR_DIMCOUNTS_          INTEGER*4 counts(*)              ! in
            zVAR_DIMINTERVALS_       INTEGER*4 intervals(*)           ! in
            zVAR_SEQPOS_             INTEGER*4 rec_num                ! in
                                     INTEGER*4 indices(*)             ! in
            zVAR_CACHESIZE_          INTEGER*4 num_buffers            ! in
            zVARs_CACHESIZE_         INTEGER*4 num_buffers            ! in
            ATTR_                    INTEGER*4 attr_num               ! in
            ATTR_NAME_               CHARACTER attr_name*(*)          ! in
            gENTRY_                  INTEGER*4 entry_num              ! in
            rENTRY_                  INTEGER*4 entry_num              ! in
            rENTRY_NAME_             CHARACTER var_name*(*)           ! in
            zENTRY_                  INTEGER*4 entry_num              ! in
            zENTRY_NAME_             CHARACTER var_name*(*)           ! in

CONFIRM_    CDF_NAME_                CHARACTER CDF_path*(*)           ! out
            CDF_READONLY_MODE_       INTEGER*4 mode                   ! out
            CDF_zMODE_               INTEGER*4 mode                   ! out
            CDF_NEGtoPOSfp0_MODE_    INTEGER*4 mode                   ! out
            CDF_DECODING_            INTEGER*4 decoding               ! out
            CDF_CACHESIZE_           INTEGER*4 num_buffers            ! out
            rVAR_                    INTEGER*4 var_num                ! out
            rVAR_EXISTENCE_          CHARACTER var_name*(*)           ! in
            rVARs_RECNUMBER_         INTEGER*4 rec_num                ! out
            rVARs_RECCOUNT_          INTEGER*4 rec_count              ! out
            rVARs_RECINTERVAL_       INTEGER*4 rec_interval           ! out
            rVARs_DIMINDICES_        INTEGER*4 indices(*)             ! out
            rVARs_DIMCOUNTS_         INTEGER*4 counts(*)              ! out
            rVARs_DIMINTERVALS_      INTEGER*4 intervals(*)           ! out
            rVAR_SEQPOS_             INTEGER*4 rec_num                ! out
                                     INTEGER*4 indices(*)             ! out
            rVAR_CACHESIZE_          INTEGER*4 num_buffers            ! out
            zVAR_                    INTEGER*4 var_num                ! out
            zVAR_EXISTENCE_          CHARACTER var_name*(*)           ! in
            zVAR_RECNUMBER_          INTEGER*4 rec_num                ! out
            zVAR_RECCOUNT_           INTEGER*4 rec_count              ! out
            zVAR_RECINTERVAL_        INTEGER*4 rec_interval           ! out
```

```
            zVAR_DIMINDICES_        INTEGER*4 indices(*)              ! out
            zVAR_DIMCOUNTS_         INTEGER*4 counts(*)               ! out
            zVAR_DIMINTERVALS_      INTEGER*4 intervals(*)            ! out
            zVAR_SEQPOS_            INTEGER*4 rec_num                 ! out
                                    INTEGER*4 indices(*)              ! out
            zVAR_CACHESIZE_         INTEGER*4 num_buffers             ! out
            ATTR_                   INTEGER*4 attr_num                ! out
            ATTR_EXISTENCE_         CHARACTER attr_name*(*)           ! in
            gENTRY_                 INTEGER*4 entry_num               ! out
            rENTRY_                 INTEGER*4 entry_num               ! out
            zENTRY_                 INTEGER*4 entry_num               ! out
            gENTRY_EXISTENCE_       INTEGER*4 entry_num               ! in
            rENTRY_EXISTENCE_       INTEGER*4 entry_num               ! in
            zENTRY_EXISTENCE_       INTEGER*4 entry_num               ! in

PUT_        CDF_ENCODING_           INTEGER*4 encoding                ! in
            CDF_MAJORITY_           INTEGER*4 majority                ! in
            CDF_FORMAT_             INTEGER*4 format                  ! in
            rVAR_NAME_              CHARACTER var_name*(*)            ! in
            rVAR_DATASPEC_          INTEGER*4 data_type               ! in
                                    INTEGER*4 num_elements            ! in
            rVAR_RECVARY_           INTEGER*4 rec_vary                ! in
            rVAR_DIMVARYS_          INTEGER*4 dim_varys(*)            ! in
            rVAR_PADVALUE_          <type> value                      ! in
            rVAR_INITIALRECS_       INTEGER*4 n_initial_recs          ! in
            rVAR_EXTENDRECS_        INTEGER*4 _nextend_recs           ! in
            rVAR_DATA_              <type> value                      ! in
            rVAR_HYPERDATA_         <type> buffer                     ! in
            rVAR_SEQDATA_           <type> value                      ! in
            rVARs_RECDATA_          <type> buffer                     ! in
            zVAR_NAME_              CHARACTER var_name*(*)            ! in
            zVAR_DATASPEC_          INTEGER*4 data_type               ! in
                                    INTEGER*4 num_elements            ! in
            zVAR_RECVARY_           INTEGER*4 rec_vary                ! in
            zVAR_DIMVARYS_          INTEGER*4 dim_varys(*)            ! in
            zVAR_PADVALUE_          <type> value                      ! in
            zVAR_INITIALRECS_       INTEGER*4 n_initial_recs          ! in
            zVAR_EXTENDRECS_        INTEGER*4 _nextend_recs           ! in
            zVAR_DATA_              <type> value                      ! in
            zVAR_HYPERDATA_         <type> buffer                     ! in
            zVAR_SEQDATA_           <type> value                      ! in
            zVARs_RECDATA_          <type> buffer                     ! in
            ATTR_SCOPE_             INTEGER*4 scope                   ! in
            ATTR_NAME_              CHARACTER attr_name*(*)           ! in
            gENTRY_DATASPEC_        INTEGER*4 data_type               ! in
                                    INTEGER*4 num_elements            ! in
            rENTRY_DATASPEC_        INTEGER*4 data_type               ! in
                                    INTEGER*4 num_elements            ! in
            zENTRY_DATASPEC_        INTEGER*4 data_type               ! in
                                    INTEGER*4 num_elements            ! in
            gENTRY_DATA_            INTEGER*4 data_type               ! in
```

```
                                INTEGER*4 num_elements              ! in
                                <type> value                        ! in
            rENTRY_DATA_        INTEGER*4 data_type                 ! in
                                INTEGER*4 num_elements              ! in
                                <type> value                        ! in
            zENTRY_DATA_        INTEGER*4 data_type                 ! in
