Copyright © 2013
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 (U.S.A.)

This software may be copied or redistributed as long as it is not sold for profit, but it can be incorporated into any other substantive product with or without modifications for profit or non-profit. If the software is modified, it must include the following notices:

- The software is not the original (for protection of the original author’s reputations from any problems introduced by others)
- Change history (e.g. date, functionality, etc.)

This Copyright notice must be reproduced on each copy made. This software is provided as is without any express or implied warranties whatsoever.

Internet: gsfc-cdf-support@lists.nasa.gov
## Contents

1 **Compiling** ..................................................................................................................... 1

   1.1 Specifying cdf.h Location in the Compile Command ....................................................... 1
      1.1.1 OpenVMS Systems .................................................................................................. 1
      1.1.2 UNIX Systems (including Mac OS X) ...................................................................... 2
   1.2 Specifying cdf.h Location in the Source File .................................................................. 3

2 **Linking** .......................................................................................................................... 5

   2.1 OpenVMS Systems ......................................................................................................... 5
      2.1.1 Combining the Compile and Link ............................................................................. 5
   2.2 Windows NT/2000/XP SYSTEMS, Microsoft Visual C++ or Microsoft Visual C++.NET ........... 6

3 **Linking Shared CDF Library** .......................................................................................... 7

   3.1 DEC VAX & Alpha (OpenVMS) ....................................................................................... 7
   3.2 SUN (Solaris) ................................................................................................................. 8
   3.3 HP 9000 (HP-UX) ......................................................................................................... 8
   3.4 IBM RS6000 (AIX) ........................................................................................................ 8
   3.5 DEC Alpha (OSF/1) ....................................................................................................... 9
   3.6 SGI (IRIX 6.x) .............................................................................................................. 9
   3.7 Linux (PC & Power PC) .................................................................................................. 9
   3.8 Windows (NT/2000/XP) ............................................................................................. 9
   3.9 Macintosh OS X ........................................................................................................... 9

4 **Programming Interface** ................................................................................................. 11

   4.1 Item Referencing .......................................................................................................... 11
   4.2 Defined Types .............................................................................................................. 11
   4.3 CDFstatus Constants .................................................................................................. 11
   4.4 CDF Formats .............................................................................................................. 12
   4.5 CDF Data Types .......................................................................................................... 12
   4.6 Data Encodings .......................................................................................................... 13
   4.7 Data Decodings .......................................................................................................... 14
   4.8 Variable Majorities .................................................................................................... 15
   4.9 Record/Dimension Variances .................................................................................... 15
   4.10 Compressions ........................................................................................................... 16
   4.11 Sparseness ............................................................................................................... 16
      4.11.1 Sparse Records ................................................................................................... 17
      4.11.2 Sparse Arrays .................................................................................................... 17
   4.12 Attribute Scopes ...................................................................................................... 17
   4.13 Read-Only Modes ...................................................................................................... 17
   4.14 zModes ..................................................................................................................... 18
   4.15 -0.0 to 0.0 Modes ....................................................................................................... 18
   4.16 Operational Limits .................................................................................................. 18
   4.17 Limits of Names and Other Character Strings ........................................................... 18
   4.18 Backward File Compatibility with CDF 2.7 ................................................................. 19
   4.19 Checksum ................................................................................................................ 20
   4.20 Data Validation ........................................................................................................ 22
   4.21 8-Byte Integer ......................................................................................................... 23
5 Standard Interface ........................................................................................................... 25

5.1 CDFattrCreate .................................................................................................................. 25
  5.1.1 Example(s) .................................................................................................................. 26
5.2 CDFattrEntryInquire ......................................................................................................... 26
  5.2.1 Example(s) .................................................................................................................. 27
5.3 CDFattrGet ......................................................................................................................... 28
  5.3.1 Example(s) .................................................................................................................. 29
5.4 CDFattrInquire ................................................................................................................... 29
  5.4.1 Example(s) .................................................................................................................. 30
5.5 CDFattrNum ....................................................................................................................... 31
  5.5.1 Example(s) .................................................................................................................. 31
5.6 CDFattrPut ........................................................................................................................ 32
  5.6.1 Example(s) .................................................................................................................. 32
5.7 CDFattrRename ................................................................................................................. 33
  5.7.1 Example(s) .................................................................................................................. 34
5.8 CDFclose .......................................................................................................................... 34
  5.8.1 Example(s) .................................................................................................................. 34
5.9 CDFcreate ........................................................................................................................ 35
  5.9.1 Example(s) .................................................................................................................. 36
5.10 CDFdelete ......................................................................................................................... 36
  5.10.1 Example(s) ................................................................................................................ 37
5.11 CDFdoc ........................................................................................................................... 37
  5.11.1 Example(s) ................................................................................................................ 38
5.12 CDFerror ........................................................................................................................ 38
  5.12.1 Example(s) ................................................................................................................ 38
5.13 CDFgetVarsRecordData ................................................................................................. 39
  5.13.1 Example(s) ................................................................................................................ 40
5.14 CDFgetzVarsRecordData ............................................................................................... 41
  5.14.1 Example(s) ................................................................................................................ 41
5.15 CDFinquire ...................................................................................................................... 42
  5.15.1 Example(s) ................................................................................................................ 43
5.16 CDFopen ........................................................................................................................ 44
  5.16.1 Example(s) ................................................................................................................ 44
5.17 CDFputVarsRecordData ................................................................................................. 45
  5.17.1 Example(s) ................................................................................................................ 45
5.18 CDFputzVarsRecordData ............................................................................................... 47
  5.18.1 Example(s) ................................................................................................................ 47
5.19 CDFvarClose .................................................................................................................... 48
  5.19.1 Example(s) ................................................................................................................ 49
5.20 CDFvarCreate ................................................................................................................ 49
  5.20.1 Example(s) ................................................................................................................ 50
5.21 CDFvarGet ....................................................................................................................... 51
  5.21.1 Example(s) ................................................................................................................ 51
5.22 CDFvarHyperGet ........................................................................................................... 52
  5.22.1 Example(s) ................................................................................................................ 52
5.23 CDFvarHyperPut ............................................................................................................. 53
  5.23.1 Example(s) ................................................................................................................ 54
5.24 CDFvarInquire ............................................................................................................... 54
  5.24.1 Example(s) ................................................................................................................ 55
5.25 CDFvarNum ...................................................................................................................... 56
  5.25.1 Example(s) ................................................................................................................ 56
5.26 CDFvarPut ....................................................................................................................... 57
  5.26.1 Example(s) ................................................................................................................ 57
5.27 CDFvarRename ................................................................................................................ 58
  5.27.1 Example(s) ................................................................................................................ 58
6  Exended Standard Interface ........................................61

6.1 Library Information ..................................................61
  6.1.1 CDFgetDataTypeSize ........................................61
  6.1.2 CDFgetLibraryCopyright .....................................62
  6.1.3 CDFgetLibraryVersion .......................................63
  6.1.4 CDFgetStatusText ...........................................64

6.2 CDF .................................................................64
  6.2.1 CDFcloseCDF ................................................65
  6.2.2 CDFcreateCDF ...............................................65
  6.2.3 CDFdeleteCDF ...............................................67
  6.2.4 CDFgetCacheSize ...........................................67
  6.2.5 CDFgetChecksum ............................................68
  6.2.6 CDFgetCompression .........................................69
  6.2.7 CDFgetCompressionCacheSize ...............................70
  6.2.8 CDFgetCompressionInfo ....................................71
  6.2.9 CDFgetCopyright ............................................72
  6.2.10 CDFgetDecoding ..........................................72
  6.2.11 CDFgetEncoding ..........................................73
  6.2.12 CDFgetFileForward .......................................74
  6.2.13 CDFgetFormat .............................................74
  6.2.14 CDFgetMajority ...........................................75
  6.2.15 CDFgetName .................................................76
  6.2.16 CDFgetNegtoPosfp0Mode ................................77
  6.2.17 CDFgetReadOnlyMode .....................................77
  6.2.18 CDFgetStageCacheSize ...................................78
  6.2.19 CDFgetValidate ...........................................79
  6.2.20 CDFgetVersion ............................................80
  6.2.21 CDFgetzMode ..............................................80
  6.2.22 CDFinquireCDF ...........................................81
  6.2.23 CDFopenCDF ...............................................83
  6.2.24 CDFsetCacheSize .........................................84
  6.2.25 CDFsetChecksum ...........................................84
  6.2.26 CDFsetCompression ........................................85
  6.2.27 CDFsetCompressionCacheSize ..............................86
  6.2.28 CDFsetDecoding ..........................................87
  6.2.29 CDFsetEncoding ..........................................88
  6.2.30 CDFsetFileForward .......................................88
  6.2.31 CDFsetFormat ...............................................89
  6.2.32 CDFsetMajority ...........................................90
  6.2.33 CDFsetNegtoPosfp0Mode ................................91
  6.2.34 CDFsetReadOnlyMode .....................................91
  6.2.35 CDFsetStageCacheSize ...................................92
  6.2.36 CDFsetValidate ...........................................93
  6.2.37 CDFsetzMode ...............................................93

6.3 Variable ..........................................................94
  6.3.1 CDFclosezVar ...............................................94
  6.3.2 CDFconfirmzVarExistence ..................................95
  6.3.3 CDFconfirmzVarPadValueExistence ........................96
  6.3.4 CDFcreatezVar ...............................................97
  6.3.5 CDFdeletezVar ...............................................99
  6.3.6 CDFdeletezVarRecords .....................................100
  6.3.7 CDFgetMaxWrittenRecNums ................................101
  6.3.8 CDFgetNumrVars ............................................101
  6.3.9 CDFgetNumzVars ............................................102
  6.3.10 CDFgetVarAllRecordsByVarName ..........................103
6.3.11 CDFgetVarNum .................................................................104
6.3.12 CDFgetVarRangeRecordsByVarName ..................................106
6.3.13 CDFgetzVarAllocRecords ..................................................107
6.3.14 CDFgetzVarAllRecordsByVarID .........................................108
6.3.15 CDFgetzVarBlockingFactor ..............................................109
6.3.16 CDFgetzVarCacheSize .....................................................110
6.3.17 CDFgetzVarCompression ..................................................111
6.3.18 CDFgetzVarData ..............................................................112
6.3.19 CDFgetzVarDataType .......................................................113
6.3.20 CDFgetzVarDimSizes .......................................................114
6.3.21 CDFgetzVarDimVariances ...............................................115
6.3.22 CDFgetzVarMaxAllocRecNum .........................................116
6.3.23 CDFgetzVarMaxWrittenRecNum .......................................116
6.3.24 CDFgetzVarName ............................................................117
6.3.25 CDFgetzVarNumDims ......................................................118
6.3.26 CDFgetzVarNumElements ...............................................119
6.3.27 CDFgetzVarNumRecsWritten ..........................................120
6.3.28 CDFgetzVarPadValue .....................................................120
6.3.29 CDFgetzVarRangeRecordsByVarID ....................................121
6.3.30 CDFgetzVarRecordData ..................................................123
6.3.31 CDFgetzVarRecVariance ................................................124
6.3.32 CDFgetzVarReservePercent ..........................................125
6.3.33 CDFgetzVarSeqData .......................................................125
6.3.34 CDFgetzVarSeqPos ........................................................126
6.3.35 CDFgetzVarsMaxWrittenRecNum .....................................127
6.3.36 CDFgetzVarSparseRecords ............................................128
6.3.37 CDFgetzVarSparseRecordsByVarName .................................129
6.3.38 CDFhyperGetzVarData ....................................................131
6.3.39 CDFhyperPutzVarData ....................................................132
6.3.40 CDFinquirezVar ............................................................134
6.3.41 CDFinsertzVarRecordsByVarID .......................................135
6.3.42 CDFinsertzVarRecordsByVarName ....................................137
6.3.43 CDFinsertzVarRecordsByVarID .......................................138
6.3.44 CDFputzVarAllRecordsByVarName ....................................139
6.3.45 CDFputzVarRangeRecordsByVarName ..................................140
6.3.46 CDFputzVarAllRecordsByVarID .......................................141
6.3.47 CDFputzVarData ............................................................142
6.3.48 CDFputzVarRangeRecordsByVarID ....................................143
6.3.49 CDFputzVarRecordData ..................................................144
6.3.50 CDFputzVarSeqData ........................................................145
6.3.51 CDFputzVarsRecordDatabyNumbers ...................................146
6.3.52 CDFrenamezVar ...........................................................148
6.3.53 CDFsetzVarAllocBlockRecords .......................................149
6.3.54 CDFsetzVarAllocRecords ...............................................150
6.3.55 CDFsetzVarBlockingFactor ............................................150
6.3.56 CDFsetzVarCacheSize ....................................................151
6.3.57 CDFsetzVarCompression ................................................152
6.3.58 CDFsetzVarDataSpec ........................................................153
6.3.59 CDFsetzVarDimVariances ...............................................154
6.3.60 CDFsetzVarInitialRecs ...................................................155
6.3.61 CDFsetzVarPadValue .....................................................156
6.3.62 CDFsetzVarRecVariance ................................................156
6.3.63 CDFsetzVarReservePercent ..........................................157
6.3.64 CDFsetzVarsCacheSize ..................................................158
6.3.65 CDFsetzVarSeqPos ........................................................159
6.3.66 CDFsetzVarSparseRecords .............................................160
6.4 Attributes/Entries ........................................................................................................161
  6.4.1 CDFconfirmAttrExistence ..................................................................................161
  6.4.2 CDFconfirmgEntryExistence..............................................................................161
  6.4.3 CDFconfirmrEntryExistence .............................................................................162
  6.4.4 CDFconfirmzEntryExistence .............................................................................163
  6.4.5 CDFcreateAttr ....................................................................................................164
  6.4.6 CDFdeleteAttr ....................................................................................................165
  6.4.7 CDFdeleteAttrgEntry .......................................................................................166
  6.4.8 CDFdeleteAttrrEntry .......................................................................................167
  6.4.9 CDFdeleteAttrzEntry .......................................................................................168
  6.4.10 CDFgetAttrgEntry ..........................................................................................168
  6.4.11 CDFgetAttrgEntryDataType ..........................................................................170
  6.4.12 CDFgetAttrgEntryNumElements ...................................................................171
  6.4.13 CDFgetAttrrEntry ..........................................................................................172
  6.4.14 CDFgetAttrMaxgEntry ..................................................................................173
  6.4.15 CDFgetAttrMaxrEntry ..................................................................................174
  6.4.16 CDFgetAttrMaxzEntry ..................................................................................175
  6.4.17 CDFgetAttrName ...........................................................................................176
  6.4.18 CDFgetAttrNum .............................................................................................177
  6.4.19 CDFgetAttrrEntryDataType ..........................................................................177
  6.4.20 CDFgetAttrrEntryNumElements ....................................................................178
  6.4.21 CDFgetAttrScope ...........................................................................................179
  6.4.22 CDFgetAttrzEntry ..........................................................................................180
  6.4.23 CDFgetAttrzEntryDataType ..........................................................................182
  6.4.24 CDFgetAttrzEntryNumElements ...................................................................183
  6.4.25 CDFgetNumAttrgEntries ...............................................................................184
  6.4.26 CDFgetNumAttributes ....................................................................................185
  6.4.27 CDFgetNumAttrrEntries ...............................................................................185
  6.4.28 CDFgetNumAttrzEntries ...............................................................................186
  6.4.29 CDFgetNumgAttributes ..................................................................................187
  6.4.30 CDFgetNumrAttributes ..................................................................................188
  6.4.31 CDFinquireAttr ..............................................................................................189
  6.4.32 CDFinquireAttrgEntry ...................................................................................190
  6.4.33 CDFinquireAttrrEntry ...................................................................................192
  6.4.34 CDFinquireAttrzEntry ...................................................................................193
  6.4.35 CDFputAttrgEntry ........................................................................................194
  6.4.36 CDFputAttrrEntry ........................................................................................195
  6.4.37 CDFputAttrzEntry ........................................................................................197
  6.4.38 CDFrenameAttr ............................................................................................198
  6.4.39 CDFsetAttrgEntryDataType ..........................................................................198
  6.4.40 CDFsetAttrrEntryDataType ..........................................................................199
  6.4.41 CDFsetAttrScope ...........................................................................................200
  6.4.42 CDFsetAttrzEntryDataType ..........................................................................201

7 Internal Interface - CDFlib ...........................................................................................203
  7.1 Example(s) ...........................................................................................................203
  7.2 Current Objects/States (Items) ............................................................................205
  7.3 Returned Status .....................................................................................................209
  7.4 Indentation/Style ..................................................................................................209
  7.5 Syntax ....................................................................................................................209
  7.6 Operations ............................................................................................................210
  7.7 More Examples .....................................................................................................265
    7.7.1 rVariable Creation .........................................................................................265
    7.7.2 zVariable Creation (Character Data Type) .....................................................265
    7.7.3 Hyper Read with Subsampling .......................................................................266
8 Interpreting CDF Status Codes ................................................................. 273