                                INTEGER*4 num_elements              ! in
                                <type> value                        ! in

GET_        rVARs_NUMDIMS_      INTEGER*4 num_dims                  ! out
            rVARs_DIMSIZES_     INTEGER*4 dim_sizes(*)              ! out
            zVAR_NUMDIMS_       INTEGER*4 num_dims                  ! out
            zVAR_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_NUMrVARS_       INTEGER*4 num_vars                  ! out
            CDF_NUMzVARS_       INTEGER*4 num_vars                  ! out
            CDF_NUMATTRS_       INTEGER*4 num_attrs                 ! out
            CDF_NUMgATTRS_      INTEGER*4 num_attrs                 ! out
            CDF_NUMvATTRS_      INTEGER*4 num_attrs                 ! out
            rVARs_MAXREC_       INTEGER*4 max_rec                   ! out
            zVARs_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
            rVAR_NAME_          CHARACTER var_name*(*)              ! out
            rVAR_DATATYPE_      INTEGER*4 data_type                 ! out
            rVAR_NUMELEMS_      INTEGER*4 num_elements              ! out
            rVAR_RECVARY_       INTEGER*4 rec_vary                  ! out
            rVAR_DIMVARYS_      INTEGER*4 dim_varys(*)              ! out
            rVAR_NUMBER_        CHARACTER var_name*(*)              ! in
                                INTEGER*4 var_num                   ! out
            rVAR_PADVALUE_      <type> value                        ! out
            rVAR_EXTENDRECS_    INTEGER*4 n_extend_recs             ! out
            rVAR_MAXREC_        INTEGER*4 max_rec                   ! out
            rVAR_MAXallocREC_   INTEGER*4 max_rec                   ! out
            rVAR_nINDEXRECORDS_ INTEGER*4 num_records               ! out
            rVAR_nINDEXENTRIES_ INTEGER*4 num_entries               ! out
            rVAR_DATA_          <type> value                        ! out
            rVAR_HYPERDATA_     <type> buffer                       ! out
            rVAR_SEQDATA_       <type> value                        ! out
            rVARs_RECDATA_      <type> buffer                       ! out
            zVAR_NAME_          CHARACTER var_name*(*)              ! out
            zVAR_DATATYPE_      INTEGER*4 data_type                 ! out
```

```
zVAR_NUMELEMS_          INTEGER*4 num_elements          ! out
zVAR_RECVARY_           INTEGER*4 rec_vary              ! out
zVAR_DIMVARYS_          INTEGER*4 dim_varys(*)          ! out
zVAR_NUMBER_            CHARACTER var_name*(*)          ! in
                        INTEGER*4 var_num               ! out
zVAR_PADVALUE_          <type> value                    ! out
zVAR_EXTENDRECS_        INTEGER*4 n_extend_recs         ! out
zVAR_MAXREC_            INTEGER*4 max_rec               ! out
zVAR_MAXallocREC_       INTEGER*4 max_rec               ! out
zVAR_nINDEXRECORDS_     INTEGER*4 num_records           ! out
zVAR_nINDEXENTRIES_     INTEGER*4 num_entries           ! out
zVAR_DATA_              <type> value                    ! out
zVAR_HYPERDATA_         <type> buffer                   ! out
zVAR_SEQDATA_           <type> value                    ! out
zVARs_RECDATA_          <type> buffer                   ! 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_MAXgENTRY_         INTEGER*4 max_entry             ! out
ATTR_NUMgENTRIES_       INTEGER*4 num_entries           ! out
ATTR_MAXrENTRY_         INTEGER*4 max_entry             ! out
ATTR_NUMrENTRIES_       INTEGER*4 num_entries           ! out
ATTR_MAXzENTRY_         INTEGER*4 max_entry             ! out
ATTR_NUMzENTRIES_       INTEGER*4 num_entries           ! out
gENTRY_DATATYPE_        INTEGER*4 data_type             ! out
gENTRY_NUMELEMS_        INTEGER*4 num_elements          ! out
gENTRY_DATA_            <type> value                    ! out
rENTRY_DATATYPE_        INTEGER*4 data_type             ! out
rENTRY_NUMELEMS_        INTEGER*4 num_elements          ! out
rENTRY_DATA_            <type> value                    ! out
zENTRY_DATATYPE_        INTEGER*4 data_type             ! out
zENTRY_NUMELEMS_        INTEGER*4 num_elements          ! out
zENTRY_DATA_            <type> value                    ! out
STATUS_TEXT_            CHARACTER text*(*)              ! out
```

## E.3 EPOCH Utility Routines

```
SUBROUTINE compute_EPOCH (year, month, day, hour, minute, second, msec, epoch)
INTEGER*4 year                                                          ! in
INTEGER*4 month                                                         ! in
INTEGER*4 day                                                           ! in
INTEGER*4 hour                                                          ! in
INTEGER*4 minute                                                        ! in
INTEGER*4 second                                                        ! in
INTEGER*4 msec                                                          ! in
REAL*8    epoch                                                         ! out

SUBROUTINE EPOCH_breakdown (epoch, year, month, day, hour, minute, second,
```

```
1                           msec)
REAL*8    epoch                                                 ! in
INTEGER*4 year                                                  ! out
INTEGER*4 month                                                 ! out
INTEGER*4 day                                                   ! out
INTEGER*4 hour                                                  ! out
INTEGER*4 minute                                                ! out
INTEGER*4 second                                                ! out
INTEGER*4 msec                                                  ! out

SUBROUTINE parse_EPOCH (ep_string, epoch)
CHARACTER ep_string*(EPOCH_STRING_LEN)                          ! in
REAL*8    epoch                                                 ! out

SUBROUTINE parse_EPOCH (ep_string, epoch)
CHARACTER ep_string*(EPOCH_STRING_LEN)                          ! in
REAL*8    epoch                                                 ! out

SUBROUTINE encode_EPOCH (epoch, ep_string)
REAL*8    epoch                                                 ! in
CHARACTER ep_string*(EPOCH_STRING_LEN)                          ! out

SUBROUTINE encode_EPOCH1 (epoch, ep_string)
REAL*8    epoch                                                 ! in
CHARACTER ep_string*(EPOCH1_STRING_LEN)                         ! out

SUBROUTINE encode_EPOCH2 (epoch, ep_string)
REAL*8    epoch                                                 ! in
CHARACTER ep_string*(EPOCH2_STRING_LEN)                         ! out
```

# Appendix F

# CDF Release Notes

## F.1 Introduction

This chapter should provide you with all of the information you need to know to begin using CDF V2.4 assuming that you are already familar with CDF V2.3. For additional information contact CDF User Support.

## F.2 Supported Systems

CDF V2.4 is currently supported on the following computers/operating systems.

1. VAX (VMS)
2. Sun (SunOS & SOLARIS)
3. DECstation (ULTRIX)
4. Silicon Graphics Iris & Power Series (IRIX)
5. IBM RS6000 series (AIX)
6. HP 9000 series (HP-UX)
7. IBM PC (MS-DOS)
8. NeXT (Mach)
9. DEC Alpha (OSF/1 & OpenVMS)

Additional ports are planned for the future. They include Cray (UNICOS), Macintosh (MacOS & AUX), and a 32-bit distribution for the IBM PC (MS-DOS).

## F.3 Changes for CDF V2.4

Several changes have been made to the CDF distribution for V2.4. CDF V2.4 differs from CDF V2.3 in the following ways.

1. The distribution has been ported to the DEC Alpha (OSF/1 & OpenVMS) and the Sun (SOLARIS). When OpenVMS is run on a DEC Alpha, both the `D_FLOAT` and `G_FLOAT` representations are supported for double-precision floating-point values. When the CDF distribution is built on a DEC Alpha running OpenVMS, an option exists to build the distribution with a default of `D_FLOAT` or `G_FLOAT` as the host encoding/decoding representation.

2. An open CDF may now be configured into zMode. This causes the rVariables of a CDF to appear as zVariables (already existing zVariables are not affected). This mode is provided to assist in the transition to zVariables. The toolkit programs have also been modified to allow CDFs to be placed into zMode. Section 2.1.2 describes zMode in detail.

3. Support for the Internal Interface (`CDFlib`) has been added to the IDL interface provided with the CDF distribution (the CDF/IDL interface). IDL has also added this support to their CDF interface (in a limited way). The EPOCH manipulation routines have also been added to the IDL interface provided with the CDF distribution. In addition, an on-line help file for the IDL interface provided with the CDF distribution is available.

4. It is now possible to request a decoding other than that of the host computer when attribute entry and variable values are passed out by the CDF library. For example, VAX encoded values may be requested while running on a Sun computer. This has been provided to assist in the development of client/server systems where the server running on one type of computer reads data values out of a CDF and then transfers those values to a client running on a different kind of computer. The server would request the host decoding of the client computer before reading the values. No conversions would be necessary by the client. Section 2.2.8 describes this option in more detail.

5. A new toolkit program, `CDFedit`, has been added to the distribution. `CDFedit` allows virtually all of the contents of a CDF to be displayed and/or modified.

6. When using the Internal Interface in a C/Fortran application, the need to select the current CDF in every call to `CDFlib`/`CDF_lib` has been eliminated. The current CDF is maintained by the CDF library from one call to the next of `CDFlib`/`CDF_lib`. In CDF's IDL interface, however, selecting the current CDF is still required in each call to the Internal Interface (`CDFlib`).

7. Support for the Internal Interface from Fortran applications (`CDF_lib`) on the IBM PC has been added. Its use requires an additional include file to be included by an application.

8. The ability to read/write a full-physical record from/to up to all of the rVariables/zVariables with a single call to the CDF library has been added to the distribution. This mode of access is referred to as multiple variable access and is described in Section 2.3.15.

9. An operation has been added to the Internal Interface that allows the current rEntry/zEntry numbers to be selected by rVariable/zVariable name.

10. An option has been added to the CDF library that allows an application program to specify the number of cache buffers to be used with an open file. Section 2.1.4 describes the caching scheme used by the CDF library.

11. The ability to rename a single-file CDF with a different (or no) extension has been added. The CDF library will attempt to open a CDF by using the specification exactly as given as well as by appending `.cdf` automatically (see Section 2.2.5).

12. An operation has been added to the Internal Interface that allows the indexing statistics of a variable in a single-file CDF to be inquired. Section 2.2.6 describes the indexing scheme used for variables in a single-file CDF.

13. New EPOCH manipulation routines have been added that allow EPOCH character strings to be parsed and EPOCH values to be encoded. These routines are described in Sections 4.9 and 5.10 for C and Fortran applications, respectively.

14. The default qualifiers for the toolkit programs can now be configured when the CDF distribution is built and installed. These defaults can be overridden when a toolkit program is executed by specifying the desired qualifiers on the command line. The `README` file for each particular type of operating systems (VMS, UNIX, or MS-DOS) explains how to configure the CDF distribution.

15. An option has been added to the CDF library that allows the floating-point value `-0.0` to be converted to `0.0` automatically when detected. A value of `-0.0` is illegal on VAXes and DEC Alphas running OpenVMS. This option can be requested by an application (via the Internal Interface) or with a qualifier when running a toolkit program.

16. `CDFlist` has been modified to allow variable values to be displayed in either a vertical or horizontal format (when in zMode). The efficiency of `CDFlist` has also been improved (by using the `MONOTON` attribute or by the user specifying the monotonicity of the variables).

17. A qualifier has been added to the `CDFlist` and `CDFwalk` toolkit programs that allows the specification of the initial settings on the various menus available when running these programs.

18. A new qualifier has been added to the `CDFskeleton` toolkit program that allows `FILLVAL` attribute entry values to be used to specify the pad values for the corresponding variables.

19. A qualifier has been added to the `CDFstats` toolkit program that requests that the corresponding `MONOTON` attribute entries be updated based on the monotonicities detected for the variables.

20. A qualifier has been added to the `CDFcompare` toolkit program that requests that the locations (record/indices) of differences detected between two variables be displayed.

21. A qualifier has been added to the `CDFinquire` toolkit program that allows the specification of the variables for which values should be displayed.

22. The capability to use the `FORMAT` attribute has been added to all of the toolkit programs.

23. A qualifier has been added to most of the toolkit programs that allows a user to select which types of status codes from the CDF library should be reported: errors, warnings, and/or informationals.