9 EPOCH Utility Routines ............................................................................. 275
  9.1 computeEPOCH .................................................................................. 275
  9.2 EPOCHbreakdown .............................................................................. 276
  9.3 encodeEPOCH .................................................................................... 276
  9.4 encodeEPOCH1 .................................................................................. 277
  9.5 encodeEPOCH2 .................................................................................. 277
  9.6 encodeEPOCH3 .................................................................................. 277
  9.7 encodeEPOCH4 .................................................................................. 277
  9.8 encodeEPOCHx .................................................................................. 278
  9.9 parseEPOCH ...................................................................................... 279
  9.10 parseEPOCH1 ................................................................................... 279
  9.11 parseEPOCH2 ................................................................................... 279
  9.12 parseEPOCH3 ................................................................................... 279
  9.13 parseEPOCH4 ................................................................................... 279
  9.14 computeEPOCH16 ............................................................................ 280
  9.15 EPOCH16breakdown ....................................................................... 280
  9.16 encodeEPOCH16 ............................................................................. 281
  9.17 encodeEPOCH16_1 ........................................................................ 281
  9.18 encodeEPOCH16_2 ........................................................................ 281
  9.19 encodeEPOCH16_3 ........................................................................ 282
  9.20 encodeEPOCH16_4 ........................................................................ 282
  9.21 encodeEPOCH16_x ........................................................................ 282
  9.22 parseEPOCH16 ............................................................................... 283
  9.23 parseEPOCH16_1 ........................................................................... 284
  9.24 parseEPOCH16_2 ........................................................................... 284
  9.25 parseEPOCH16_3 ........................................................................... 284
  9.26 parseEPOCH16_4 ........................................................................... 284

10 TT2000 Utility Routines .......................................................... 287
  10.1 CDF_TT2000_from.UTC_parts ......................................................... 287
  10.2 CDF_TIME_to.UTC_parts ................................................................. 288
  10.3 CDF_TT2000_to.UTC_string ............................................................. 289
  10.4 CDF_TT2000_from.UTC_string ........................................................ 290
  10.5 CDF_TT2000_from.UTC_EPOCH ....................................................... 290
  10.6 CDF_TT2000_to.UTC_EPOCH ........................................................... 291
  10.7 CDF_TT2000_from.UTC_EPOCH16 .................................................. 291
  10.8 CDF_TT2000_to.UTC_EPOCH16 ....................................................... 291
Chapter 1

1 Compiling

Each source file that calls the CDF library or references CDF parameters must include cdf.h. On OpenVMS systems a logical name, CDFSINC, that specifies the location of cdf.h is defined in the definitions file, DEFINITIONS.COM, provided with the CDF distribution. On UNIX systems (including Mac OS X) an environment variable, CDF_INC, that serves the same purpose is defined in the definitions file definitions.<shell-type> where <shell-type> is the type of shell being used: C for the C-shell (csh and tcsh), K for the Korn (ksh), BASH, and POSIX shells, and B for the Bourne shell (sh). This section assumes that you are using the appropriate definitions file on those systems. The location of cdf.h is specified as described in the appropriate sections for those systems.

The CDF file’s offset and size in V 3.0 use the data type off_t (__int64 on Windows), instead of 32-bit long. One or certain predefined macros needs to be defined to the C compiler to make it 64-bit long.

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

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

1.1.1 OpenVMS Systems

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

    $ CC/INCLUDEFIDIRECTORY=CDFSINC/DEFINE=_LARGEFILE <source-name>

---

1 We use OFF_T to represent either off_t or __int64 as the 64-bit data type in the following section.
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. Use /DEFINE=_LARGEFILE to make OFF_T 64-bit long.

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 IEEE_FLOAT, you will also have to specify /FLOAT=IEEE_FLOAT on the CC command line in order to correctly process double-precision floating-point values.

1.1.2 UNIX Systems (including Mac OS X)

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

```
% cc -c -I${CDF_INC} -D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE
          -D_LARGEFILE_SOURCE <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 2.2.) The object module created will be named <source-name>.o. Note that in a "makefile" where CDF_INC is imported, $(CDF_INC) would be specified instead of ${CDF_INC}. The defined Macros, _FILE_OFFSET_BITS=64, _LARGEFILE64_SOURCE and _LARGEFILE_SOURCE, are needed to make the data type OFF_T 64-bit long.


An example of the command to compile a source file on Windows systems using Microsoft Visual C++ would be as follows. It is extracted from an NMAKE file, generated by Microsoft Visual C++, to compile the CDF library source code.

```
C:\> CL /c /nologo /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_FILE_OFFSET_BITS=64"
          /D "_LARGEFILE_SOURCE" /D "_LARGEFILE64_SOURCE" /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 file name of the directory containing cdf.h. You will need to know where on your system cdf.h has been installed. <inc-path> may be either an absolute or relative file name.

You may also need to specify the location of system include files. For Microsoft Visual C++ this is usually accomplished by setting MS-DOS environment variables, e.g., execute VCVARS32.BAT for VC++.

The /c option specifies that only an object module is to be produced. The object module will be named <source-name>.obj.

The /nologo option specifies that the Copyright message is suppressed.

The /W3 option specifies the warning level for compiling.

The /Gm option specifies that minimal rebuild is enabled.

---

2 You may not need to define these all three macros on a certain Unix platform. But defining all of them should work on all compilers that support 64-bit off_t data type.
The /GX option specifies that C++ EH is enabled.

The /ZI option specifies that edit/continue debug information is enabled.

The /Od option specifies that optimization is disabled.

WIN32, _FILE_OFFSET_BITS=64, _LARGEFILE_SOURCE and _LARGEFILE64_SOURCE are defined macros.

Consult the documents for Microsoft Visual C++ or contact gsfc-cdf-support@lists.nasa.gov for inquiries.

All distributed libraries (static and dynamic) as well as the executables for the toolkit programs for WIN32 are created by the Microsoft Visual C++.

## 1.2 Specifying cdf.h Location in the Source File

The second method involves specifying the file name 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:

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

where `<inc-path>` is the file name of the directory containing cdf.h. The source file would then be compiled as shown in Section 1.1 but without specifying the location of cdf.h on the command line (where applicable).

On OpenVMS systems CDFSINC: may be used for `<inc-path>`. On UNIX, MS-DOS, and Macintosh systems, `<inc-path>` must be a relative or absolute file name. (An environment variable may not be used for `<inc-path>` on UNIX systems.) You will need to know where on your system the cdf.h file has been installed. on Macintosh systems, file names are constructed by separating volume/folder names with colons.
Chapter 2

2 Linking

Your applications must be linked with the CDF library. Both the Standard and Internal interfaces for C applications are built into the CDF library. On OpenVMS systems, a logical name, CDFSLIB, which specifies the location of the CDF library, is defined in the definitions file, DEFINITIONS.COM, provided with the CDF distribution. On UNIX systems (including Mac OS X) an environment variable, CDF_LIB, which serves the same purpose, is defined in the definitions file definitions.<shell-type> where <shell-type> is the type of shell being used: C for the C-shell (csh and tsh), K for the Korn (ksh), BASH, and POSIX shells, and B for the Bourne shell (sh). This section assumes that you are using the appropriate definitions file on those systems. The location of the CDF library is specified as described in the appropriate sections for those systems.

2.1 OpenVMS Systems

An example of the command to link your application with the CDF library (LIBCDF.OLB) on DEC Alpha/OpenVMS systems would be as follows:

```bash
$ 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 VAXCRTLQD for a default of D_FLOAT or VAXCRTL for a default of IEEE_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.

UNIX Systems (including Mac OS X)
An example of the command to link your application with the CDF library (libcdf.a) on UNIX flavored systems would be as follows:

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

A shareable version of the CDF library is also available on Open/VMS and some flavors of UNIX. Its use is described in Chapter 3. A dynamic link library (DLL), LIBCDF.DLL, is available on Window NT/2000/XP. Consult the Microsoft documentation for details on using a DLL. Note that the DLL for Microsoft is created using Microsoft VC++.

---

3 A shareable version of the CDF library is also available on Open/VMS and some flavors of UNIX. Its use is described in Chapter 3. A dynamic link library (DLL), LIBCDF.DLL, is available on Window NT/2000/XP. Consult the Microsoft documentation for details on using a DLL. Note that the DLL for Microsoft is created using Microsoft VC++.
% 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 run-time 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.

2.1.1 Combining the Compile and Link

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

% cc -I${CDF_INC} -D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE
   -D_LARGEFILE_SOURCE <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.

2.2 Windows NT/2000/XP SYSTEMS, Microsoft Visual C++ or Microsoft Visual C++ .NET

An example of the command to link your application with the CDF library (LIBCDF.LIB) on Windows systems using Microsoft Visual C++ or Microsoft Visual C++ .NET would be as follows:4

> LINK /nologo /nodefaultlib:libcd /nodefaultlib:libcm /nodefaultlib:msvcrt \ 
/output:where_to.exe <objs> <lib-path>\libcdf.lib

where <objs> is your application's object module(s); <where_to.exe> is the name of the executable file to be created (with an extension of .exe); and <lib-path> is the file name of the directory containing the CDF library. You will need to know where on your system the CDF library has been installed. <lib-path> may be either an absolute or relative directory name that contains libcdf.lib.

Consult the manuals for Microsoft Visual C++ to set up the proper project/workspace to compile/link your applications.

---

4 This example is extracted from an NMAKE file, created by Microsoft Developer Studio, for compiling/linking the toolkit programs.
Chapter 3

3 Linking Shared CDF Library

A shareable version of the CDF library is also available on OpenVMS systems, some flavors of UNIX and Windows NT/2000/XP. The shared version is put in the same directory as the non-shared version and is named as follows:

<table>
<thead>
<tr>
<th>Machine/Operating System</th>
<th>Shared CDF Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC VAX &amp; Alpha (OpenVMS)</td>
<td>LIBCDF.EXE</td>
</tr>
<tr>
<td>Sun (SunOS)¹</td>
<td>libcdf.so</td>
</tr>
<tr>
<td>Sun (Solaris)</td>
<td>libcdf.so</td>
</tr>
<tr>
<td>HP 9000 (HP-UX)³</td>
<td>libcdf.sl</td>
</tr>
<tr>
<td>IBM RS6000 (AIX)³</td>
<td>libcdf.o</td>
</tr>
<tr>
<td>DEC Alpha (OSF/1)</td>
<td>libcdf.so</td>
</tr>
<tr>
<td>SGi (IRIX 6.x)</td>
<td>libcdf.so</td>
</tr>
<tr>
<td>Linux (PC &amp; Power PC)</td>
<td>libcdf.so</td>
</tr>
<tr>
<td>Windows NT/2000/XP</td>
<td>dllcdf.dll</td>
</tr>
<tr>
<td>Macintosh OS X</td>
<td>libcdf.dylib</td>
</tr>
</tbody>
</table>

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

3.1 DEC VAX & Alpha (OpenVMS)

$ ASSIGN CDF$LIB:LIBCDF.EXE CDF$LIBCDFEXE
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
   CDF$LIBCDFEXE/S-account
   SYS$LIBRARY:<crtl>/LIBRARY
   <contro1-Z>
$ DEASSIGN CDF$LIBCDFEXE

⁵ On UNIX systems, when executing a program linked to the shared CDF library, the environment variable LD_LIBRARY_PATH must be set to include the directory containing libcdf.so or libcdf.sl.

⁶ When executing a program linked to the dynamically linked CDF library (DLL), the environment variable PATH must be set to include the directory containing dllcdf.dll.

⁷ Not yet tested. Please contact gsfc-cdf-support@lists.nasa.gov to coordinate a test.
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 or VAXCRTLT for a default of IEEE_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 DEC Alpha/OpenVMS systems the shareable CDF library may also be installed in SYS$SHARE. If that is the case, the link command would be as follows:

```
$ LINK <object-file(s)>, SYSSINPUT:/OPTIONS
SYSSSHARE:LIBCDF/SHAREABLE
SYSSLIBRARY:<crtl>/LIBRARY
<Control-Z>
```

### 3.2 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. Note that in a “makefile” where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

### 3.3 HP 9000 (HP-UX)

```
% cc -o <exe-file> <object-file(s>).o ${CDF_LIB}/libcdf.sl -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. Note that in a “makefile” where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

### 3.4 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. Note that in a “makefile” where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

---

8 Yet to be tested.
3.5    DEC Alpha (OSF/1)

\texttt{% cc -o <exe-file> <object-file(s)>o \$\{CDF\_LIB\}/libcdf.so -lm -lc}

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. Note that in a “makefile” where CDF\_LIB is imported, \$\{CDF\_LIB\} would be specified instead of $\{CDF\_LIB\}.

3.6    SGi (IRIX 6.x)

\texttt{% cc -o <exe-file> <object-file(s)>o \$\{CDF\_LIB\}/libcdf.so -lm -lc}

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. Note that in a “makefile” where CDF\_LIB is imported, \$\{CDF\_LIB\} would be specified instead of $\{CDF\_LIB\}.

3.7    Linux (PC & Power PC)

\texttt{% cc -o <exe-file> <object-file(s)>o \$\{CDF\_LIB\}/libcdf.so -lm -lc}

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. Note that in a “makefile” where CDF\_LIB is imported, \$\{CDF\_LIB\} would be specified instead of $\{CDF\_LIB\}.

3.8    Windows (NT/2000/XP)

\texttt{% link /out:<exe-file>.exe <object-file(s)>obj <lib-path>dllcdf.lib
/nodefaultlib:libcdf}

where <object-file(s)>obj is your application's object module(s) (the .obj extension is required) and <exe-file>.exe is the name of the executable file created, and <lib-path> may be either an absolute or relative directory name that has dllcdf.lib. The environment variable LIB has to set to the directory that contains LIBC.LIB. Your PATH environment variable needs to be set to include the directory that contains dllcdf.dll when the executable is run.

3.9    Macintosh OS X

\texttt{% cc -o <exe-file> <object-file(s)>o \$\{CDF\_LIB\}/libcdf.dylib -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. Note that in a “makefile” where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.
Chapter 4

4 Programming Interface

4.1 Item Referencing

The following sections describe various aspects of the C programming interface for CDF applications. These include 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 routines/parameters.

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).

4.2 Defined Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDFstatus</td>
<td>All CDF functions, except CDFvarNum, CDFgetVarNum, CDFattrNum and CDFgetAttrNum, 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 A lists the possible status codes along with their explanations. Chapter 8 describes how to interpret status codes.</td>
</tr>
<tr>
<td>CDFid</td>
<td>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.</td>
</tr>
</tbody>
</table>

4.3 CDFstatus Constants

These constants are of type CDFstatus.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF_OK</td>
<td>A status code indicating the normal completion of a CDF function.</td>
</tr>
</tbody>
</table>
CDF_WARN  
Threshold constant for testing severity of non-normal CDF status codes.

Chapter 8 describes how to use these constants to interpret status codes.

### 4.4 CDF Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE_FILE</td>
<td>The CDF consists of only one file. This is the default file format.</td>
</tr>
<tr>
<td>MULTI_FILE</td>
<td>The CDF consists of one header file for control and attribute data and one additional file for each variable in the CDF.</td>
</tr>
</tbody>
</table>

### 4.5 CDF Data Types

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF_BYTE</td>
<td>1-byte, signed integer.</td>
</tr>
<tr>
<td>CDF_CHAR</td>
<td>1-byte, signed character.</td>
</tr>
<tr>
<td>CDF_INT1</td>
<td>1-byte, signed integer.</td>
</tr>
<tr>
<td>CDF_UCHAR</td>
<td>1-byte, unsigned character.</td>
</tr>
<tr>
<td>CDF_UINT1</td>
<td>1-byte, unsigned integer.</td>
</tr>
<tr>
<td>CDF_INT2</td>
<td>2-byte, signed integer.</td>
</tr>
<tr>
<td>CDF_UINT2</td>
<td>2-byte, unsigned integer.</td>
</tr>
<tr>
<td>CDF_INT4</td>
<td>4-byte, signed integer.</td>
</tr>
<tr>
<td>CDF_UINT4</td>
<td>4-byte, unsigned integer.</td>
</tr>
<tr>
<td>CDF_INT8</td>
<td>8-byte, signed integer.</td>
</tr>
<tr>
<td>CDF_REAL4</td>
<td>4-byte, floating point.</td>
</tr>
<tr>
<td>CDF_FLOAT</td>
<td>4-byte, floating point.</td>
</tr>
<tr>
<td>CDF_REAL8</td>
<td>8-byte, floating point.</td>
</tr>
<tr>
<td>CDF_DOUBLE</td>
<td>8-byte, floating point.</td>
</tr>
<tr>
<td>CDF_EPOCH</td>
<td>8-byte, floating point.</td>
</tr>
<tr>
<td>CDF_EPOCH16</td>
<td>two 8-byte, floating point.</td>
</tr>
</tbody>
</table>
CDF_CHAR and CDF_UCHAR are considered character 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). Both CDF_INT8 and CDF_TIME_TT2000, 8-byte integer, can be presented in “long long” in C.

**NOTE:** When using a 64-bit OS. E.g., DEC Alpha running OSF/1, or Linux running 64-bit Intel, 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 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 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 the Concepts chapter in the CDF User's Guide).

- **HOST_ENCODING** Indicates host machine data representation (native). This is the default encoding, and it 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.

- **ALPHAVMSi_ENCODING** Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in IEEE representation.

- **ALPHAOSF1_ENCODING** Indicates DEC Alpha running OSF/1 data representation.

- **SUN_ENCODING** Indicates SUN data representation.

- **SGi_ENCODING** Indicates Silicon Graphics Iris and Power Series data representation.

- **DECSTATION_ENCODING** Indicates DECstation data representation.

- **IBMRS(Encoding)** Indicates IBMRS data representation (IBM RS6000 series).
HP_ENCODING Indicates HP data representation (HP 9000 series).
PC_ENCODING Indicates PC data representation.
NeXT_ENCODING Indicates NeXT data representation.
MAC_ENCODING Indicates Macintosh data representation.

When creating a CDF (via the Standard interface) or respecifying a CDF's encoding (via the Internal Interface), you may specify any of the encodings listed above. 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 a specific machine encoding will be returned. (HOST_ENCODING is never returned.)

### 4.7 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 they are decoded by the CDF library. Any decoding may be used with any of the supported encodings. The Concepts chapter in the CDF User's Guide 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.
ALPHAVMSi_DECODING Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in IEEE representation.
ALPHAOSF1_DECODING Indicates DEC Alpha running OSF/1 data representation.
SUN_DECODING Indicates SUN data representation.
SGi_DECODING Indicates Silicon Graphics Iris and Power Series data representation.
DECSTATION_DECODING Indicates DECstation data representation.
IBMRS_DECODING Indicates IBMRS data representation (IBM RS6000 series).
HP_DECODING Indicates HP data representation (HP 9000 series).

PC_DECODING Indicates PC data representation.

NeXT_DECODING Indicates NeXT data representation.

MAC_DECODING Indicates Macintosh data representation.

The default decoding is HOST_DECODING. The other decodings may be selected via the Internal Interface with the <SELECT_CDF_DECODING> operation. The Concepts chapter in the CDF User's Guide describes those situations in which a decoding other than HOST_DECODING may be desired.

4.8 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. This is the default.

COLUMN_MAJOR Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

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.

As with hyper reads and writes, the majority of a CDF's variables affect multiple variable reads and writes. When performing a multiple variable write, the full-physical records in the buffer passed to the CDF library must have the CDF's variable majority. Likewise, the full-physical records placed in the buffer by the CDF library during a multiple variable read will be in the CDF's variable majority.

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

4.9 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.)

4.10 Compressions

The following types of compression for CDFs and variables are supported. For each, the required parameters are also listed. The Concepts chapter in the CDF User's Guide describes how to select the best compression type/parameters for a particular data set.

**NO_COMPRESSION**
No compression.

**RLE_COMPRESSION**
Run-length encoding compression. There is one parameter.

1. The style of run-length encoding. Currently, only the run-length encoding of zeros is supported. This parameter must be set to RLE_OF_ZEROS.

**HUFF_COMPRESSION**
Huffman compression. There is one parameter.

1. The style of Huffman encoding. Currently, only optimal encoding trees are supported. An optimal encoding tree is determined for each block of bytes being compressed. This parameter must be set to OPTIMAL_ENCODING_TREES.

**AHUFF_COMPRESSION**
Adaptive Huffman compression. There is one parameter.

1. The style of adaptive Huffman encoding. Currently, only optimal encoding trees are supported. An optimal encoding tree is determined for each block of bytes being compressed. This parameter must be set to OPTIMAL_ENCODING_TREES.

**GZIP_COMPRESSION**
Gnu's “zip” compression. There is one parameter.

1. The level of compression. This may range from 1 to 9. 1 provides the least compression and requires less execution time, 9 provides the most compression but requires the most execution time. Values in-between provide varying compromises of these two extremes.

4.11 Sparseness

---

* Disabled for PC running 16-bit DOS/Windows 3.x.
4.11.1 Sparse Records

The following types of sparse records for variables are supported.

- **NO_SPARSERECORDS** No sparse records.
- **PAD_SPARSERECORDS** Sparse records - the variable's pad value is used when reading values from a missing record.
- **PREV_SPARSERECORDS** Sparse records - values from the previous existing record are used when reading values from a missing record. If there is no previous existing record the variable's pad value is used.

4.11.2 Sparse Arrays

The following types of sparse arrays for variables are supported.¹⁰

- **NO_SPARSEARRAYS** No sparse arrays.

4.12 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.)

4.13 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. When read-only mode is set, all metadata is read into memory for future reference. This improves overall metadata access performance but is extra overhead if metadata is not needed. Note that if the CDF is modified while not in read-only mode, subsequently setting read-only mode in the same session will not prevent future modifications to the CDF.

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

¹⁰Obviously, sparse arrays are not yet supported.
4.14  zModes

Once a CDF has been opened, it may be placed into one of two variations of zMode. zMode is fully explained in the Concepts chapter in the CDF User’s Guide. 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.

4.15  -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 or written to 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 or written to a CDF.

- **NEGtoPOSfp0off**  
  Do not convert -0.0 to 0.0 when read from or written to a CDF.

4.16  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.

- **CDF_MAX_PARMS**  
  Maximum number of compression or sparseness parameters.

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

4.17  Limits of Names and Other Character Strings

- **CDF_PATHNAME_LEN**  
  Maximum length of a CDF file name (excluding the NUL\(^\text{11}\) terminator and the .cdf or .vnn appended by the CDF library to construct file names). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating systems being used (including logical

\(^{11}\) The ASCII null character, 0x0.
names on OpenVMS systems and environment variables on UNIX systems).

CDF_VAR_NAME_LEN256  Maximum length of a variable name (excluding the NUL terminator).
CDF_ATTR_NAME_LEN256  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).

4.18 Backward File Compatibility with CDF 2.7

By default, a CDF file created by CDF V3.0 or a later release is not readable by any of the CDF releases before CDF V3.0 (e.g. CDF 2.7.x, 2.6.x, 2.5.x, etc.). The file incompatibility is due to the 64-bit file offset used in CDF 3.0 and later releases (to allow for files greater than 2G bytes). Note that before CDF 3.0, 32-bit file offset was used.

There are two ways to create a file that’s backward compatible with CDF 2.7 and 2.6, but not 2.5. Function CDFsetFileBackward, can be called to control the backward compatibility from an application before a CDF file is created (e.g., via CDFcreateCDF). This function takes an argument to control the backward file compatibility. Passing a flag value of BACKWARDFILEon, defined in cdf.h, to the function will cause the new files being created to be backward compatible. The created files are of version V2.7.2, not V3.*. This option is useful for those who wish to create and share files with colleagues who still use a CDF V2.6/V2.7 library. If this option is specified, the maximum file size is limited to 2G bytes. Passing a flag value of BACKWARDFILEoff, also defined in cdf.h, will use the default file creation mode and the newly created files will not be backward compatible with older libraries. The created files are of version 3.* and thus their file sizes can be greater than 2G bytes. Not calling this function has the same effect of calling the function with an argument value of BACKWARDFILEoff.

The following example uses the Internal Interface to create two CDF files: “MY_TEST1.cdf” is a V3.* file while “MY_TEST2.cdf” a V2.7 file. Alternatively, the Standard Interface function CDFcreateCDF can be used for the file creation.

```
#include "cdf.h"

CDFid     id1, id2;  /* CDF identifier. */
CDFstatus status;    /* Returned status code. */
long      numDims = 0; /* Number of dimensions. */
long      dimSizes[1] = {0}; /* Dimension sizes. */

status = CDFlib (CREATE_, CDF_, "MY_TEST1", numDims, dimSizes, &id1, NULL_);  
if (status != CDF_OK) UserStatusHandler (status);

CDFsetFileBackward(BACKWARDFILEon);
status = CDFlib (CREATE_, CDF_, "MY_TEST2", numDims, dimSizes, &id2, NULL_);
```

19
if (status != CDF_OK) UserStatusHandler (status);

Another method is through an environment variable and no function call is needed (and thus no code change involved in any existing applications). The environment variable, CDF_FILEBACKWARD on all Unix platforms and Windows, or CDFSFILEBACKWARD on Open/VMS, is used to control the CDF file backward compatibility. If its value is set to "TRUE", all new CDF files are backward compatible with CDF V2.7 and 2.6. This applies to any applications or CDF tools dealing with creation of new CDFs. If this environment variable is not set, or its value is set to anything other than “TRUE”, any files created will be of the CDF 3.* version and these files are not backward compatible with the CDF 2.7.2 or earlier versions.

Normally, only one method should be used to control the backward file compatibility. If both methods are used, the function call through CDFsetFileBackward will take the precedence over the environment variable.

You can use the CDFgetFileBackward function to check the current value of the backward-file-compatibility flag. It returns 1 if the flag is set (i.e. create files compatible with V2.7 and 2.6) or 0 otherwise.

#include "cdf.h"

CDFstatus status; /* Returned status code. */

flag = CDFgetFileBackward();

4.19 Checksum

To ensure the data integrity while transferring CDF files from/to different platforms at different locations, the checksum feature was added in CDF V3.2 as an option for the single-file format CDF files (not for the multi-file format). By default, the checksum feature is not turned on for new files. Once the checksum bit is turned on for a particular file, the data integrity check of the file is performed every time it is open; and a new checksum is computed and stored when it is closed. This overhead (performance hit) may be noticeable for large files. Therefore, it is strongly encouraged to turn off the checksum bit once the file integrity is confirmed or verified.

If the checksum bit is turned on, a 16-byte signature message (a.k.a. message digest) is computed from the entire file and appended to the end of the file when the file is closed (after any create/write/update activities). Every time such file is open, other than the normal steps for opening a CDF file, this signature, serving as the authentic checksum, is used for file integrity check by comparing it to the re-computed checksum from the current file. If the checksums match, the file’s data integrity is verified. Otherwise, an error message is issued. Currently, the valid checksum modes are: NO_CHECKSUM and MD5_CHECKSUM, both defined in cdf.h. With MD5_CHECKSUM, the MD5 algorithm is used for the checksum computation. The checksum operation can be applied to CDF files that were created with V2.7 or later.

There are several ways to add or remove the checksum bit. One way is to use the Interface call (Standard or Internal) with a proper checksum mode. Another way is through the environment variable. Finally, CDFedit and CDFconvert (CDF tools included as part of the standard CDF distribution package) can be used for adding or removing the checksum bit. Through the Interface call, you can set the checksum mode for both new or existing CDF files while the environment variable method only allows to set the checksum mode for new files.

See Section 6.2.5 and 6.2.25 for the Standards Interface functions and Section 7.6 for the Internal Interface functions. The environment variable method requires no function calls (and thus no code change is involved for existing applications). The environment variable CDF_CHECKSUM on all Unix platforms and Windows, or CDFSCHECKSUM on Open/VMS, is used to control the checksum option. If its value is set to "MD5", all new CDF
files will have their checksum bit set with a signature message produced by the MD5 algorithm. If the environment variable is not set or its value is set to anything else, no checksum is set for the new files.

The following example uses the Internal Interface to set a new CDF file with the MD5 checksum and set another existing file’s checksum to none.

```
#include "cdf.h"

CDFid id1, id2; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long numDims = 0; /* Number of dimensions. */
long dimSizes[1] = {0}; /* Dimension sizes. */
long checksum; /* Checksum code. */

status = CDFlib (CREATE_, CDF_, "MY_TEST1", numDims, dimSizes, &id1, NULL_);
if (status != CDF_OK) UserStatusHandler (status);

checksum = MD5_CHECKSUM;
status = CDFlib (SELECT_, CDF_, id1, PUT_, CDF_CHECKSUM_, checksum, NULL_);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (OPEN_, CDF_, "MY_TEST2", &id2, NULL_);
if (status != CDF_OK) UserStatusHandler (status);

checksum = NO_CHECKSUM;
status = CDFlib (SELECT_, CDF_, id2, PUT_, CDF_CHECKSUM_, checksum, NULL_);
if (status != CDF_OK) UserStatusHandler (status);
```

Alternatively, the Standard Interface function `CDFsetChecksum` can be used for the same purpose.

The following example uses the Internal Interface whether the checksum mode is enabled for a CDF.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long checksum; /* Checksum code. */
```
status = CDFlib (OPEN_, CDF_, “MY_TEST1”, &id,
    NULL_);  
if (status != CDF_OK) UserStatusHandler (status); 

status = CDFlib (SELECT_, CDF_, id,
    GET_, CDF_CHECKSUM_, &checksum,
    NULL_);  
if (status != CDF_OK) UserStatusHandler (status); 
if (checksum == MD5_CHECKSUM)
    
    

Alternatively, the Standard Interface function CDFgetChecksum can be used for the same purpose.

4.20 Data Validation

To ensure the data integrity from CDF files and secure operation of CDF-based applications, a data validation feature is added while a CDF file is opened. This process, as the default, performs sanity checks on the data fields in the CDF internal data structures to make sure that the values are within ranges and consistent with the defined values/types/entries. It also tries to ensure that the linked lists within the file that connect the attributes and variables are not broken or short-circuited. Any compromised CDF files, if not validated properly, could cause applications to function unexpectedly, e.g., segmentation fault due to a buffer overflow. The main purpose of this feature is to safeguard the CDF operations: catch any bad data in the file and end the application gracefully if any bad data is identified. An overhead (performance hit) is expected and it may be noticeable for large or very fragmented files. Therefore, it is advised that this feature be turned off once a file’s integrity is confirmed or verified. Or, the file in question may need a file conversion, which will consolidate the internal; data structures and eliminate the fragmentations. Check the cdfconvert tool program in the CDF User’s Guide.

This validation feature is controlled by the setting/unsetting the environment variable CDF_VALIDATE on all Unix platforms, Mac OS X and Windows, or CDFSVALIDATE on Open/VMS. If its value is not set or set to “yes”, all open CDF files are subjected to this data validation process. If the environment variable is set to “no”, then no validation is performed. The environment variable can be set at logon or through command line, which becomes in effective during terminal session, or by an application, which is good only while the application is run. Setting the environment variable, CDFsetValidate, at application level will overwrite the setup from the command line. The validation is set to be on when VALIDATEFILEon is passed into as the argument. VALIDATEFILEoff will set off the validation. CDFgetValidate will return the validation mode, 1 (one) means data being validated, 0 (zero) otherwise. If the environment variable is not set, the default is to have the data validated when a CDF file is open.

The following example sets the data validation on when the CDF file, “TEST”, is open.

```c
#include ‘cdf.h’

CDFid id /* CDF identifier. */
CDFstatus status /* Returned status code. */

CDFsetValidate (VALIDATEFILEon)
status = CDF_lib (OPEN_, CDF_, “TEST”, &id,
    NULL_)
if (status .NE. CDF_OK) UserStatusHandler (status)
```
The following example turns off the data validation when the CDF file, “TEST” is open.

```c
#include ‘cdf.h’

CDFid id /* CDF identifier. */
CDFstatus status /* Returned status code. */

CDFsetValidate (VALIDATEFILEoff)
status = CDF_lib (OPEN_, CDF_, “TEST”, &id, 
               NULL_)
if (status .NE. CDF_OK) UserStatusHandler (status)
```

## 4.21 8-Byte Integer

Both data types of CDF_INT8 and CDF_TIME_TT2000 use 8-bytes signed integer. While there are several ways to define such integer by various C compilers on various platforms, “long long” appears to be accepted by all ports that support CDF. This is the data type that CDF library uses for these two CDF data types.
Chapter 5

5 Standard Interface

The Standard Interface functions described in this chapter represents the original Standard Interface functions. As most of them were developed when CDF was first introduced in early 90’s and they only provide a very limited functionality within the CDF library. For example, it can not handle zVariables thoroughly and has no access to attribute’s entry corresponding to the zVariables (zEntries). If you want to create or access zVariables and zEntries, you must use the newer Standard Interface functions (a new feature in CDF Version 3.1) in Chapter 6 or the Internal Interface described in Chapter 7.

Standard Interface functions are easier-to-use and require a much shorter learning curve than the Internal Interface, but they are not as efficient as Internal Interface. If you are not familiar with Internal Interface, the use of Standard Interface is recommended.

There are two types of variables (rVariable and zVariable) in CDF, and they can happily coexist in a CDF: Every rVariable in a CDF must have the same number of dimensions and dimension sizes while each zVariable can have its own dimensionality. Since all the rVariables in a CDF must have the same dimensions and dimension sizes, there'll be a lot of disk space wasted if a few variables need big arrays and many variables need small arrays. Since zVariable is more efficient in terms of storage and offers more functionality than rVariable, use of zVariable is strongly recommended. As a matter of fact, there’s no reason to use rVariables at all if you are creating a CDF file from scratch. One may wonder why there are rVariables and zVariables, not just zVariables. When CDF was first introduced, only rVariables were available. The inefficiencies with rVariables were quickly realized and addressed with the introduction of zVariables in later CDF releases.

The following sections describe the original Standard Interface functions callable from C applications. Most functions return a status code of type CDFstatus (see Chapter 8). The Internal Interface is described in Chapter 7. An application can use either or both interfaces when necessary.

Each section begins with a function prototype for the routine being described. The include file cdf.h contains the same function prototypes (as well as function prototypes for the Internal Interface and EPOCH utility routines). Note that many of the Standard Interface functions in this chapter are implemented as macros (which call the Internal Interface).

5.1 CDFattrCreate\(^\text{12}\)

CDFstatus CDFattrCreate(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier */

\(^{12}\) Same as CDFcreateAttr.
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 CDFcreateCDF) or CDFopen.
- **attrName**: The name of the attribute to create. This may be at most CDF_ATTR_NAME_LEN256 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 4.12.
- **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 CDFgetAttrNum function.

### 5.1.1 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).