24. The `CDFbrowse` toolkit program has been removed from the distribution. Its functionality remains by running `CDFedit` in a browsing mode.[1]

[1] The alias/symbol `CDFbrowse` still exists in the "definitions" file on UNIX/VMS systems but now executes `CDFedit` in a browse-only mode.

# Appendix G

# Glossary

| | |
|---|---|
| assumed scope | An attribute scope used by the CDF library when it cannot determine the definite scope. This can occur when Version 1 CDFs are accessed. |
| attribute | A CDF object with which entries of metadata are associated. |
| big-endian | The byte ordering in which the most significant byte (MSB) is stored in the lowest memory location. |
| caching | The method used by the CDF library to improve performance when accessing a file. An attempt is made to keep commonly accessed blocks of the file in memory rather than repeatedly reading them from disk. |
| CDF | This term is used in more than one way...<br>1. The actual files that contain your data/metadata. For example: The CDF library must be used to create a "CDF."<br>2. The software distribution containing the CDF library, include files, and toolkit. For example: We like using "CDF" to store our data. |
| CDF distribution | The directory of software consisting of the CDF library, include files, and toolkit. |
| CDF library | The software library that is used to access a CDF. |
| CDF toolkit | A set of utility programs which ease the creation, modification, and verification of CDFs. |
| `CDF_OK` | A completion status code indicating unqualified success. |
| `cdf.h` | An include file used in C applications. |
| `cdf.inc` | An include file used in Fortran applications. |

| | |
|---|---|
| `cdfmsf.inc` | An include file used in Microsoft Fortran applications. |
| column-major | The variable majority where the first index of a multidimensional array of values increments the fastest. |
| conceptual view | The way that values along a dimension having a variance of `NOVARY` are made to appear as if they do actually exist (only one value is actually physically stored). This also applies to records beyond the last record actually stored. The conceptual view of a variable consists of "virtual" records and values (in addition to the physical records and values actually stored). |
| current | When the Internal Interface is used, current objects/states are those items affected when an operation is performed. For example, a current CDF is selected and then any operation performed involving a CDF is performed on that CDF (until a different current CDF is selected). |
| data specification | For a variable or attribute entry the data type and number of elements of that data. |
| data type | For a variable or attribute entry, the type of data being stored (e.g., integer, floating-point, character). |
| decoding | The integer/floating-point representation of data values passed to an application by the CDF library as they are read from a CDF. This is independent of the way the data values are physically stored in the CDF. |
| dimensionality | The number of dimensions and the dimension sizes for the rVariables or a zVariable. |
| encoding | The integer/floating-point representation of the data values physically stored in a CDF. |
| entry | A CDF object in which metadata is stored. An entry is associated with an attribute. |
| `EPOCH` | A variable of the NSSDC Standard used to store "time" values. The `CDF_EPOCH` data type is used with this variable. |
| error code | A status code indicating that a fatal condition was encountered. The operation was aborted. |
| extend records | The number of records actually allocated for a variable in a single-file CDF when a new record is written. More records may be allocated than are actually needed in order to keep a variable's records as contiguous as possible (with the assumption that the records will eventually be written by an application). |
| format | In reference to a CDF, the way in which files are used to store the CDF's control/data/metadata. This may be single-file or multi-file. |

| | |
|---|---|
| full-physical record | A variable record consisting of values exactly as physically stored in the CDF. |
| gAttribute | A global scoped attribute. |
| gEntry | An entry for a gAttribute. |
| global scope | Global scope indicates that an attribute describes some property of the entire CDF. |
| host decoding | The decoding of the computer currently being used. |
| host encoding | The encoding of the computer currently being used. |
| hyper access | A variable access method in which multiple records/values are read/written for a variable. |
| IDL Interface | A set of functions callable from within IDL (Interactive Data Language) that allow access to CDFs. The CDF distribution contains an IDL interface in addition to the CDF interface built into IDL by Research Systems, Inc. (RSI — the distributors of IDL). |
| IEEE 754 | The floating-point representation of XDR. |
| include file | A file, included by a C or Fortran application, that contains constants recognized by the CDF library pertaining to various aspects of CDF objects/states. |
| indexing | The method used in a single-file CDF to keep track of where each variable's records are located. |
| informational code | A status code indicating success but providing some additional information that may be of interest. |
| Internal Interface | A set of routines in the CDF library callable from C and Fortran applications that provide all types of access to CDFs. |
| item | When the Internal Interface is used, an object or state on which an operation is performed. |
| `libcdf.a` | The CDF library on UNIX systems. |
| `libcdf.lib` | The CDF library on MS-DOS systems. |
| `LIBCDF.OLB` | The CDF library on VMS and OpenVMS systems. |
| little-endian | The byte ordering in which the least significant byte (LSB) is stored in the lowest memory location. |
| majority | The order in which the values of a multidimensional array are stored. This may be either row-major or column-major. |