```c
#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);
```

### 5.2 CDFattrEntryInquire
CDFstatus CDFattrEntryInquire(
    /* out -- Completion status code. */
    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. 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 CDFcreateCDF) or CDFopen.
- **attrNum**: The attribute number for which to inquire an entry. This number may be determined with a call to CDFattrNum (see Section 5.5).
- **entryNum**: The entry number to inquire. If the attribute is global in scope, this is simply the gEntry 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 4.5.
- **NumElements**: The number of elements of the data type. For character 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.

### 5.2.1 Example(s)

The following example returns 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.

```c
#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_LEN256+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 = CDFgetAttrNum (id, "TMP");
if (attrN < CDF_OK) UserStatusHandler (attrN);
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 */
        
    }
}

5.3 CDFattrGet

CDFstatus CDFattrGet(
    /* out -- Completion status code. */
    CDFid id,
    /* in -- CDF identifier. */
    long attrNum,
    /* in -- Attribute number. */
    long entryNum,
    /* in -- Entry number. */
    void *value);
    /* out -- Attribute entry 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 CDFcreateCDF) or CDFopen.

attrNum The attribute number. This number may be determined with a call to CDFattrNum (Section 5.5).

entryNum The entry number. If the attribute is global in scope, this is simply the gEntry 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.

13 An original Standard Interface function. While it is still available in V3.1, CDFgetAttrgEntry or CDFgetAttrrEntry is the preferred name for it.
5.3.1 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 < CDF_OK) UserStatusHandler (attrN);
entryN = CDFvarNum (id, "PRES_LVL"); /* The rEntry number is the rVariable number. */
if (entryN < CDF_OK) UserStatusHandler (entryN);
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);
}
```

5.4 CDFattrInquire\(^\text{14}\)

\(^\text{14}\) An original Standard Interface function. While it is still available in V3.1, CDFinquireAttr is the preferred name for it.
CDFstatus CDFattrInquire(
    /* out -- Completion status code. */
    CDFid id,
    /* in -- CDF identifier. */
    long attrNum,
    /* in -- Attribute number. */
    char *attrName,
    /* out -- Attribute name. */
    long *attrScope,
    /* out -- Attribute scope. */
    long *maxEntry);
    /* out -- Maximum gEntry or rEntry number. */

CDFattrInquire is used to inquire about the specified attribute. To inquire about a specific attribute entry, use CDFattrEntryInquire.

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 CDFcreateCDF) or CDFopen.

- **attrNum**
  The number of the attribute to inquire. This number may be determined with a call to CDFattrNum (see Section 5.5).

- **attrName**
  The attribute's name. This character string must be large enough to hold CDF_ATTR_NAME_LEN256 + 1 characters (including the NUL terminator).

- **attrScope**
  The scope of the attribute. Attribute scopes are defined in Section 4.12.

- **maxEntry**
  For gAttributes this is the maximum 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 7). If no entries exist for the attribute, then a value of -1 will be passed back.

5.4.1 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.

```c
#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_LEN256+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);
}

5.5  CDFattrNum

long CDFattrNum( /* out -- attribute number. */
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 CDFcreateCDF) or CDFopen.

attrName The name of the attribute for which to search. This may be at most CDF_ATTR_NAME_LEN256 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.

5.5.1 Example(s)

In the following example the attribute named pressure will be renamed to 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.

#include "cdf.h"

An original Standard Interface function. While it is still available in V3.1, CDFgetAttrNum is the preferred name for it.
CDFid id;          /* CDF identifier. */
CDFstatus status;  /* Returned status code. */

status = CDFattrRename (id, CDFattrNum(id,"pressure"), "PRESSURE");
if (status != CDF_OK) UserStatusHandler (status);

5.6 CDFattrPut

CDFstatus CDFattrPut(    /* out -- Completion status code. */
    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 -- Attribute entry value. */

CDFattrPut is used to write an entry to a global or rVariable attribute in 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 CDFcreateCDF) or CDFopen.

attrNum
The attribute number. This number may be determined with a call to CDFgetAttrNum.

entryNum
The entry number. If the attribute is global in scope, this is simply the gEntry 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 4.5.

numElements
The number of elements of the data type. For character 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.

5.6.1 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[2] = {15,30}; /* Value(s) of VALIDs attribute, rEntry for rVariable TMP. */

entryNum = 0;
status = CDFattrPut (id, CDFgetAttrNum(id,"TITLE"), entryNum, CDF_CHAR, TITLE_LEN, title);
if (status != CDF_OK) UserStatusHandler (status);

numElements = 2;
status = CDFattrPut (id, CDFgetAttrNum(id,"VALIDs"), CDFgetVarNum(id,"TMP"),
CDF_INT2, numElements, TMPvalids);
if (status != CDF_OK) UserStatusHandler (status);

5.7 CDFattrRename

CDFstatus CDFattrRename(/* out -- Completion status code. */
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 CDFcreateCDF) or CDFopen.

attrNum The number of the attribute to rename. This number may be determined with a call to CDFattrNum (see Section 5.5).

16 An original Standard Interface function. While it is still available in V3.1, CDFrenameAttr is the preferred name for it.
attrName The new attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters (excluding the NUL terminator). Attribute names are case-sensitive.

5.7.1 Example(s)

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

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */

status = CDFattrRename (id, CDFgetAttrNum(id,"LAT"), "LATITUDE");
if (status != CDF_OK) UserStatusHandler (status);
```

5.8 CDFclose

```
CDFstatus CDFclose( /* out -- Completion status code. */
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 CDFcreateCDF) or CDFopen.

5.8.1 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);

5.9 CDFcreate

CDFstatus CDFcreate( /* out -- Completion status code. */
char *CDFname, /* in -- CDF file name. */
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 dotCDF file (having an extension of .cdf), and if the CDF has the multi-file format, delete all of the variable files (having extensions of .v0,.v1, . . . and .z0,.z1, . . . ).

The arguments to CDFcreate are defined as follows:

CDFname The file name 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 file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

UNIX: File names 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. Each element of dimSizes specifies the corresponding dimension size. Each size must be greater then zero (0). For 0-dimensional rVariables this argument is ignored (but must be present).

encoding The encoding for variable data and attribute entry data. Specify one of the encodings described in Section 4.6.

majority The majority for variable data. Specify one of the majorities described in Section 4.8.

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.

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 5.8).

5.9.1 Example(s)

The following example creates a CDF named “test1.cdf” 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.
```

5.10 CDFdelete

CDFstatus CDFdelete(   /* out -- Completion status code. */  
CDFid id);             /* in  -- CDF identifier. */

CDFdelete deletes the specified CDF. The CDF files deleted include the dotCDF file (having an extension of .cdf), and if a multi-file CDF, the variable files (having extensions of .v0,.v1, . . . and .z0,.z1, . . . ).

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 CDFcreateCDF) or CDFopen.
5.10.1 Example(s)

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

```c
#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);
}
```

5.11 CDFdoc

CDFstatus CDFdoc( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    long *version, /* out -- Version number. */
    long *release, /* out -- Release number. */
    char *Copyright); /* out -- Copyright. */

CDFdoc is used to inquire general information about a CDF. The version/release of the CDF library that created the CDF is provided (e.g., CDF V3.1 is version 3, release 1) along with the CDF Copyright notice. The Copyright notice is formatted for printing without modification.

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 CDFcreateCDF) 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.
5.11.1 Example(s)

The following example returns and displays the version/release and Copyright notice.

```c
#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);
}
```

5.12 CDFerror

CDFstatus CDFerror( /* out -- Completion status code. */
CDFstatus status, /* in -- Status code. */
char *message); /* out -- Explanation text for the status code. */

CDFerror is used to inquire the explanation of a given status code (not just error codes). Chapter 8 explains how to interpret status codes and Appendix A 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).

5.12.1 Example(s)

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

---

17 An original Standard Interface function. While it is still available in V3.1, CDFgetStatusText is the preferred name for it.


```c
#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> %sn", text);
}
```

5.13 **CDFgetrVarsRecordData**

CDFgetrVarsRecordData reads an entire record from a specified record number for a number of the specified variables in a CDF. This function provides an easier and higher level interface to acquire data for a group of variables, instead of doing it one variable at a time if calling the lower-level function. The retrieved record data from the variable group is added to the buffer. The specified variables are identified by their names. Use CDFgetrVarsRecordDataByNumbers function to perform the similar operation by providing the variable numbers, instead of the names.

The arguments to CDFgetrVarsRecordData are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
- **varsNum**: The number of variables in the operation.
- **varNames**: The names of variables in the operation.
- **recNum**: The record number.
- **buffer**: The data holding buffer.

---

18 An original Standard Interface function.
5.13.1 Example(s)

The following example will read an entire single record data for a group of rVariables. The CDF's rVariables are 2-
dimensional with sizes [2,2]. The rVariables involved in the read are Time, Longitude, Latitude, Temperature and NAME. The record to be read is 4. Since the dimension variances for Time are [NONVARY,NONVARY], a scalar variable of type int is allocated for its data type CDF INT4. For Longitude, a 1-dimensional array of type float (size [2]) is allocated for its dimension variances [VARY,NONVARY] and data type CDF REAL4. A similar allocation is done for Latitude for its [NONVARY,VARY] dimension variances and CDF REAL4 data type. For Temperature, since its [VARY,VARY] dimension variances and CDF REAL4 data type, a 2-dimensional array of type float is allocated. For NAME, a 2-dimensional array of type char (size [2,10]) is allocated for its [VARY,NONVARY] dimension variances and CDF CHAR data type with the number of element 10.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long numVars = 5; /* Number of rVariables to read. */
long varRecNum = 4; /* The record number to read data. */
char *rVar1 = "Time",
    *rVar2 = "Longitude",
    *rVar3 = "Latitude",
    *rVar4 = "Temperature",
    *rVar5 = "NAME";
char *varNames[5];

void *buffer; /* Array of buffer pointers. */
int time; /* rVariable: Time; Datatype: INT4. */
    /* Dim/Rec Variances: T/FF. */
float longitude[2]; /* rVariable: Longitude; Datatype: REAL4. */
    /* Dim/Rec Variances: T/TF. */
float latitude[2]; /* rVariable: Latitude; Datatype: REAL4. */
    /* Dim/Rec Variances: T/FT. */
float temperature[2][2]; /* rVariable: Temperature; Datatype: REAL4. */
    /* Dim/Rec Variances: T/TT. */
char name[2][10]; /* rVariable: Name; Datatype: CHAR/10. */
    /* Dim/Rec Variances: T/TF. */

varNames[0] = rVar1; /* Name of each rVariable. */
varNames[1] = rVar2;
varNames[2] = rVar3;
varNames[3] = rVar4;
varNames[4] = rVar5;

buffer = (void *) malloc(sizeof(time) + sizeof(longitude) + sizeof(latitude) + sizeof(temperature) + sizeof(name));

status = CDFgetrVarsRecordData(id, numVars, varNames, varRecNum, buffer);
if (status != CDF_OK) UserStatusHandler (status);
```
5.14 CDFgetzVarsRecordData

CDFstatus CDFgetzVarsRecordData
( /* out -- Completion status code. */
 CDFid id, /* in -- CDF identifier. */
 long numVars, /* in -- Number of zVariables. */
 char *varNames[], /* in -- Names of zVariables. */
 long varRecNum, /* in -- Number of record. */
 void *buffers[]; /* out -- Array of buffers for holding data. */
);

CDFgetzVarsRecordData reads an entire record of the specified record number from the specified zVariables in a CDF. This function provides an easier and higher level interface to acquire data from a group of variables, instead of reading data one variable at a time. The retrieved record data from the zVariable group is put into the respective buffer. The specified variables are identified by their names. Use the CDFgetzVarsRecordDatabyNumbers function to perform the similar operation by providing the variable numbers, instead of the variable names.

The arguments to CDFgetzVarsRecordData are defined as follows:

- **id** The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDFopen or a similar CDF creation or opening functionality from the Internal Interface.
- **numVars** The number of the zVariables in the group involved this read operation.
- **varNames** The names of the zVariables from which to read data.
- **varRecNum** The record number at which to read data.
- **buffers** An array of buffers, each holding the retrieved data for the given zVariables. Each buffer should be big enough to allow full physical record data to fill.

5.14.1 Example(s)

The following example will read an entire single record data for a group of zVariables: Time, Longitude, Delta and Name. The record to be read is the sixth record that is record number 5 (record number starts at 0). For Longitude, a 1-dimensional array of type short (size [3]) is given based on its dimension variance [VARY] and data type CDF_INT2. For Delta, it is 2-dimensional of type int (sizes [3,2]) for its dimension variances [VARY,VARY] and data type CDF_INT4. For zVariable Time, a 2-dimensional array of type unsigned int (size [3,2]) is needed. It has dimension variances [VARY,VARY] and data type CDF_UINT4. For Name, a 2-dimensional array of type char (size [2,10]) is allocated for its [VARY] dimension variances and CDF_CHAR data type with the number of element 10.

```
#include "cdf.h"

CDFid          id;                                   /* CDF identifier. */
CDFstatus   status;                            /* Returned status code. */
long             numVars = 4;                 /* Number of zVariables to read. */
long             varRecNum = 5;             /* The record number to read data – 6th record */
char             *zVar1 = "Longitude",     /* Names of the zVariables to read. */
                *zVar2 = "Delta",
                *zVar3 = "Time",
                *zVar4 = "Name";
```
void **varNames; /* Variable names array. */
void **buffers; /* Array of buffers to hold the returned data. */
unsigned int time[3][2]; /* Variable: Time; Datatype: UINT4. */
    /* Dimensions: 2:[3,2]; Dim/Rec Variances: T/TT. */
short longitude[3]; /* Variable: Longitude; Datatype: INT2. */
    /* Dimensions: 1:[3]; Dim/Rec Variances: T/T. */
int delta[3][2]; /* Variable: Delta; Datatype: INT4. */
    /* Dimensions: 2:[3,2]; Dim/Rec Variances: T/TT. */
char name[2][10]; /* Variable: Name; Datatype: CHAR/10. */
    /* Dimensions: 1:[2]; Dim/Rec Variances: T/T. */

int i;

varNames = (void **) malloc (4 * sizeof(char *));
for (I = 0; I < 4; ++I)
    varNames[I] = (char *) malloc (CDF_VAR_NAME_LEN256+1);

strcpy(varNames[0], zVar1); /* Name of each variable. */
strcpy(varNames[1], zVar2);
strcpy(varNames[2], zVar3);
strcpy(varNames[3], zVar4);

buffers = (void **) malloc(4 * (sizeof(void *));
buffers[0] = time;
buffers[1] = longitude;
buffers[2] = delta;
buffers[3] = name;

status = CDFgetzVarsRecordData(id, numVars, varNames, varRecNum, buffers);
if (status != CDF_OK) UserStatusHandler (status);
.
.
for (i = 0; i < 4; ++i)
    free (varNames[i]);
free (varNames);
free (buffers);

5.15 CDFinquire

CDFstatus CDFinquire,
    /* out -- Completion status code. */
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 returns 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 (since all rVariables’ dimension and dimension size are
the same). 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 CDFcreateCDF) 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. For 0-dimensional rVariables this argument is ignored (but must be present).

- **encoding**
  - The encoding of the variable data and attribute entry data. The encodings are defined in Section 4.6.

- **majority**
  - The majority of the variable data. The majorities are defined in Section 4.8.

- **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. Use CDFrVarMaxWrittenRecNum to inquire the maximum record written for an individual rVariable.

- **numVars**
  - The number of rVariables in the CDF.

- **numAttrs**
  - The number of attributes in the CDF.

### 5.15.1 Example(s)

The following example returns the basic information about a CDF.

```c
#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);
```
5.16 CDFopen

CDFstatus CDFopen( /* out -- Completion status code. */
char *CDFname,   /* in -- CDF file name. */
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:

CDFname The file name 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 file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

UNIX: File names 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.

5.16.1 Example(s)

The following example will open a CDF named “NOAA1.cdf”.

```
#include "cdf.h"

CDFId id;     /* CDF identifier. */
CDFstatus status;    /* Returned status code. */
static char CDFname[] = { "NOAA1" };    /* file name of CDF. */

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


5.17  CDFputrVarsRecordData\(^{19}\)

CDFstatus CDFputrVarsRecordData( /* out -- Completion status code. */ \
    CDFid id, /* in -- CDF identifier. */  \
    long numVars, /* in -- Number of rVariables. */  \
    char *varNames[], /* in -- Names of rVariables. */  \
    long varRecNum, /* in -- Number of record. */  \
    void *buffers[]; /* in -- Array of buffers for input data. */
)

CDFputrVarsRecordData is used to write a whole record data at a specific record number for a group of rVariables in a CDF. It expects that the each buffer matches up to the total full physical record size of its corresponding rVariables to be written. Passed record data is filled into its respective rVariable’s buffer. This function provides an easier and higher level interface to write data for a group of variables, instead of doing it one variable at a time if calling the lower-level function. The specified variables are identified by their names. Use CDFputrVarsRecordDataByNumbers function to perform the similar operation by providing the variable numbers, instead of the names.

The arguments to CDFputrVarsRecordData are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDFopen or a similar CDF creation or opening functionality from the Internal Interface.
- **numVars**: The number of the rVariables in the group involved this write operation.
- **varNames**: The names of the rVariables involved for which to write a whole record data.
- **varRecNum**: The record number at which to write the whole record data for the group of rVariables.
- **buffers**: The array of buffers, each holding the output data for the full record of a given rVariables.

5.17.1  Example(s)

The following example will write an entire single record data for a group of rVariables. The CDF's rVariables are 2-dimensional with sizes [2,2]. The rVariables involved in the write are Time, Longitude, Latitude and Temperature. The record to be written is 4. Since the dimension variances for Time are [NONVARY,NONVARY], a scalar variable of type int is allocated for its data type CDF_INT4. For Longitude, a 1-dimensional array of type float (size [2]) is allocated as its dimension variances are [VARY,NONVARY] with data type CDF_REAL4. A similar 1-dimensional array is provided for Latitude for its [NONVARY,VARY] dimension variances and CDF_REAL4 data type. For Temperature, since its [VARY,VARY] dimension variances and CDF_REAL4 data type, a 2-dimensional array of type float is provided. For NAME, a 2-dimensional array of type char (size [2,10]) is allocated due to its [VARY, NONVARY] dimension variances and CDF_CHAR data type with the number of element 10.

```c
#include "cdf.h"

CDFid id; /* Dim/Rec Variances: T/TF. */  
CDFstatus status; /* CDF identifier. */
```

\(^{19}\) An original Standard Interface function.
long numVars = 5; /* Number of rVariables to write. */
long varRecNum = 4; /* The record number to write data. */
char *rVar1 = "Time", /* Names of the rVariables to write. */
    *rVar2 = "Longitude",
    *rVar3 = "Latitude",
    *rVar4 = "Temperature",
    *rVar5 = "NAME";

void *buffer; /* The ouput buffer. */
void *bufferptr; /* Buffer place keeper */
int time = {123} /* rVariable: Time; Datatype: INT4. */
    {/* Dim/Rec Variances: T/FF. */
    float longitude[2] = /* rVariable: Longitude; Datatype: REAL4. */
    {11.1, 22.2}; /* Dim/Rec Variances: T/TF. */
    float latitude[2] = /* rVariable: Latitude; Datatype: REAL4. */
    {-11.1, -22.2}; /* Dim/Rec Variances: T/FT. */
    float temperature[2][2] = /* rVariable: Temperature; Datatype: REAL4. */
    {100.0, 200.0, 300.0, 400.0}; /* Dim/Rec Variances: T/TT. */
    char name[2][10] = /* rVariable: NAME; Datatype: CHAR/10. */
    {"1", "3", "5", "7", "9", "2", "4", "6", "8", "0", "z", "Z", "y", "Y", "x", "X", "w", "W", "v", "V"};

int i;

varNames = (void **) malloc(4 * sizeof (char *));
for (i = 0; i < 4; ++i)
    varNames[i] = (char *) malloc(CDF_VAR_NAME_Len256+1));

strcpy (varName[0], rVar1); /* Name of each rVariable. */
strcpy (varNames[1], rVar2);
strcpy (varNames[2], rVar3);
strcpy (varNames[3], rVar4);
buffers = (void **) malloc (4 * sizeof(void *));
buffers[0] = (void *) malloc(sizeof(longitude));
memcpy(buffers[0], (void *) longitude, sizeof(longitude));
buffers[1] = (void *) malloc(sizeof(delta));
memcpy(buffers[1], (void *) delta, sizeof(delta));
buffers[2] = (void *) malloc(sizeof(time));
memcpy(buffers[2], (void *) time, sizeof(time));
buffers[3] = (void *) malloc(sizeof(name));
memcpy(buffers[3], (void *) name, sizeof(name));

status = CDFputrVarsRecordData(id, numVars, varNames, varRecNum, buffers);
if (status != CDF_OK) UserStatusHandler (status);

for (i = 0; i < 4; ++i) {
    free (varNames[i]);
    free (buffers[i]);
}
free (varNames);
free (buffers);
5.18  CDFputzVarsRecordData

CDFstatus CDFputzVarsRecordData(  /* out -- Completion status code. */
    CDFid id,  /* in -- CDF identifier. */
    long numVars,  /* in -- Number of zVariables. */
    char *varNames[],  /* in -- Names of zVariables. */
    long recNum,  /* in -- Record number. */
    void *buffers[]);  /* in -- Array of buffers for input data. */

CDFputzVarsRecordData is used to write a whole record data at a specific record number for a group of zVariables in a CDF. It expects that each data buffer matches up to the total full physical record size for its corresponding zVariable. Passed record data is filled into its respective zVariable. Use CDFputzVarsRecordDataByNumbers function to perform the similar operation by providing the variable numbers, instead of the names.

The arguments to CDFputzVarsRecordData are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDFopen or a similar CDF creation or opening functionality from the Internal Interface.

numVars The number of the zVariables in the group involved this write operation.

varNames The names of the zVariables involved for which to write a whole record data.

recNum The record number at which to write the whole record data for the group of zVariables.

buffers An array of buffers, each holding the output data for a full record of a given zVariable.

5.18.1  Example(s)

The following example will write an entire single record data for a group of zVariables. The zVariables involved in the write are Time, Longitude, Delta and Name. The record to be written is 5. For Longitude, a 1-dimensional array of type short (size [3]) is provided for its dimension variance [VARY] and data type CDF_INT2. For Delta, a 2-dimensional array of type int (size [3, 2]) is provided as its dimension variances are [VARY, VARY] with data type CDF_INT4. For Time, it is 2-dimensional of type unsigned int (sizes [3, 2]) for its dimension variances [VARY, VARY] and data type CDF_UINT4. For Name, a 2-dimensional array of type char (size [2, 10]) is provided due to its [VARY] dimension variances and CDF_CHAR data type with the number of element 10.

```
#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long numVars = 4;  /* Number of zVariables to write. */
long varRecNum = 5;  /* The record number to write data. */
char *zVar1 = "Longitude",
    *zVar2 = "Delta",
    *zVar3 = "Time",
    *zVar4 = "Name";
```
This function can be a replacement for the similar functionality provided from the Internal Interface as `<PUT_, zVARs_RECDATA_>`.

5.19 CDFvarClose

CDFstatus CDFvarClose(  
/* out -- Completion status code. */  
CDFid id,  
/* in -- CDF identifier. */  
long varNum);  
/* in -- rVariable number. */

CDFvarClose closes the specified rVariable file from a multi-file format CDF. The variable's cache buffers are flushed before the variable's open file is closed. However, the CDF file is still open.

20 An original Standard Interface function, handling rVariables only.
NOTE: You must close all open variable files 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:

\[
\begin{align*}
\text{id} & \quad \text{The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.} \\
\text{varNum} & \quad \text{The variable number for the open rVariable’s file. This identifier must have been initialized by a call to CDFgetVarNum.}
\end{align*}
\]

5.19.1 Example(s)

The following example will close an open 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);
```

5.20 CDFvarCreate\textsuperscript{21}

CDFstatus CDFvarCreate(     /* out -- Completion status code. */
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:

\textsuperscript{21} An original Standard Interface function, handling rVariables only.
id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.

varName The name of the rVariable to create. This may be at most CDF_VAR_NAME_LEN256 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 4.5.

numElements The number of elements of the data type at each value. For character 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 data types.

recVariance The rVariable's record variance. Specify one of the variances defined in Section 4.9.

dimVariances The rVariable's dimension variances. Each element of dimVariances specifies the corresponding dimension variance. For each dimension specify one of the variances defined in Section 4.9. For 0-dimensional rVariables this argument is ignored (but must be present).

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 rVariable's number may be determined with the CDFvarNum or CDFgetVarNum function.

5.20.1 Example(s)

The following example will create several rVariables in a 2-dimensional CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
static long EPOCHEpochRecVary = {VARY}; /* EPOCH record variance. */
static long LATEpochRecVary = {NOVARY}; /* LAT record variance. */
static long LONLONRecVary = {NOVARY}; /* LON record variance. */
static long TMPTMPRecVary = {VARY}; /* TMP record variance. */
static long EPOCHEpochDimVarys[1] = {NOVARY,NOVARY}; /* EPOCH dimension variances. */
static long LATLATEpochDimVarys[2] = {VARY,VARY}; /* LAT dimension variances. */
static long LONLONLONDimVarys[2] = {VARY,VARY}; /* LON dimension variances. */
static long TMPTMPTMPDimVarys[2] = {VARY,VARY}; /* TMP dimension variances. */
long EPOCHEpochVarNum; /* EPOCH zVariable number. */
long LATEpochVarNum; /* LAT zVariable number. */
long LONLONLONVarNum; /* LON zVariable number. */
long TMPTMPTMPVarNum; /* TMP zVariable number. */

status = CDFvarCreate (id, "EPOCH", CDF_EPOCH, 1,
EPOCHEpochRecVary, EPOCHEpochDimVarys, &EPOCHEpochVarNum);
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);

5.21 CDFvarGet

CDFstatus CDFvarGet(id,  /* out -- Completion status code. */
    varNum,  /* in -- CDF identifier. */
    recNum,  /* in -- rVariable number. */
    indices[],  /* in -- Record number. */
    value);  /* out -- Dimension indices. */

CDFvarGet is used to read a single value from an rVariable.

The arguments to CDFvarGet are defined as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.</td>
</tr>
<tr>
<td>varNum</td>
<td>The rVariable number from which to read data.</td>
</tr>
<tr>
<td>recNum</td>
<td>The record number at which to read.</td>
</tr>
<tr>
<td>indices[]</td>
<td>The dimension indices within the record.</td>
</tr>
<tr>
<td>value</td>
<td>The data value read. This buffer must be large enough to hold the value.</td>
</tr>
</tbody>
</table>

5.21.1 Example(s)

The following example returns two data values, the first and the fifth element, in Record 0 from an rVariable named MY_VAR, a 2-dimensional (2 by 3) CDF_DOUBLE type variable, in a row-major CDF.

```c
#include "cdf.h"
```
CDFid id;  /* CDF identifier. */
long varNum;  /* rVariable number. */
long recNum;  /* The record number. */
long indices[2];  /* The dimension indices. */
double value1, value2;  /* The data values. */

varNum = CDFvarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“….
recNum = 0L;
indices[0] = 0L;
indices[1] = 0L;
status = CDFvarGet (id, varNum, recNum, indices, &value1);
if (status != CDF_OK) UserStatusHandler (status);
indices[0] = 1L;
indices[1] = 1L;
status = CDFvarGet (id, varNum, recNum, indices, &value2);
if (status != CDF_OK) UserStatusHandler (status);

5.22  CDFvarHyperGet

CDFstatus CDFvarHyperGet(  /* out -- Completion status code. */
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 fill a buffer of one or more values from the specified 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 can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

5.22.1  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 provided for CDFvarGet except that it uses a single call to CDFvarHyperGet rather than numerous calls to CDFvarGet.

23 An original Standard Interface function, handling rVariables only.
#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 = CDFgetVarNum (id, "Temperature");
if (varN < CDF_OK) UserStatusHandler (varN);
status = CDFgetHyperGet (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 COLUMN_MAJOR, the tmp array would have been declared float tmp[10][91][180] for proper indexing.

5.23 CDFvarHyperPut

CDFstatus CDFvarHyperPut( /* out -- Completion status code. */
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 one or more values from the data holding buffer to the specified rVariable. It is important to know the variable majority of the CDF before using this routine because the values in the buffer to be written must be in the same majority. CDFinquire can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

---

An original Standard Interface function, handling rVariables only.
5.23.1 Example(s)

The following example writes values to the rVariable LATITUDE of a CDF that is an 2-dimensional array 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 CDFvarPut example except that it uses a single call to CDvarHyperPut 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. */
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 < CDF_OK) UserStatusHandler (varN);
for (lat = -90; lat <= 90; lat ++)
    lats[90+lat] = lat;

status = CDFvarHyperPut (id, varN, recStart, recCount, recInterval, indices, counts, intervals, lats);
if (status != CDF_OK) UserStatusHandler (status);
```

5.24 CDFvarInquire

CDFstatus CDFvarInquire( /* out -- Completion status code. */
    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 CDFcreateCDF) or CDFopen.

varNum
The number of the rVariable to inquire. This number may be determined with a call to
CDFvarNum (see Section 5.25).

varName
The rVariable's name. This character string must be large enough to hold
CDF_VAR_NAME_LEN256 + 1 characters (including the NUL terminator).

dataType
The data type of the rVariable. The data types are defined in Section 4.5.

numElements
The number of elements of the data type at each rVariable value. For character 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 data types.

recVariance
The record variance. The record variances are defined in Section 4.9.

dimVariance
The dimension variances. Each element of dimVariance receives the corresponding
dimension variance. The dimension variances are defined in Section 4.9. For 0-
dimensional rVariables this argument is ignored (but a placeholder is necessary).

5.24.1 Example(s)

The following example returns 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 CDFgetVarNum.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
char varName[CDF_VAR_NAME_LEN256+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, CDFgetVarNum(id,"HEAT_FLUX"), varName, &dataType,
&numElems, &recVary, dimVarys);
if (status != CDF_OK) UserStatusHandler (status);
```

55
5.25  **CDFvarNum**

long CDFvarNum( /* out -- Variable number. */
CDFid id,                   /* in -- CDF identifier. */
char *varName);            /* in -- Variable name. */

CDFvarNum is used to determine the number associated with a given variable name. If the variable is found, CDFvarNum returns its variable number - which will be equal to or greater than zero (0). If an error occurs (e.g., the variable does not exist in the CDF), an error code (of type CDFstatus) is returned. Error codes are less than zero (0). The returned variable number should be used in the functions of the same variable type, rVariable or zVariable. If it is an rVariable, functions dealing with rVariables should be used. Similarly, functions for zVariables should be used for zVariables.

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 CDFcreateCDF) or CDFopen.

- **varName**  
  The name of the variable to search. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator). Variable names are case-sensitive.

### 5.25.1 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_LEN256+1]; /* Variable 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 CDFgetVarNum 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

---

25 An original Standard Interface function. It used to handle only rVariables. It has been extended to include zVariables. While it is still available in V3.1, CDFgetVarNum is the preferred name for it.
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 5.24.

## 5.26 CDFvarPut

CDFstatus CDFvarPut( /* out -- Completion status code. */
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 writes a single data value to an rVariable. CDFvarPut may be used to write more than one value with a single call.

The arguments to CDFvarPut are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
- **varNum**: The rVariable number to which to write. This number may be determined with a call to CDFvarNum.
- **recNum**: The record number at which to write.
- **indices**: The dimension indices within the specified record at which to write. Each element of indices specifies the corresponding dimension index. For 0-dimensional variables, this argument is ignored (but must be present).
- **value**: The data value to write.

### 5.26.1 Example(s)

The following example will write two data values (1\textsuperscript{st} and 5\textsuperscript{th} elements) of a 2-dimensional rVariable (2 by 3) named MY_VAR to record number 0.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* rVariable number. */
long recNum; /* The record number. */
long indices[2]; /* The dimension indices. */
double value1, value2; /* The data values. */
```

---

\[26\] An original Standard Interface function, handling rVariables only.
```c
varNum = CDFgetVarNum(id, "MY_VAR");
if (varNum < CDF_OK) Quit ("....");
recNum = 0L;
indices[0] = 0L;
indices[1] = 0L;
value1 = 10.1;
status = CDFvarPut(id, varNum, recNum, indices, &value1);
if (status != CDF_OK) UserStatusHandler (status);
indices[0] = 1L;
indices[1] = 1L;
value2 = 20.2;
status = CDFvarPut(id, varNum, recNum, indices, &value2);
if (status != CDF_OK) UserStatusHandler (status);
```

5.27 **CDFvarRename**

`CDFstatus CDFvarRename(
  CDFid id,
  long varNum,
  char *varName);
`  

CDFvarRename is used to rename an existing rVariable. A variable (rVariable or zVariable) name must be unique.

The arguments to CDFvarRename are defined as follows:

- **id**
  - The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.

- **varNum**
  - The rVariable number to rename. This number may be determined with a call to CDFvarNum.

- **varName**
  - The new rVariable name. The maximum length of the new name is CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator). Variable names are case-sensitive.

5.27.1 **Example(s)**

In the following example the rVariable named TEMPERATURE is renamed to 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.

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

---

27 An original Standard Interface function, handling rVariables only.
CDFid id; /**< CDF identifier. */
CDFstatus status; /**< Returned status code. */
long varNum; /**< rVariable number. */

varNum = CDFvarNum (id, "TEMPERATURE");
if (varNum < CDF_OK) {
    if (varNum != NO_SUCH_VAR) UserStatusHandler (varNum);
}
else {
    status = CDFvarRename (id, varNum, "TMP");
    if (status != CDF_OK) UserStatusHandler (status);
}
Chapter 6

6 Exended Standard Interface

The following sections describe the new, extended set of Standard Interface functions callable from C applications that were added to CDF library since Version 3.1. Most functions return a status code of type CDFstatus (see Chapter 8). The Internal Interface is described in Chapter 7. An application can use either or both interfaces when necessary.

The original Standard Interface only provided a very limited functionality within the CDF library. For example, it could not handle zVariables and vAttribute zEntries (they were only accessible via the Internal Interface). Since V3.1, the Standard Interface has been expanded to include many new operations that are previously only available through the Internal Interface. The new functions in this chapter that deal with variables and variable attribute entries are only applicable to zVariables and variable attribute’s zEntries, not rVariables and rEntries. If you need to deal with rVariables for some reason (no need to use rVariables at all unless you are dealing with a CDF file that only contains rVariables), use the appropriate original Standard Interface routines in Chapter 5 or the Internal Interface in Chapter 7. Read Chapter 5 to understand why zVariables are recommended over the rVariables.

Each section begins with a function prototype for the routine being described. The include file cdf.h contains the same function prototypes (as well as function prototypes for the Internal Interface and EPOCH utility routines). Note that many of the Standard Interface functions in this chapter are implemented as macros (which call the Internal Interface).

The new functions, based on the operands, are grouped into four (4) categories: library, CDFs, variables and attributes/entries.

6.1 Library Information

The functions in this section are related to the current CDF library being used for the CDF operations, and they provide useful information such as the current library version number and Copyright notice.

6.1.1 CDFgetDataTypeSize

CDFstatus CDFgetDataTypeSize ( /* out -- Completion status code. */
long  dataType,           /* in -- CDF data type. */
long  *numBytes);         /* out -- Number of bytes for the given CDF type. */
CDFgetDataTypeSize returns the size (in bytes) of the specified CDF data type.

The arguments to CDFgetDataTypeSize are defined as follows:

- **dataType**  
  The CDF supported data type.

- **numBytes**  
  The size of dataType.

### 6.1.1.1. Example(s)

The following example returns the size of the data type CDF_INT4 that is 4 bytes.