| monotonicity | The property of a variable that specifies whether or not that variable's values increment or decrement (or neither) along a dimension or from record to record. |
| multi-file | A CDF format. Multi-file CDFs consist of one file for control/metadata and one file per variable of data. |

| | |
|---|---|
| multiple variable access | A variable access method in which one full-physical record is read/written for each of one or more variables. |
| network encoding | The encoding that uses the XDR representation. |
| `NOVARY` | A record/dimension variance indicating that the values do not change from record to record or along a dimension. |
| NRV variable | Non-record variant variable. A variable whose values do not change from record to record (a record variance of `NOVARY`). |
| NSSDC | National Space Science Data Center. |
| NSSDC Standard | A set of guidelines for creating a CDF containing attributes expected by NSSDC tools. |
| number of elements | For a variable the number of instances of the data type at each value. For an attribute entry the number of instances of the data type for that entry. |
| object | When the Internal Interface is used, an item that exists and may be accessed/manipulated (e.g., a CDF or variable). |
| Obsolete Interface | The CDF Version 1 programming interface for Fortran applications on VAX/VMS systems only. The use of this interface is discouraged. |
| operation | When the Internal Interface is used, an action performed on an item (e.g., creating or writing). |
| pad value | A value written to a variable by the CDF library in those cases where a physical record must be written but not all of its values have been specified by an application. For example, when a single value is written to a new record, all of the other values are written using the pad value. |
| physical record | A variable record actually stored in a CDF. |
| physical value | A variable value actually stored in a CDF. |
| read-only | A mode of the CDF library in which modifications to a CDF are not allowed. |
| rEntry | An entry for a vAttribute corresponding to an rVariable. |
| row-major | The variable majority where the last index of a multidimensional array of values increments the fastest. |
| RV variable | Record variant variable. A variable whose values change from record to record (a record variance of `VARY`). |
| rVariable | "R" variable. A CDF object in which data values are stored. All rVariables have the same dimensionality. |
| scope | The intended use for an attribute. This may be global scope or variable scope. |

| | |
|---|---|
| sequential access | A variable access method in which values are read/written in the physical order in which they are stored in the CDF. |
| single-file | A CDF format. Single-file CDFs are entirely contained within one file. |
| single value access | A variable access method in which exactly one value is read/written for a variable. |
| skeleton CDF | A CDF consisting of only control, metadata (attributes and attribute entries), and NRV variable values. |
| skeleton table | A text file containing the control, metadata, and NRV variable values of a skeleton CDF. A skeleton table is read by the `CDFskeleton` toolkit program, which then creates the corresponding skeleton CDF. The `SkeletonTable` toolkit program can be used to create a skeleton table from a CDF. |
| Standard Interface | A set of routines in the CDF library callable from C and Fortran applications that provide access to a commonly used subset of the capabilities of the Internal Interface. This interface was defined with the release of CDF V2.0 and has not changed since. New features since that time are available only through the Internal Interface (e.g., zVariables and zMode). |
| state | When the Internal Interface is used, a property pertaining to an object (e.g., a CDF's format or variable's data specification). |
| status code | The result of a CDF function/subroutine call. `CDF_OK` indicates unqualified success. |
| status handler | A function/subroutine that acts upon a status code received from the CDF library. |
| variable scope | Variable scope indicates that an attribute describes some property of each variable. |
| variance, record | The property of a variable that specifies whether or not its values change from record to record. |
| variance, dimension | The property of a variable that specifies whether or not the values along a dimension change or stay the same. |
| `VARY` | A record/dimension variance indicating that the values change from record to record or along a dimension. |
| vAttribute | A variable scoped attribute. |
| virtual record | A variable record that is not actually stored in a CDF but does appear in the conceptual view of the CDF. Virtual records would be those records beyond the first record of an NRV variable and those records beyond the last record actually written to an RV variable. |
| virtual value | A variable value this is not actually stored in a CDF but does appear in the conceptual view of the CDF. Virtual values would be those values beyond the first value of a dimension whose |

| | |
|---|---|
| | variance is `NOVARY`. |
| warning code | A status code indicating that the operation did complete but probably not as expected. |
| XDR | External Data Representation. An integer/floating-point representation using big-endian byte ordering and the IEEE 754 floating-point representation. |
| zEntry | An entry for a vAttribute corresponding to a zVariable. |
| zMode | A mode of the CDF library in which rVariables are made to appear as zVariables (and rEntries appear as zEntries). |
| zVariable | "Z" variable. A CDF object in which data values are stored. zVariables can have dimensionalities that are different than those of the rVariables (and each other). |

# Index


-

-0.0 to 0.0 mode, 29

## A

ALPHAOSF1_DECODING, 111, 218
ALPHAOSF1_ENCODING, 110, 218
ALPHAVMSd_DECODING, 111, 218
ALPHAVMSd_ENCODING, 110, 217
ALPHAVMSg_DECODING, 111, 218
ALPHAVMSg_ENCODING, 110, 217
assumed scope, 54