```c
#include "cdf.h"

CDFstatus status;  /* Returned status code. */
long numBytes;  /* Number of bytes. */

status = CDFgetDataTypeSize((long)CDF_INT4, &numBytes);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.1.2  CDFgetLibraryCopyright

CDFstatus CDFgetLibraryCopyright ( /* out -- Completion status code. */
char *Copyright);  /* out -- Library Copyright. */

CDFgetLibraryCopyright returns the Copyright notice of the CDF library being used.

The arguments to CDFgetLibraryCopyright are defined as follows:

- **Copyright**  
  The Copyright notice. This character string must be large enough to hold CDF_COPYRIGHT_LEN + 1 characters (including the NUL terminator).

### 6.1.2.1. Example(s)

The following example returns the Copyright of the CDF library being used.

```c
#include "cdf.h"
```
6.1.3  CDFgetLibraryVersion

CDFgetLibraryVersion returns the version and release information of the CDF library being used.

The arguments to CDFgetLibraryVersion are defined as follows:

- **version**: The library version number.
- **release**: The library release number.
- **increment**: The library incremental number.
- **subIncrement**: The library sub-incremental character.

### 6.1.3.1. Example(s)

The following example returns the version and release information of the CDF library that is being used.

```c
#include "cdf.h"

long version;  /* CDF library version number. */
long release;  /* CDF library release number. */
long increment;  /* CDF library incremental number. */
char subIncrement;  /* CDF library sub-incremental character. */

status = CDFgetLibraryVersion(&version, &release, &increment, &subIncrement);
if (status != CDF_OK) UserStatusHandler (status);
```
6.1.4  CDFgetStatusText

CDFgetStatusText(  /* out -- Completion status code. */
CDFstatus status,  /* in -- The status code. */
char *message);  /* out -- The status text description. */

CDFgetStatusText is identical to the original Standard Interface function CDFerror (see section 5.12), and the use of this function is strongly encouraged over CDFerror as it might not be supported in the future. This function is used to inquire the text explanation of a given status code. Chapter 8 explains how to interpret status codes and Appendix A lists all of the possible status codes.

The arguments to CDFgetStatusText 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).

6.1.4.1. Example(s)

The following example displays the explanation text for the error code that is returned from a call to CDFopenCDF.

```
#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 = CDFopenCDF ("giss_wetl", &id);
if (status < CDF_WARN)  {  /* INFO and WARNING codes ignored. */
    CDFgetStatusText (status, text);
    printf ("ERROR> %s\n", text);
}
CDFcloseCDF (id);
```

6.2  CDF

The functions in this section provide CDF file-specific operations. Any operations involving variables or attributes are described in the following sections. This CDF has to be a newly created or opened from an existing one.
6.2.1 CDFcloseCDF

CDFstatus CDFcloseCDF (/* out -- Completion status code. */
CDFid id); /* in -- CDF identifier. */

CDFcloseCDF closes the specified CDF. This function is identical to the original Standard Interface function CDFclose (see section 5.8), and the use of this function is strongly encouraged over CDFclose as it might not be supported in the future. 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 CDFcloseCDF 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 CDFcloseCDF, the CDF's cache buffers are left unflushed.

The arguments to CDFcloseCDF are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreateCDF or CDFopenCDF.

6.2.1.1. Example(s)

The following example will close an open CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */

status = CDFopenCDF ("giss_wetl", &id);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFcloseCDF (id);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.2 CDFcreateCDF

CDFstatus CDFcreateCDF (/* out -- Completion status code. */
char *CDFname, /* in -- CDF file name. */
CDFid *id); /* out -- CDF identifier. */

CDFcreateCDF creates a CDF file. This function, a new and simple form of CDFcreate (see section 5.9 for details) without the encoding and majority arguments, works just like the CDF creation function from the Internal Interface.
The created CDF will use the default encoding (HOST_ENCODING) and majority (ROW_MAJOR), specified in the configuration file of your CDF distribution. 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 can either manually delete the file or open it with CDFopenCDF, delete it with CDFdeleteCDF, and then recreate it with CDFcreateCDF. If the existing CDF is corrupted, the call to CDFopenCDF will fail. (An error code will be returned.) In this case you must delete the CDF at the command line. Delete the dotCDF file (having an extension of .cdf), and if the CDF has the multi-file format, delete all of the variable files (having extensions of .v0,.v1,. . . and .z0,.z1,. . .).

Note that a CDF file created with CDFcreateCDF can only accept zVariables, not rVariables. But this is fine since zVariables are more flexible than rVariables. See the third paragraph of Chapter 5 for the differences between rVariables and zVariables.

The arguments to CDFcreateCDF are defined as follows:

- **CDFname**: The file name 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 file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

  **UNIX**: File names are case-sensitive.

- **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 CDFcreateCDF 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.

**NOTE**: CDFcloseCDF 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).

### 6.2.2.1. Example(s)

The following example creates a CDF named “test1.cdf” with the default encoding and majority.

```c
#include "cdf.h"

CDFid id;               /* CDF identifier. */
CDFstatus status;       /* Returned status code. */

status = CDFcreateCDF ("test1", &id);
if (status != CDF_OK) UserStatusHandler (status);

CDFclose (id);
```

66
6.2.3  CDFdeleteCDF

CDFstatus CDFdelete( /* out -- Completion status code. */
CDFid  id);      /* in -- CDF identifier. */

CDFdeleteCDF deletes the specified CDF. This function is identical to the original Standard Interface function CDFdelete (see section 5.10), and the use of this function is strongly encouraged over CDFdelete as it might not be supported in the future. The CDF files deleted include the dotCDF file (having an extension of .cdf), and if a multi-file CDF, the variable files (having extensions of .v0,.v1,... and .z0,.z1,...).

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 CDFdeleteCDF are defined as follows:

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

6.2.3.1. Example(s)

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

```c
#include "cdf.h"

CDFid  id; /* CDF identifier. */
CDFstatus  status; /* Returned status code. */

status = CDFopenCDF ("test2", &id);
if (status < CDF_OK) /* INFO status codes ignored. */
  UserStatusHandler (status);
else {
  status = CDFdeleteCDF (id);
  if (status != CDF_OK) UserStatusHandler (status);
}
```

6.2.4  CDFgetCacheSize

CDFstatus CDFgetCacheSize ( /* out -- Completion status code. */
CDFid  id,   /* in -- CDF identifier. */
long *numBuffers); /* out -- CDF’s cache buffers. */
CDFgetCacheSize returns the number of cache buffers being used for the dotCDF file when a CDF is open. Refer to the CDF User’s Guide for description of caching scheme used by the CDF library.

The arguments to CDFgetCacheSize are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreateCDF (or CDFcreate) or CDFopen.
- **numBuffers**: The number of cache buffers.

### 6.2.4.1. Example(s)

The following example returns the cache buffers for the open CDF file.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long numBuffers; /* CDF’s cache buffers. */

status = CDFgetCacheSize (id, &numBuffers);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.2.5 CDFgetChecksum

CDFstatus CDFgetChecksum (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *checksum); /* out -- CDF’s checksum mode. */

CDFgetChecksum returns the checksum mode of a CDF. The CDF checksum mode is described in Section 4.19.

The arguments to CDFgetChecksum are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreateCDF (or CDFcreate) or CDFopen.
- **checksum**: The checksum mode (NO_CHECKSUM or MD5_CHECKSUM).

### 6.2.5.1. Example(s)

The following example returns the checksum code for the open CDF file.


```
#include "cdf.h"

CDFid id;      /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long checksum;  /* CDF’s checksum. */

status = CDFgetChecksum (id, &checksum);
if (status != CDF_OK) UserStatusHandler (status);
```

**6.2.6 CDFgetCompression**

CDFstatus CDFgetCompression ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *compressionType, /* out -- CDF’s compression type. */
long compressionParms[], /* out -- CDF’s compression parameters. */
long *compressionPercentage); /* out -- CDF’s compressed percentage. */

CDFgetCompression gets the compression information of the CDF. It returns the compression type (method) and, if compressed, the compression parameters and compression percentage. The compression percentage is the result of the compressed file size divided by its original, uncompressed file size\(^{28}\). CDF compression types/parameters are described in Section 4.10.

The arguments to CDFgetCompression are defined as follows:

- **id** The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **compressionType** The type of the compression.
- **compressionParms** The parameters of the compression.
- **compressionPercentage** The compression percentage, the percentage of a uncompressed file size to hold the compressed data.

**6.2.6.1. Example(s)**

The following example returns the compression information of the open CDF file.

\(^{28}\) The compression ratio is \((100 – \text{compression percentage})\). The lower the compression percentage, the better the compression ratio.
#include "cdf.h"

CDFid id;       /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long compressionType; /* CDF’s compression type. */
long compressionParms[CDF_MAX_PARMS]; /* CDF’s compression parameters. */
long compressionPercentage; /* CDF’s compression rate. */

status = CDFgetCompression (id, &compression, compressionParms, &compressionPercentage);
if (status != CDF_OK) UserStatusHandler (status);
if (compressionType == NO_COMPRESSION) {

}

6.2.7 CDFgetCompressionCacheSize

CDFstatus CDFgetCompressionCacheSize (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *numBuffers); /* out -- CDF’s compressed cache buffers. */

CDFgetCompressionCacheSize gets the number of cache buffers used for the compression scratch CDF file. Refer to the CDF User’s Guide for description of caching scheme used by the CDF library.

The arguments to CDFgetCompressionCacheSize are defined as follows:

Id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numBuffers The number of cache buffers.

6.2.7.1. Example(s)

The following example returns the number of cache buffers used for the scratch file from the compressed CDF file.

#include "cdf.h"

CDFid id;       /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long numBuffers; /* CDF’s compression cache buffers. */
status = CDFgetCompressionCacheSize (id, &numBuffers);
if (status != CDF_OK) UserStatusHandler (status);

6.2.8 CDFgetCompressionInfo

CDFstatus CDFgetCompressionInfo (/* out -- Completion status code. */
    char *CDFname, /* in -- CDF name. */
    long *cType, /* out -- CDF compression type. */
    long cParms[], /* out -- CDF compression parameters. */
    OFF_T *cSize, /* out -- CDF compressed size. */
    OFF_T *uSize); /* out -- CDF decompressed size. */

CDFgetCompressionInfo returns the compression type/parameters of a CDF without having to open the CDF. This
refers to the compression of the CDF - not of any compressed variables.

The arguments to CDFgetCompressionInfo are defined as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDFname</td>
<td>The pathname of a CDF file without the .cdf file extension.</td>
</tr>
<tr>
<td>cType</td>
<td>The CDF compression type.</td>
</tr>
<tr>
<td>cParms</td>
<td>The CDF compression parameters.</td>
</tr>
<tr>
<td>cSize</td>
<td>The compressed CDF file size.</td>
</tr>
<tr>
<td>uSize</td>
<td>The size of CDF when decompress the originally compressed CDF.</td>
</tr>
</tbody>
</table>

6.2.8.1. Example(s)

The following example returns the compression information from a “unopen” CDF named “MY_TEST.cdf”.

```
#include "cdf.h"

CDFstatus status; /* Returned status code. */
long cType; /* Compression type. */
long cParms[CDF_MAX_PARMS]; /* Compression parameters. */
OFF_T cSize; /* Compressed file size. */
OFF_T uSize; /* Decompressed file size. */

status = CDFgetCompressionInfo("MY_TEST", &cType, cParms, &cSize, &uSize);
if (status != CDF_OK) UserStatusHandler (status);
```
6.2.9  CDFgetCopyright

CDFstatus CDFgetCopyright (    /* out -- Completion status code. */
CDFid id,                      /* in -- CDF identifier. */
char *Copyright);              /* out -- Copyright notice. */

CDFgetCopyright gets the Copyright notice in a CDF.

The arguments to CDFgetCopyright are defined as follows:

\begin{itemize}
  \item \textbf{id} \hspace{1cm} The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
  \item \textbf{Copyright} \hspace{1cm} CDF Copyright. This character string must be large enough to hold CDF_COPYRIGHT_LEN + 1 characters (including the NUL terminator).
\end{itemize}

6.2.9.1.  Example(s)

The following example returns the Copyright in a CDF.
\begin{verbatim}
#include "cdf.h"

CDFid id;       /* CDF identifier. */
CDFstatus status; /* Returned status code. */
char Copyright[CDF_COPYRIGHT_LEN+1]; /* CDF’s Copyright. */

status = CDFgetCopyright (id, Copyright);
if (status != CDF_OK) UserStatusHandler (status);
\end{verbatim}

6.2.10  CDFgetDecoding

CDFstatus CDFgetDecoding (    /* out -- Completion status code. */
CDFid id,                      /* in -- CDF identifier. */
long *decoding);              /* out -- CDF decoding. */

CDFgetDecoding returns the decoding code for the data in a CDF. The decodings are described in Section 4.7.

The arguments to CDFgetDecoding are defined as follows:

id
The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

decoding
The decoding of the CDF.

6.2.10.1. Example(s)
The following example returns the decoding for the CDF.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long decoding; /* Decoding. */

status = CDFgetDecoding(id, &decoding);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.11 CDFgetEncoding

CDFstatus CDFgetEncoding (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *encoding); /* out -- CDF encoding. */

CDFgetEncoding returns the data encoding used in a CDF. The encodings are described in Section 4.6.

The arguments to CDFgetEncoding are defined as follows:

id
The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

encoding
The encoding of the CDF.

6.2.11.1. Example(s)
The following example returns the data encoding used for the given CDF.

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

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long encoding; /* Encoding. */

status = CDFgetEncoding(id, &encoding);
if (status != CDF_OK) UserStatusHandler (status);

6.2.12  CDFgetFileBackward

int CDFgetFileBackward(/* out – File Backward Mode. */
    );

CDFgetFileBackward returns the backward mode information dealing with the creation of a new CDF file. A mode of value 1 indicates when a new CDF file is created, it will be a backward version of V2.7, not the current library version.

The arguments to CDFgetFileBackward are defined as follows:

N/A

6.2.12.1.  Example(s)

In the following example, the CDF’s file backward mode is acquired.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
int mode; /* Backward mode. */

mode = CDFgetFileBackward ();
if (mode == 1) {
    ...
}
```

6.2.13  CDFgetFormat

CDFstatus CDFgetFormat (/* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    );
long *format); /* out -- CDF format. */

CDFgetFormat returns the file format, single or multi-file, of the CDF. The formats are described in Section 4.4.

The arguments to CDFgetFormat are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

format The format of the CDF.

### 6.2.13.1. Example(s)

The following example returns the file format of the CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long format; /* Format. */

status = CDFgetFormat(id, &format);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.2.14 CDFgetMajority

CDFstatus CDFgetMajority ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *majority); /* out -- Variable majority. */

CDFgetMajority returns the variable majority, row or column-major, of the CDF. The majorities are described in Section 4.8.

The arguments to CDFgetMajority are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

majority The variable majority of the CDF.
6.2.14.1. Example(s)

The following example returns the majority of the CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long majority; /* Majority. */

status = CDFgetMajority (id, &majority);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.15 CDFgetName

CDFstatus CDFgetName (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
char *name); /* out -- CDF name. */

CDFgetName returns the file name of the specified CDF.

The arguments to CDFgetName are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **name**: The file name of the CDF.

6.2.15.1. Example(s)

The following example returns the name of the CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
char name[CDF_PATHNAME_LEN]; /* Name of the CDF. */

status = CDFgetName (id, name);
```
if (status != CDF_OK) UserStatusHandler (status);

6.2.16 CDFgetNegtoPosfp0Mode

CDFstatus CDFgetNegtoPosfp0Mode ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *negtoPosfp0); /* out -- -0.0 to 0.0 mode. */

CDFgetNegtoPosfp0Mode returns the –0.0 to 0.0 mode of the CDF. You can use CDFsetNegtoPosfp0 function to set the mode. The –0.0 to 0.0 modes are described in Section 4.15.

The arguments to CDFgetNegtoPosfp0Mode are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

negtoPosfp0 The –0.0 to 0.0 mode of the CDF.

6.2.16.1. Example(s)

The following example returns the –0.0 to 0.0 mode of the CDF.

#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long negtoPosfp0; /* -0.0 to 0.0 mode. */

status = CDFgetNegtoPosfp0Mode (id, &negtoPosfp0);
if (status != CDF_OK) UserStatusHandler (status);

6.2.17 CDFgetReadOnlyMode

CDFstatus CDFgetReadOnlyMode ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *readOnlyMode); /* out -- CDF read-only mode. */
CDFgetReadOnlyMode returns the read-only mode for a CDF. You can use CDFsetReadOnlyMode to set the mode of readOnlyMode. The read-only modes are described in Section 4.13.

The arguments to CDFgetReadOnlyMode are defined as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate or CDFcreateCDF or CDFopenCDF.</td>
</tr>
<tr>
<td>readOnlyMode</td>
<td>The read-only mode (READONLYon or READONLYoff).</td>
</tr>
</tbody>
</table>

6.2.17.1. Example(s)

The following example returns the read-only mode for the given CDF.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
long readMode;  /* CDF read-only mode. */

status = CDFgetReadOnlyMode (id, &readMode);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.18 CDFgetStageCacheSize

CDFstatus CDFgetStageCacheSize(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long *numBuffers);  /* out -- The stage cache size. */

CDFgetStageCacheSize returns the number of cache buffers being used for the staging scratch file a CDF. Refer to the CDF User's Guide for the description of the caching scheme used by the CDF library.

The arguments to CDFgetStageCacheSize are defined as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate or CDFcreateCDF or CDFopenCDF.</td>
</tr>
<tr>
<td>numBuffers</td>
<td>The number of cache buffers.</td>
</tr>
</tbody>
</table>

6.2.18.1. Example(s)

The following example returns the number of cache buffers used in a CDF.
#include "cdf.h"

CDFid id;  /* CDF identifier. */
long numBuffers;  /* The number of cache buffers. */

status = CDFgetStageCacheSize (id, &numBuffers);
if (status != CDF_OK) UserStatusHandler (status);

6.2.19  CDFgetValidate

int CDFgetValidate();

CDFgetValidate returns the data validation mode. This information reflects whether when a CDF is open, its certain data fields are subjected to a validation process. 1 is returned if the data validation is to be performed, 0 otherwise.

The arguments to CDFgetVersion are defined as follows:

N/A

6.2.19.1. Example(s)

In the following example, it gets the data validation mode.

#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
int validate;  /* Data validation flag. */

validate = CDFgetValidate();
6.2.20  CDFgetVersion

CDFstatus  CDFgetVersion(  /*  out -- Completion status code. */
CDFid  id,  /*  in -- CDF identifier. */
long  *version,  /*  out -- CDF version. */
long  *release,  /*  out -- CDF release. */
long  *increment);  /*  out -- CDF increment. */

CDFgetVersion returns the version/release information for a CDF file. This information reflects the CDF library that was used to create the CDF file.

The arguments to CDFgetVersion are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **version**: The CDF version number.
- **release**: The CDF release number.
- **increment**: The CDF increment number.

6.2.20.1. Example(s)

In the following example, a CDF’s version/release is acquired.

```c
#include "cdf.h"

CDFid  id;  /*  CDF identifier. */
CDFstatus  status;  /*  Returned status code. */
long  version;  /*  CDF version. */
long  release;  /*  CDF release */
long  increment;  /*  CDF increment. */

status = CDFgetVersion (id, &version, &release, &increment);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.21  CDFgetzMode

CDFstatus  CDFgetzMode(  /*  out -- Completion status code. */
CDFid  id,  /*  in -- CDF identifier. */
long  *zMode);  /*  out -- CDF zMode. */
CDFgetzMode returns the zMode for a CDF file. The zModes are described in Section 4.14.

The arguments to CDFgetzMode are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

zMode The CDF zMode.

6.2.21.1. Example(s)

In the following example, a CDF’s zMode is acquired.

```c
#include "cdf.h"

CDFid id; /* CDF identifier */
CDFstatus status; /* Returned status code */
long zMode; /* CDF zMode */

status = CDFgetzMode (id, &zMode);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.22 CDFinquireCDF

CDFstatus CDFinquireCDF(
    CDFid id, /* in -- CDF identifier */
    long *numDims, /* out -- Number of dimensions for rVariables */
    long *dimSizes[CDF_MAX_DIMS], /* out -- Dimension sizes for rVariables */
    long *encoding, /* out -- Data encoding */
    long *majority, /* out -- Variable majority */
    long *maxrRec, /* out -- Maximum record number among rVariables in the CDF */
    long *numrVars, /* out -- Number of rVariables in the CDF */
    long *maxzRec, /* out -- Maximum record number among zVariables in the CDF */
    long *numzVars, /* out -- Number of zVariables in the CDF */
    long *numAttrs); /* out -- Number of attributes in the CDF */

CDFinquireCDF returns the basic characteristics of a CDF. This function expands the original Standard Interface function CDFinquire by acquiring extra information regarding the zVariables. Knowing the variable majority can be used to optimize performance and is necessary to properly use the variable hyper-get/put functions.

The arguments to CDFinquireCDF are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
numDims The number of dimensions for the rVariables in the CDF. Note that all the rVariables’ dimensionality in the same CDF file must be the same.

dimSizes The dimension sizes of the rVariables in the CDF (note that all the rVariables’ dimension sizes in the same CDF file must be the same). dimSizes is a 1-dimensional array containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional rVariables this argument is ignored (but must be present).

encoding The encoding of the variable data and attribute entry data. The encodings are defined in Section 4.6.

majority The majority of the variable data. The majorities are defined in Section 4.8.

maxrRec 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.

numrVars The number of rVariables in the CDF.

maxzRec The maximum record number written to a zVariable in the CDF. Note that the maximum record number written is also kept separately for each zVariable in the CDF. The value of maxRec is the largest of these. Some zVariables may have fewer records than actually written. Use CDFgetzVarMaxWrittenRecNum to inquire the actual number of records written for an individual zVariable.

numzVars The number of zVariables in the CDF.

numAttrs The number of attributes in the CDF.

6.2.22.1. Example(s)

The following example returns the basic information about a CDF.

```c
#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 maxrRec; /* Maximum record number, rVariables. */
long numrVars; /* Number of rVariables in CDF. */
long maxzRec; /* Maximum record number, zVariables. */
long numzVars; /* Number of zVariables in CDF. */
long numAttrs; /* Number of attributes in CDF. */
```
status = CDFinquireCDF (id, &numDims, dimSizes, &encoding, &majority,
    &maxrRec, &numrVars, &maxzRec, &numzVars, &numAttrs);
if (status != CDF_OK) UserStatusHandler (status);

6.2.23  CDFopenCDF

CDFstatus CDFopenCDF(/* out -- Completion status code. */
    char *CDFname, /* in -- CDF file name. */
    CDFid *id); /* out -- CDF identifier. */

CDFopenCDF opens an existing CDF. This function is identical to the original Standard Interface function CDFopen (see section 5.16), and the use of this function is strongly encouraged over CDFopen as it might not be supported in the future. 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 CDFopenCDF are defined as follows:

**CDFname**  The file name 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 file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

**UNIX:** File names are case-sensitive.

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

**NOTE:** CDFcloseCDF must be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk.

6.2.23.1. Example(s)

The following example will open a CDF named “NOAA1.cdf”.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
static char CDFname[] = { "NOAA1" }; /* file name of CDF. */

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

6.2.24  CDFsetCacheSize

CDFstatus CDFsetCacheSize (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier */
long numBuffers); /* in -- CDF’s cache buffers */

CDFsetCacheSize specifies the number of cache buffers being used for the dotCDF file when a CDF is open. Refer to the CDF User’s Guide for the description of the cache scheme used by the CDF library.

The arguments to CDFsetCacheSize are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numBuffers The number of cache buffers.

6.2.24.1. Example(s)

The following example extends the number of cache buffers to 500 for the open CDF file. The default number is 300 for a single-file format CDF on Unix systems.

```
#include "cdf.h"

CDFid id; /* CDF identifier */
CDFstatus status; /* Returned status code */
long cacheBuffers; /* CDF’s cache buffers */

cacheBuffers = 500L;
status = CDFsetCacheSize (id, cacheBuffers);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.25  CDFsetChecksum

CDFstatus CDFsetChecksum (/* out -- Completion status code. */
CDFsetChecksum specifies the checksum mode for the CDF. The CDF checksum mode is described in Section 4.19.

The arguments to CDFsetChecksum are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

checksum The checksum mode (NO_CHECKSUM or MD5_CHECKSUM).

### 6.2.25.1. Example(s)

The following example turns off the checksum flag for the open CDF file.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long checksum; /* CDF’s checksum. */

checksum = NO_CHECKSUM;
status = CDFsetChecksum (id, checksum);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.2.26 CDFsetCompression

CDFstatus CDFsetCompression ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long compressionType, /* in -- CDF’s compression type. */
long compressionParms[]); /* in -- CDF’s compression parameters. */

CDFsetCompression specifies the compression type and parameters for a CDF. This compression refers to the CDF, not of any variables. The compressions are described in Section 4.10.

The arguments to CDFsetCompression are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

compressionType The compression type.
compressionParms  The compression parameters.

6.2.26.1. Example(s)

The following example uses GZIP.9 to compress the CDF file.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long compressionType; /* CDF’s compression type. */
long compressionParms[CDF_MAX_PARMS] /* CDF’s compression parameters. */

compressionType = GZIP_COMPRESSION;
compressionParms[0] = 9L;
status = CDFsetCompression (id, compressionType, compressionParms);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.27  CDFsetCompressionCacheSize

CDFstatus CDFsetCompressionCacheSize (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long compressionNumBuffers); /* in -- CDF’s compressed cache buffers. */

CDFsetCompressionCacheSize specifies the number of cache buffers used for the compression scratch CDF file. Refer to the CDF User’s Guide for the description of the cache scheme used by the CDF library.

The arguments to CDFsetCompressionCacheSize are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **compressionNumBuffers**: The number of cache buffers.

6.2.27.1. Example(s)

The following example extends the number of cache buffers used for the scratch file from the compressed CDF file to 100. The default cache buffers is 80 for Unix systems.
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long compressionNumBuffers; /* CDF’s compression cache buffers. */

compressionNumBuffers = 100L;
status = CDFsetCompressionCacheSize (id, compressionNumBuffers);
if (status != CDF_OK) UserStatusHandler (status);

6.2.28 CDFsetDecoding

CDFstatus CDFsetDecoding (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long decoding); /* in -- CDF decoding. */

CDFsetDecoding sets the decoding of a CDF. The decodings are described in Section 4.7.

The arguments to CDFsetDecoding are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.

decoding The decoding of a CDF.

6.2.28.1. Example(s)

The following example sets NETWORK_DECODING to be the decoding scheme in the CDF.

#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long decoding; /* Decoding. */

decoding = NETWORK_DECODING;
status = CDFsetDecoding (id, decoding);
if (status != CDF_OK) UserStatusHandler (status);
6.2.29  CDFsetEncoding

CDFstatus CDFsetEncoding (   /* out -- Completion status code. */
CDFid id,                   /* in -- CDF identifier. */
long encoding);             /* in -- CDF encoding. */

CDFsetEncoding specifies the data encoding of the CDF. A CDF’s encoding may not be changed after any variable values have been written. The encodings are described in Section 4.6.

The arguments to CDFsetEncoding are defined as follows:

id                  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

encoding            The encoding of the CDF.

6.2.29.1. Example(s)

The following example sets the encoding to HOST_ENCODING for the CDF.

```
#include "cdf.h"

CDFid id;         /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long encoding;    /* Encoding. */

encoding = HOST_ENCODING;
status = CDFsetEncoding(id, encoding);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.30  CDFsetFileBackward

void CDFsetFileBackward(   /* in -- File backward Mode. */
long mode)

CDFsetFileBackward sets the backward mode. When the mode is set as FILEBACKWARDOn, any new CDF files created are of version 2.7, instead of the underlining library version. If mode FILEBACKWARDOff is used, the default for creating new CDF files, the library version is the version of the file.

The arguments to CDFsetFileBackward are defined as follows:
mode The backward mode.

6.2.30.1. Example(s)

In the following example, it sets the file backward mode to FILEBACKWARDoff, which means that any files to be created will be of version V3.*, the same as the library version.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */

CDFsetFileBackward (FILEBACKWARDoff);
```

6.2.31 CDFsetFormat

CDFstatus CDFsetFormat ( /* out -- Completion status code. */
CFD id, /* in -- CDF identifier. */
long format); /* in -- CDF format. */

CDFsetFormat specifies the file format, either single or multi-file format, of the CDF. A CDF’s format may not be changed after any variable values have been written. The formats are described in Section 4.4.

The arguments to CDFsetFormat are defined as follows:

- `id` The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- `format` The file format of the CDF.

6.2.31.1. Example(s)

The following example sets the file format to MULTI_FILE for the CDF. The default is SINGLE_FILE format.

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

format = MULTI_FILE;  
status = CDFsetFormat(id, format);  
if (status != CDF_OK) UserStatusHandler (status);

6.2.32 CDFsetMajority

CDFstatus CDFsetMajority (/* out -- Completion status code. */  
CDFid id, /* in -- CDF identifier. */  
long majority); /* in -- CDF variable majority. */

CDFsetMajority specifies the variable majority, either row or column-major, of the CDF. A CDF’s majority may not be changed after any variable values have been written. The majorities are described in Section 4.8.

The arguments to CDFsetMajority are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

majority The variable majority of the CDF.

6.2.32.1. Example(s)

The following example sets the majority to COLUMN_MAJOR for the CDF. The default is ROW_MAJOR.

#include "cdf.h"

CDFid id; /* CDF identifier. */  
CDFstatus status; /* Returned status code. */  
long majority; /* Majority. */

majority = COLUMN_MAJOR;  
status = CDFsetMajority (id, majority);  
if (status != CDF_OK) UserStatusHandler (status);

90
6.2.33  CDFsetNegtoPosfp0Mode

CDFstatus CDFsetNegtoPosfp0Mode ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long negtoPosfp0); /* in -- -0.0 to 0.0 mode. */

CDFsetNegtoPosfp0Mode specifies the –0.0 to 0.0 mode of the CDF. The –0.0 to 0.0 modes are described in Section 4.15.

The arguments to CDFsetNegtoPosfp0Mode are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- negtoPosfp0: The –0.0 to 0.0 mode of the CDF.

6.2.33.1. Example(s)

The following example sets the –0.0 to 0.0 mode to ON for the CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long negtoPosfp0; /* -0.0 to 0.0 mode. */

negtoPosfp0 = NEGtoPOSfp0on;
status = CDFsetNegtoPosfp0Mode (id, negtoPosfp0);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.34  CDFsetReadOnlyMode

CDFstatus CDFsetReadOnlyMode ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long readOnlyMode); /* in -- CDF read-only mode. */

CDFsetReadOnlyMode specifies the read-only mode for a CDF. The read-only modes are described in Section 4.13.

The arguments to CDFsetReadOnlyMode are defined as follows:
id
The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

readOnlyMode
The read-only mode.

6.2.34.1. Example(s)

The following example sets the read-only mode to OFF for the CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long readMode; /* CDF read-only mode. */

readMode = READONLYoff;
status = CDFsetReadOnlyMode (id, readMode);
if (status != CDF_OK) UserStatusHandler (status);
```

6.2.35 CDFsetStageCacheSize

CDFstatus CDFsetStageCacheSize(/* out -- Completion status code. */
CFDid id, /* in -- CDF identifier. */
long numBuffers); /* in -- The stage cache size. */

CDFsetStageCacheSize specifies the number of cache buffers being used for the staging scratch file a CDF. Refer to the CDF User’s Guide for the description of the caching scheme used by the CDF library.

The arguments to CDFsetStageCacheSize are defined as follows:

id
The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numBuffers
The number of cache buffers.

6.2.35.1. Example(s)

The following example sets the number of stage cache buffers to 10 for a CDF.
#include "cdf.h"

CDFid id; /* CDF identifier. */
long numBuffers; /* The number of cache buffers. */

numBuffers = 10L;
status = CDFsetStageCacheSize (id, numBuffers);
if (status != CDF_OK) UserStatusHandler (status);

6.2.36 CDFsetValidate

void CDFsetValidate(
    long   mode);  /* in -- File Validation Mode. */

CDFsetValidate sets the data validation mode. The validation mode dedicates whether certain data in an open CDF file will be validated. This mode should be set before the any files are opened. Refer to Data Validation Section 4.20.

The arguments to CDFgetVersion are defined as follows:

    mode            The validation mode.

6.2.36.1. Example(s)

In the following example, it sets the validation mode to be on, so any following CDF files are subjected to the data validation process when they are open.