attributes, 11, 53
    creating, 53, 122, 157, 230, 267
    current, 149, 259
        confirming, 170, 280
        selecting
            by name, 164, 274
            by number, 164, 274
    entries, 11, 55
        accessing, 55, 114, 221
        current, 149–150, 259
            confirming, 171, 281
            selecting
                by name, 165, 275
                by number, 164–165, 274–275
        data specification, 55
            changing, 128, 180, 237, 290–291
            data type, 55
                inquiring, 126, 196–197, 235, 308–309
            number of elements, 55
                inquiring, 126, 196–198, 235, 308–309
        existence, determining, 171–172, 281–282
        gEntries, 11, 55
        maximum
            inquiring, 125, 195–196, 233, 306–307
        number of
            inquiring, 195–196, 307
        numbering, 55, 108, 216
        reading, 126, 130, 196–198, 235, 238, 308–310
        rEntries, 11, 55

writing, 128, 181–182, 237, 291–292
zEntries, 11, 55
existence, determining, 170, 281
gAttributes, 11, 54
naming, 39, 53, 114, 122, 221, 230
case sensitivity, 53
inquiring, 125, 194, 233, 306
renaming, 124, 180, 232, 290
trailing blanks, 54
number of
inquiring, 118, 184, 226, 295
numbering, 54, 63, 108, 216
assigning, 54
inquiring, 123, 195, 231, 306
scopes, 54
assumed, 54, 65, 112, 220
correcting, 54, 68, 131, 179, 240, 290
occurring, 54
constants, 112, 220
GLOBAL_SCOPE, 112, 220
GLOBAL_SCOPE_ASSUMED, 112, 220
VARIABLE_SCOPE, 112, 220
VARIABLE_SCOPE_ASSUMED, 112, 220
definite, 112, 220
global, 54
inquiring, 125, 194, 233, 306
purpose, 54
restrictions, 54
variable, 54
vAttributes, 11, 54

# B

big-endian, 35

# C

C programming interface, 101, 114
compiling, 101
customizing, 208
linking, 103
summary, 359
typedef's, 108
CDFid, 108
CDFstatus, 108
caching scheme, files, 29
CATDESC attribute, 57

CDF
    defined, 1
CDF distribution, 77
CDF library, 2, 25
    caching scheme, 29
        selecting, 159, 161, 164, 166, 168, 170, 269, 271, 274, 276, 278, 280
    copyright notice
        length, 114, 221
        reading, 184, 294
    interfaces, 12, 25
    limits, 29
        open CDFs, 29
    modes, 27
        -0.0 to 0.0, 29
            confirming, 166, 276
            constants, 113, 220
                NEGtoPOSfp0off, 113, 221
                NEGtoPOSfp0on, 113, 221
            selecting, 113, 159, 220, 269
        decoding, 36
            confirming, 166, 276
            constants, 110, 218
                HOST_DECODING, 111, 218
                NETWORK_DECODING, 111, 218
                SUN_DECODING...VAX_DECODING, 111, 218–219
            performance considerations, 37
            selecting, 159, 269
        read-only, 27, 96
            confirming, 165, 276
            constants, 113, 220
                READONLYoff, 113, 220
                READONLYon, 113, 220
            selecting, 113, 159, 220, 269
        zMode, 27, 63, 66, 71, 77, 83, 97
            confirming, 166, 276
            constants, 113, 220
                zMODEoff, 113, 220
                zMODEon1, 113, 220
                zMODEon2, 113, 220
            example, 28
            selecting, 28, 113, 159, 220, 269
            zMode/1, 28
            zMode/2, 28
    shareable, 106, 213
    version
        inquiring, 185–186, 296–297
CDF toolkit, 12, 59
    command line syntax, 59
    default settings, 60
    executable names, 59, 62, 65, 75, 82, 85
CDF$INC, 101, 209

CDF$LIB, 103, 211
cdf.h, 101, 108
cdf.inc, 209, 216
CDF_attr_correct_scope, 240
CDF_attr_create, 230
CDF_attr_entry_inquire, 235
CDF_attr_get, 238
CDF_attr_inquire, 233
CDF_ATTR_NAME_LEN, 53, 114, 221
CDF_attr_num, 231
CDF_attr_put, 237
CDF_attr_rename, 232
CDF_close, 227
CDF_COPYRIGHT_LEN, 114, 221
CDF_create, 221
CDF_delete, 228
CDF_doc, 224
CDF_EPOCH, 38
CDF_error, 229, 317, 351
CDF_INC, 101, 209
CDF_inquire, 226
CDF_LIB, 103, 211
CDF_lib, 221, 255
CDF_MAX_DIMS, 113, 221
CDF_OK, 108, 216
CDF_open, 223
CDF_PATHNAME_LEN, 113, 221
CDF_STATUSTEXT_LEN, 114, 221
CDF_var_close, 254
CDF_var_create, 241
CDF_var_get, 249
CDF_var_hyper_get, 252
CDF_var_hyper_put, 250
CDF_var_inquire, 246
CDF_VAR_LEN, 114, 221
CDF_VAR_NAME_LEN, 41
CDF_var_num, 243
CDF_var_put, 247
CDF_var_rename, 245
CDF_WARN, 108, 216
CDFattrCorrectScope, 131
CDFattrCreate, 122
CDFattrEntryInquire, 126
CDFattrGet, 130
CDFattrInquire, 125
CDFattrNum, 123
CDFattrPut, 128
CDFattrRename, 124
CDFbrowse, 87
CDFclose, 119
CDFcompare, 61

executing, 61
output, 64
CDFconvert, 33, 36, 64
executing, 64
output, 67
CDFcreate, 114
CDFdelete, 120
CDFdir, 60
executing, 60
output, 61
CDFdoc, 117
CDFedit, 96
executing, 96
interaction with, 98
window types, 98
CDFerror, 121, 206, 351
CDFid (typedef), 108
CDFinquire, 74, 118
executing, 75
output, 78
CDFlib, 114, 146
CDFlist, 87
executing, 87
interaction with, 89
cdfmsf.inc, 210
CDFopen, 116
CDFs
accessing, 27, 30, 116, 120, 223, 228
browsing, 96, 113, 220
closing, 31, 115–116, 119, 158, 222, 224, 227, 268
comparing, 61
converting, 64
copyright notice, 0, 74, 117, 224
length, 114, 221
reading, 117, 183, 224, 294
corrupted, 114, 120, 222, 228
creating, 31, 114, 154, 221, 264
current, 149, 258
confirming, 165, 275
selecting, 158, 268
deleting, 66, 86, 114, 120, 158, 222, 228, 268
encoding, 3, 33
changing, 33, 64, 172, 282
constants, 110, 217
HOST_ENCODING, 110, 217
NETWORK_ENCODING, 110, 217
SUN_ENCODING...VAX_ENCODING, 110, 217–218
default, 155, 265
equivalent, 35
host, 34
inquiring, 118, 183, 226, 294

network, 34
performance considerations, 36
files
directory listing, 60
extensions, 32
filtering, 87
format, 2, 32
changing, 32, 64, 172, 282
constants, 109, 216
MULTI_FILE, 109, 216
SINGLE_FILE, 109, 216
default, 32, 155, 265
inquiring, 183, 294
multi-file, 32
performance considerations, 33
single-file, 32
inquiring, 74
limits, 39
listing, 87
modifying, 27, 71, 96
naming, 31, 39, 85, 113, 115–116, 221–223
trailing blanks, 31
wildcards, 59
opening, 31, 116, 157, 223, 267
overwriting, 114, 222
skeleton, 65
statistics for, 69
subsetting, 87
updating, 70
verifying, 61
version, 117, 224
inquiring, 117, 185, 224, 296
CDFscope, 68, 132, 240
executing, 68
interaction with, 68
output, 69
CDFskeleton, 32, 34, 82, 84, 321
executing, 85
CDFstats, 69
executing, 70
output, 72
CDFstatus (typedef), 108, 114
CDFvarClose, 145
CDFvarCreate, 133
CDFvarGet, 140
CDFvarHyperGet, 143
CDFvarHyperPut, 141
CDFvarInquire, 137
CDFvarNum, 135
CDFvarPut, 138
CDFvarRename, 136

CDFwalk, 92
    executing, 92
    interaction with, 94
COL_MAJOR, 111, 219
compiling
    C applications, 101
    Fortran applications, 209
compute_EPOCH, 317
computeEPOCH, 206
computers, supported, 379
conceptual view, 1

# D

data specification
    attribute entry, 55
    variable, 44
data types, 37
    character, 38, 45
    constants, 109, 216
        CDF_BYTE...CDF_EPOCH, 109, 216–217
    EPOCH, 38, 206, 317
    equivalent, 39, 44
    floating point, 38
        -0.0, 38, 63, 67, 72, 77, 83, 98
    integer, 37
    non-character, 45
    number of elements, 38
decoding, CDF, 36
DECSTATION_DECODING, 111, 218
DECSTATION_ENCODING, 110, 218
definitions file, 26, 101, 103, 209, 211
DEFINITIONS.COM, 101, 103, 209, 211
definitions.csh, 101, 103, 209, 211
definitions.ksh, 101, 103, 209, 211
definitions.sh, 101, 103, 209, 211
definitions.tcsh, 101, 103, 209, 211
dimensionality, variable, 41
dimensions
    limit, 39, 113, 221
    numbering, 108, 216

# E

encode_EPOCH, 318
encode_EPOCH1, 318
encode_EPOCH2, 318

encodeEPOCH, 207
encodeEPOCH1, 207
encodeEPOCH2, 207
encoding, CDF, 33
EPOCH, 38
    breaking down, 206, 317
    computing, 206, 317
    encoding, 207, 318
    parsing, 207, 318
    syntax, 39
    utility routines, 114, 206, 317
        C applications
            computeEPOCH, 206
            encodeEPOCH, 207
            encodeEPOCH1, 207
            encodeEPOCH2, 207
            EPOCHbreakdown, 206
            parseEPOCH, 207
        Fortran applications
            compute_EPOCH, 317
            encode_EPOCH, 318
            encode_EPOCH1, 318
            encode_EPOCH2, 318
            EPOCH_breakdown, 317
            parse_EPOCH, 318
EPOCH variable, 57
EPOCH_breakdown, 317
EPOCHbreakdown, 206
examples
    C applications
        closing
            CDF, 120
            rVariable, 145
        correcting
            attribute scope, 132
        creating
            attribute, 122
            CDF, 115, 146
            rVariable, 134, 198
            zVariable, 199
        deleting
            CDF, 121
        inquiring
            attribute, 125
                entry, 127
                number, 124
            CDF, 117, 119
                format, 208
            explanation text, 121
            rVariable, 138
                number, 135

Internal Interface, 146, 198
interpreting
status codes, 205
opening
CDF, 116
program, 14
reading
attribute entry, 130
rVariable values
hyper, 144, 200
single, 140
zVariable values
sequential, 201
renaming
attribute, 124, 201
rVariable, 136
status handler, 205
writing
attribute
gEntry, 129
rEntry, 129, 202
rVariable values
hyper, 142
single, 139
zVariable values
multiple variable, 203
conceptual view, 9
data set, flat, 8
Fortran applications
closing
CDF, 228
rVariable, 255
correcting
attribute scope, 240
creating
attribute, 231
CDF, 223, 256
rVariable, 242, 310
zVariable, 311
deleting
CDF, 229
inquiring
attribute, 234
entry, 236
number, 232
CDF, 225, 227
explanation text, 229
rVariable, 247
number, 244
Internal Interface, 256, 310
interpreting

status codes, 316
opening
CDF, 224
program, 21
reading
attribute entry, 239
rVariable values
hyper, 254, 312
single, 249
zVariable values
sequential, 313
renaming
attribute, 233, 313
rVariable, 245
status handler, 316
writing
attribute
gEntry, 237
rEntry, 237, 314
rVariable values
hyper, 251
single, 248
zVariable values
multiple variable, 314
physical view, 10
skeleton table, 19, 333
extend records, variable, 44

# F

FIELDNAM attribute, 56
FILLVAL attribute, 57, 69, 86
FORMAT attribute, 57, 71, 77, 83, 97
format, CDF, 32
Fortran programming interface, 209, 221
argument passing, 215
compiling, 209
linking, 211
summary, 369
function prototypes, 114

# G

GLOBAL_SCOPE, 112, 220
GLOBAL_SCOPE_ASSUMED, 112, 220

# H

host decoding, 37
host encoding, 34
HOST_DECODING, 111, 218
HOST_ENCODING, 110, 217
HP_DECODING, 111, 218
HP_ENCODING, 110, 218
hyper access, variable, 48

# I

IBMPC_DECODING, 111, 218
IBMPC_ENCODING, 110, 218
IBMRS_DECODING, 111, 218
IBMRS_ENCODING, 110, 218
IDL
    CDF's interface, 27, 339
IEEE 754, 29, 35, 38
include files, 101, 209
indexing, variable records, 32
initial records, variable, 42
interfaces, 12, 25
    IDL, 27, 339
    Internal, 13, 25, 146, 255
    mixing, 114, 221
    Obsolete, 26
    Standard, 13, 25
Internal Interface, 13, 25, 114, 146, 221, 255
    common mistakes, 204
    current objects/states, 149, 258
        attribute, 149, 259
        attribute entries, 149–150, 259
        CDF, 149, 258
        preselected, required, 154, 264
        records/dimensions, 150–152, 260–261
        sequential value, 151–152, 260, 262
        status code, 152, 262
        variables, 149, 259
    examples, 146, 198, 256, 310
    indentation/style, 153, 262
    items, 154, 264
    operations, 154, 264
        CLOSE_, 158, 268
        CONFIRM_, 165, 275
        CREATE_, 154, 264
        DELETE_, 158, 268
        GET_, 182, 293
        OPEN_, 157, 267

PUT_, 172, 282
SELECT_, 158, 268
status codes, returned, 152, 262
syntax, 153, 263
argument list, 153, 263
limitations, 153, 263
item referencing, 108, 216

## L

libcdf.a, 103, 211
libcdf.lib, 103, 211
LIBCDF.OLB, 103, 211
limits, 29, 39, 113, 221
attribute names, 114, 221
copyright text, 114, 221
dimensions, 39, 113, 221
explanation text, 114, 221
IBM PC, 113, 221
Internal Interface, 153, 263
open CDFs, 29
pathnames, 113, 221
variable names, 39, 114, 221
linking
C applications, 103
Fortran applications, 211
Obsolete Interface, 26
little-endian, 35

## M

majority
variable, 46
memory models (MS-DOS), 102, 105, 211, 213
MODS attribute, 57
MONOTON attribute, 57, 70
multi-file format, 32
MULTI_FILE, 109, 216
multiple variable access, 51

## N

NEGtoPOSfp0off, 113, 221
NEGtoPOSfp0on, 113, 221
network encoding, 34

NETWORK_DECODING, 111, 218
NETWORK_ENCODING, 110, 217
NeXT_DECODING, 111, 219
NeXT_ENCODING, 110, 218
NOVARY, 112, 219
NSSDC Standard, 56

## O

Obsolete Interface, 26
operating systems, supported, 379

## P

pad values, variable, 53
parse_EPOCH, 318
parseEPOCH, 207
performance considerations
    decoding, 37
    encoding, 36
    format, 33
    majority, 47
CDFbrowse, 96

## R

read-only mode, 27
READONLYoff, 113, 220
READONLYon, 113, 220
release notes, 379
ROW_MAJOR, 111, 219

## S

SCALEMAX attribute, 57, 70
SCALEMIN attribute, 57, 70
scope, attribute, 54
sequential access, variable, 50
SGi_DECODING, 111, 218
SGi_ENCODING, 110, 217
single-file format, 32
SINGLE_FILE, 109, 216
skeleton CDF, 85

skeleton table, 32, 82, 84
creating, 82, 86
example, 333
file extension, 86
format, 321
SkeletonTable, 82, 86
executing, 82
output, 84
SkeletonTable, 321
Standard Interface, 13, 25, 114, 221
standards
NSSDC, 56
attributes, 56
variables, 57
status codes, 351
constants, 108, 216, 351
CDF_OK, 108, 216
CDF_WARN, 108, 216
current, 152, 262
selecting, 158, 268
displaying, 63, 67, 69, 72, 77, 84, 86, 98
error, 351
explanation text, 351
inquiring, 121, 198, 229, 310, 351
length, 114, 221
informational, 351
interpreting, 204, 315, 351
returned, 114, 221
status handler, example, 205, 316
warning, 351
SUN_DECODING, 111, 218
SUN_ENCODING, 110, 217

# T

TEXT attribute, 57
TITLE attribute, 56
trailing blanks
attribute names, 54
CDF pathnames, 31
variable names, 41

# U

UNITS attribute, 57

# V


VALIDMAX attribute, 56, 69
VALIDMIN attribute, 56, 69
VARIABLE_SCOPE, 112, 220
VARIABLE_SCOPE_ASSUMED, 112, 220
variables, 2, 7, 40
    accessing, 6, 40, 114, 221
        hyper values, 48
            dimension counts, 48
                current, 150–151, 260–261
                    confirming, 167, 169, 277, 279
                    selecting, 160, 163, 270, 273
            dimension indices, starting, 48
                current, 150–151, 260–261
                    confirming, 167, 169, 277, 279
                    selecting, 160, 163, 270, 273
            dimension intervals, 48
                current, 150, 152, 260–261
                    confirming, 167, 170, 278, 280
                    selecting, 161, 163, 271, 273
            example, 49
            reading, 47, 143, 189, 193, 252, 300, 304
            record count, 48
                current, 150–151, 260–261
                    confirming, 167, 169, 277, 279
                    selecting, 160, 162, 270, 272
            record interval, 48
                current, 150–151, 260–261
                    confirming, 167, 169, 277, 279
                    selecting, 160, 162, 270, 272
            record number, starting, 48
                current, 150–151, 260–261
                    confirming, 167, 169, 277, 279
                    selecting, 160, 162, 270, 272
            writing, 47, 141, 175, 178, 250, 285, 289
        multiple variable, 51
            reading, 189, 193, 301, 305
            record numbers, 51
                current, 150–151, 260–261
                    selecting, 162, 272
            variable count, 51
            variable list, 51
            writing, 175, 179, 286, 289
        sequential values, 47, 50
            current value, 50, 151–152, 260, 262
                confirming, 168, 170, 278, 280
                selecting, 161, 163, 271, 273
            example, 50
            reading, 189, 193, 300, 305
            writing, 175, 178, 285, 289

single values, 47
dimension indices, 47
current, 150–151, 260–261
confirming, 167, 169, 277, 279
selecting, 160, 163, 270, 273
reading, 140, 189, 193, 249, 299, 304
record number, 47
current, 150–151, 260–261
confirming, 167, 169, 277, 279
selecting, 160, 162, 270, 272
writing, 138, 174, 178, 247, 285, 288
arrays, 41, 46
closing, 40, 145, 158, 254, 268
creating, 40, 133, 155–156, 241, 265–266
current, 149, 259
confirming, 166, 168, 276, 278
selecting
by name, 160, 162, 270, 272
by number, 159, 162, 269, 271
data specification, 44
changing, 44, 173, 176, 283, 286
data type, 44–45
inquiring, 137, 186, 190, 246, 297, 301
number of elements, 44–45
inquiring, 137, 186, 190, 246, 297, 302
selecting, 44
dimensionality, 40–41
inquiring, 118, 182–183, 226, 293–294
existence, determining, 166, 168, 277–278
fill values, 57, 69
indices
numbering, 108, 216
majority, 46
changing, 47, 64, 172, 282
considering, 111–112, 219
constants, 111, 219
`COL_MAJOR`, 111, 219
`ROW_MAJOR`, 111, 219
default, 155, 265
example, 46
inquiring, 118, 183, 226, 294
selecting, 47
naming, 39, 41, 114, 133, 221, 242
case sensitivity, 41
inquiring, 137, 186, 190, 246, 297, 301
renaming, 136, 173, 176, 245, 283, 286
trailing blanks, 41
non-record-variant (NRV), 45, 83
number of, inquiring, 118, 184, 226, 295
numbering, 41, 63, 108, 216
assigning, 41

inquiring, 135, 187, 191, 243, 298, 302
opening, 40
pad values, 42, 45, 49, 53, 57, 86
default, 42, 53
inquiring, 187, 191, 298, 303
specifying, 53, 174, 177, 284, 287
when used, 53
record-variant (RV), 45
records, 41
allocated, 44
inquiring, 188, 192, 299, 303
contiguous, 43–44
extend, 44
inquiring, 187, 192, 298, 303
specifying, 174, 178, 284, 288
indexing, 32
inquiring, 188, 192, 299, 303–304
initial, 42
writing, 174, 177, 284, 288
maximum, 42
inquiring, 118, 184–185, 188, 192, 226, 295–296, 299, 303
numbering, 42, 108, 216
physical, 41, 45
virtual, 41, 45
rVariables, 7, 40
subarrays, 41, 45
values
monotonicity, 57, 70
virtual, 46
variances, 8, 48
changing, 45–46
constants, 112, 219
NOVARY, 112, 219
VARY, 112, 219
dimensional, 45
changing, 173, 177, 283, 287
example, 46
inquiring, 137, 187, 191, 246, 298, 302
record, 42, 45
changing, 173, 177, 283, 287
example, 42
inquiring, 137, 187, 191, 246, 297, 302
selecting, 45–46
zVariables, 10, 40
variance
dimensional, 45
record, 45
VARY, 112, 219
VAX_DECODING, 111, 218
VAX_ENCODING, 110, 217

## W

wildcard characters, 59

## X

XDR, 34

## Z

zMode, 27
`zMODEoff`, 113, 220
`zMODEon1`, 113, 220
`zMODEon2`, 113, 220