    #include "cdf.h"

    CDFid id;  /* CDF identifier. */
    CDFstatus status;  /* Returned status code. */

    CDFsetValidate (VALIDATEFILEon);

6.2.37 CDFsetzMode

CDFstatus CDFsetzMode(  /* out -- Completion status code. */
CDFsetzMode specifies the zMode for a CDF file. The zModes are described in Section 4.14 and see the Concepts chapter in the CDF User’s Guide for a more detailed information on zModes. zMode is used when dealing with a CDF file that contains 1) rVariables or 2) rVariables and zVariables. If you want to treat rVariables as zVariables, it’s highly recommended to set the value of zMode to zMODEon2.

The arguments to CDFsetzMode are defined as follows:

- **id**
  - The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **zMode**
  - The CDF zMode.

### 6.2.37.1. Example(s)

In the following example, a CDF’s zMode is specified to zMODEon2: all rVariables are treated as zVariables with NOVARY dimensions being eliminated.

```c
#include "cdf.h"

CDFid id;    /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long zMode;  /* CDF zMode. */

zMode = zMODEon2;
status = CDFsetzMode (id, zMode);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3 Variable

The functions in this section provides CDF variable-specific functions. A variable is identified by its unique name in a CDF or a variable number. Before you can perform any operation on a variable, the CDF in which it resides in must be opened.

#### 6.3.1 CDFclosezVar

CDFstatus CDFclosezVar(  /* out -- Completion status code. */
CDFid id,          /* in -- CDF identifier. */
long varNum)      /* in -- zVariable number. */
CDFclosezVar closes the specified zVariable file from a multi-file format CDF. Note that zVariables in a single-file CDF don’t need to be closed. The variable's cache buffers are flushed before the variable's open file is closed. However, the CDF file is still open.

**NOTE:** For the multi-file CDF, you must close all open variable files 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 CDFcloseCDF, the CDF's cache buffers are left unflushed.

The arguments to CDFclosezVar are defined as follows:

- **id** The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum** The variable number for the open zVariable’s file. This identifier must have been initialized by a call to CDFcreatezVar or CDFgetVarNum.

### 6.3.1.1. Example(s)

The following example will close an open zVariable file from a multi-file CDF.

```c
#include "cdf.h"

CDFid id;          /*  CDF identifier. */
CDFstatus status;  /*  Returned status code. */
long varNum;       /*  zVariable number. */

varNum = CDFgetVarNum (id, "VAR_NAME1");
if (varNum < CDF_OK) QuitError(…….);

status = CDFclosezVar (id, varNum);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.2 CDFconfirmzVarExistence

PDFstatus CDFconfirmzVarExistence(  /* out -- Completion status code. */
CDFid id,                /* in -- CDF identifier. */
Char *varName);           /* in -- zVariable name. */

CDFconfirmzVarExistence confirms the existence of a zVariable with a given name in a CDF. If the zVariable does not exist, an error code will be returned.

The arguments to CDFconfirmzVarExistence are defined as follows:
id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varName The zVariable name to check.

6.3.2.1. Example(s)

The following example checks the existence of zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */

status = CDFconfirmzVarExistence (id, "MY_VAR");
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.3 CDFconfirmzVarPadValueExistence

CDFstatus CDFconfirmzVarPadValueExistence( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum) /* in -- zVariable number. */

CDFconfirmzVarPadValueExistence confirms the existence of an explicitly specified pad value for the specified zVariable in a CDF. If an explicit pad value has not been specified, the informational status code NO_PADVALUE_SPECIFIED will be returned.

The arguments to CDFconfirmzVarPadValueExistence are defined as follows:

- id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum The zVariable number.

6.3.3.1. Example(s)

The following example checks the existence of the pad value of zVariable “MY_VAR” in a CDF.

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

varNum = CDFgetVarNum(id, “MY_VAR”);
if (varNum < CDF_OK) QuitError(…);
status = CDFconfirmzVarPadValueExistence (id, varNum);
if (status != NO_PADVALUE_SPECIFIED) {
}

6.3.4 CDFcreatezVar

CDFstatus CDFcreatezVar( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
char *varName, /* in -- zVariable name. */
long dataType, /* in -- Data type. */
long numElements, /* in -- Number of elements (of the data type). */
long numDims, /* in -- Number of dimensions. */
long dimSizes[], /* in -- Dimension sizes */
long recVariance, /* in -- Record variance. */
long dimVariances[], /* in -- Dimension variances. */
long *varNum); /* out -- zVariable number. */

CDFcreatezVar is used to create a new zVariable in a CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF.

The arguments to CDFcreatezVar are defined as follows:

id
The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varName
The name of the zVariable to create. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator). Variable names are case-sensitive.

dataType
The data type of the new zVariable. Specify one of the data types defined in Section 4.5.

numElements
The number of elements of the data type at each value. For character 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 data types.

numDims
Number of dimensions the zVariable. This may be as few as zero (0) and at most CDF_MAX_DIMS.
**dimSizes**
The size of each dimension. Each element of dimSizes specifies the corresponding dimension size. Each size must be greater than zero (0). For 0-dimensional zVariables this argument is ignored (but must be present).

**recVariance**
The zVariable's record variance. Specify one of the variances defined in Section 4.9.

**dimVariances**
The zVariable's dimension variances. Each element of dimVariances specifies the corresponding dimension variance. For each dimension specify one of the variances defined in Section 4.9. For 0-dimensional zVariables this argument is ignored (but must be present).

**varNum**
The 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 CDFgetVarNum function.

### 6.3.4.1. Example(s)

The following example will create several zVariables in a CDF. In this case EPOCH is a 0-dimensional, LAT and LON are 2-dimensional, and TMP is a 1-dimensional.

```c
#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[1] = {NOVARY}; /* EPOCH dimension variances. */
static long LATdimVarys[2] = {VARY,VARY}; /* LAT dimension variances. */
static long LONdimVarys[2] = {VARY,VARY}; /* LON dimension variances. */
static long TMPdimVarys[2] = {VARY,VARY}; /* TMP dimension variances. */
long EPOCHvarNum; /* EPOCH zVariable number. */
long LATvarNum; /* LAT zVariable number. */
long LONvarNum; /* LON zVariable number. */
long TMPvarNum; /* TMP zVariable number. */
static long EPOCHdimSizes[1] = {3}; /* EPOCH dimension sizes. */
static long LATLONdimSizes[2] = {2,3} /* LAT/LON dimension sizes. */
static long TMPdimSizes[1] = {3}; /* TMP dimension sizes. */

status = CDFcreatezVar (id, "EPOCH", CDF_EPOCH, 1, 0L, EPOCHdimSizes, EPOCHrecVary, EPOCHdimVarys, &EPOCHvarNum);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFcreatezVar (id, "LATITUDE", CDF_INT2, 1, 2L, LATLONdimSizes, LATrecVary, LATdimVarys, &LATvarNum);
if (status != CDF_OK) UserStatusHandler (status);

98
status = CDFcreatezVar (id, "LONGITUDE", CDF_INT2, 1, 2L, LATLONdimSizes, LONrecVary, LONdimVarys, &LONvarNum);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFcreatezVar (id, "TEMPERATURE", CDF_REAL4, 1, 1L, TMPdimSizes, TMPrecVary, TMPdimVarys, &TMPvarNum);
if (status != CDF_OK) UserStatusHandler (status);

6.3.5  CDFdeletezVar

CDFstatus CDFdeletezVar(
    CDFid  id,           /*  in   --  CDF identifier. */
    long varNum);       /*  in   --  zVariable identifier. */

CDFdeletezVar deletes the specified zVariable from a CDF.

The arguments to CDFdeletezVar are defined as follows:

id        The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or
           CDFcreateCDF) or CDFopenCDF.

varNum    The zVariable number to be deleted.

6.3.5.1.  Example(s)

The following example deletes the zVariable named MY_VAR in a CDF.

```
#include "cdf.h"

CDFid   id;             /*  CDF identifier. */
CDFstatus status;       /*  Returned status code. */
long    varNum;         /*  zVariable number. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) QuitError(…);
status = CDFdeletezVar (id, varNum);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.6  CDFdeletezVarRecords

CDFstatus CDFdeletezVarRecords(/* out -- Completion status code. */
CDFid  id,        /* in -- CDF identifier. */
long varNum,     /* in -- zVariable identifier. */
long startRec,   /* in -- Starting record number. */
long endRec);    /* in -- Ending record number. */

CDFdeletezVarRecords deletes a range of data records from the specified zVariable in a CDF.

The arguments to CDFdeletezVarRecords are defined as follows:

- **id**  The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**  The identifier of the zVariable.
- **startRec**  The starting record number to delete.
- **endRec**  The ending record number to delete.

6.3.6.1.  Example(s)

The following example deletes 11 records (from record numbered 11 to 21) from the zVariable “MY_VAR” in a CDF. Note: The first record is numbered as 0.

```c
#include "cdf.h"

CDFid  id;        /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long varNum;      /* zVariable number. */
long startRec;    /* Starting record number. */
long endRec;      /* Ending record number. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) QuitError(…);
startRec = 10L;
endRec = 20L;
status = CDFdeletezVarRecords (id, varNum, startRec, endRec);
if (status != CDF_OK) UserStatusHandler (status);
```
6.3.7 CDFgetMaxWrittenRecNums

CDFstatus CDFgetMaxWrittenRecNums (  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long *rVarsMaxNum,  /* out -- Maximum record number among all rVariables. */
long *zVarsMaxNum);  /* out -- Maximum record number among all zVariables. */

CDFgetMaxWrittenRecNums returns the maximum written record number for the rVariables and zVariables in a CDF. The maximum record number for rVariables or zVariables is one less than the maximum number of records among all respective variables.

The arguments to CDFgetMaxWrittenRecNums are defined as follows:

id          The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

rVarsMaxNum  The maximum record number among all rVariables.

zVarsMaxNum  The maximum record number among all zVariables.

6.3.7.1. Example(s)

The following example returns the maximum written record numbers among all rVariables and zVariables of the CDF.

```
#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long rVarsMaxNum;  /* Maximum record number among all rVariables. */
long zVarsMaxNum;  /* Maximum record number among all zVariables. */

status = CDFgetMaxWrittenRecNums (id, &rVarsMaxNum, &zVarsMaxNum);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.8 CDFgetNumrVars

CDFstatus CDFgetNumrVars (  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long *numVars);  /* out -- Total number of rVariables. */

CDFgetNumrVars returns the total number of rVariables in a CDF.

The arguments to CDFgetNumrVars are defined as follows:
id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numVars The number of rVariables.

6.3.8.1. Example(s)

The following example returns the total number of rVariables in a CDF.

```c
#include "cdf.h"

CDFstatus status; /* Returned status code. */
CDFid id;    /* CDF identifier. */
long numVars; /* Number of rVariables. */

status = CDFgetNumrVars (id, &numVars);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.9 CDFgetNumzVars

CDFstatus CDFgetNumzVars (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *numVars); /* out -- Total number of zVariables. */

CDFgetNumzVars returns the total number of zVariables in a CDF.

The arguments to CDFgetNumzVars are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numVars The number of zVariables.

6.3.9.1. Example(s)

The following example returns the total number of zVariables in a CDF.

```c
```
6.3.10 CDFgetVarAllRecordsByVarName

CDFstatus CDFgetVarAllRecordsByVarName(CDFid id, char *varName, void *buffer);

CDFgetVarAllRecordsByVarName reads the whole records from the specified variable in a CDF. This function provides an easier way of getting all data from a variable. Since a variable name is unique in a CDF, this function can be used for either an rVariable or zVariable. For zVariable, this function is similar to CDFgetzVarAllRecordsByVarID, which requires the zVariable id, instead. Make sure that the buffer is big enough to hold the data. Otherwise, a segmentation fault may happen.

The arguments to CDFgetVarAllRecordsByVarName are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varName The variable’s name.

buffer The buffer that holds the returned data.

6.3.10.1. Example(s)

The following example returns the whole record data for zVariable “MY_VAR” in a CDF.

Assuming that the variable has 100 records, each record being a 1-dimensional, with 3 elements, of double type.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
double buffer[100][3]; /* The buffer holding the data. */
```
status = CDFgetVarAllRecordsByVarName (id, “MY_VAR”, buffer);
if (status != CDF_OK) UserStatusHandler (status);

A more general approach: for a variable of double type, but not knowing the total number of records, number of
dimensions, etc:

#include "cdf.h"

CDFid id;    /*  CDF identifier. */
long varNum; /*  zVariable number. */
long numRecs; /*  Number of written records. */
long numDims; /*  Number of zVariable’s dimensions. */
long dimSizes[CDF_MAX_DIMS]; /*  zVariable’s dimensioality. */
long numValues; /*  Total number of values. */
double *buffer; /*  The buffer holding the data. */

varNum = CDFgetVarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“….”);

status = CDFgetzVarMaxWrittenRecNum (id, varNum, &numRecs);
if (status != CDF_OK) ….
status = CDFgetzVarNumDims (id, varNum, &numDims);
if (status != CDF_OK) ….
status = CDFgetzVarDimSizes (id, varNum, dimSizes);
if (status != CDF_OK) ….
numValues = 1;
for (i=1; i<numDims;++i) numValues *= dimSizes[i];
numvalue *= numRecs;
buffer = (double *) malloc((sizeof(double) * (size_t) numValues);
status = CDFgetVarAllRecordsByVarName (id, “MY_VAR”, buffer);
if (status != CDF_OK) UserStatusHandler (status);

free (buffer);

6.3.11 CDFgetVarNum

long CDFgetVarNum (    /* out -- Variable number. */
    CDFid id,    /* in -- CDF identifier. */
    char *varName); /* in -- Variable name. */

29 Expanded from the original Standard Interface function CDFvarNum that returns the rVariable number. Since no two
variables, either rVariable or zVariable, can have the same name, this function now returns the variable number for the
given rVariable or zVariable name (if the variable name exists in a CDF).
CDFgetVarNum returns the variable number for the given variable name (rVariable or zVariable). If the variable is found, CDFgetVarNum returns its variable number - which will be equal to or greater than zero (0). If an error occurs (e.g., the variable does not exist in the CDF), an error code (of type CDFstatus) is returned. Error codes are less than zero (0). The returned variable number should be used in the functions of the same variable type, rVariable or zVariable. If it is an rVariable, functions dealing with rVariables should be used. Similarly, functions for zVariables should be used for zVariables.

The arguments to CDFgetVarNum are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varName**: The name of the variable to search. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator). Variable names are case-sensitive.

CDFgetVarNum may be used as an embedded function call where an rVariable or zVariable number is needed.

### 6.3.11.1. Example(s)

In the following example CDFgetVarNum is used as an embedded function call when inquiring about a zVariable.

```c
#include "cdf.h"

CDFid id;
/* CDF identifier. */
CDFstatus status;
/* Returned status code. */
char varName[CDF_VAR_NAME_LEN256+1];
/* Variable name. */
long dataType;
/* Data type of the zVariable. */
long numElements;
/* Number of elements (of the data type). */
long numDims;
/* Number of dimensions. */
long dimSizes[CDF_MAX_DIMS];
/* Dimension sizes. */
long recVariance;
/* Record variance. */
long dimVariances[CDF_MAX_DIMS];
/* Dimension variances. */

status = CDFinquirezVar (id, CDFgetVarNum(id,"LATITUDE"), varName, &dataType, 
&numElements, &numDims, dimSizes, &recVariance, dimVariances);
if (status != CDF_OK) UserStatusHandler (status);
```

In this example the zVariable named LATITUDE was inquired. Note that if LATITUDE did not exist in the CDF, the call to CDFgetVarNum would have returned an error code. Passing that error code to CDFinquirezVar as a zVariable number would have resulted in CDFinquirezVar also returning an error code. Also note that the name written into varName is already known (LATITUDE). In some cases the zVariable names will be unknown - CDFinquirezVar would be used to determine them. CDFinquirezVar is described in Section 6.3.40.
6.3.12 CDGetVarRangeRecordsByVarName

CDFstatus CDGetVarRangeRecordsByVarName( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
char *varName, /* in -- Variable name. */
long startRec, /* in -- Starting record number. */
long stopRec, /* in -- Stopping record number. */
void *buffer); /* out -- Buffer for the returned record data. */

CDGetVarRangeRecordsByVarName reads a range of records from the specified variable in a CDF. This function provides an easier way of getting data from a variable. Since a variable name is unique in a CDF, this function can be used by either an rVariable or zVariable. For zVariable, this function is similar to CDGetzVarRangeRecordsByVarID, only it requires the variable’s id. Make sure that the buffer is big enough to hold the data. Otherwise, a segmentation fault may happen.

The arguments to CDGetVarRangeRecordsByVarName are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDCreate (or CDCreateCDF) or CDOpenCDF.
- varName: The variable name.
- startRec: The zero-based starting record number.
- stopRec: The zero-based stopping record number.
- buffer: The buffer that holds the returned data.

6.3.12.1. Example(s)

The following example reads the 100 record data, from record number 10 to 109 for zVariable “MY_VAR” in a CDF. Assuming each record is a 1-dimensional, with 3 elements, of double type.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
double buffer[100][3]; /* The buffer holding the data. */

status = CDGetVarRangeRecordsByVarName (id, "MY_VAR", 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);
```

More general approach: for a variable of double type:

```
#include "cdf.h"

status = CDGetzVarRangeRecordsByVarID (id, "MY_VAR", 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);
```
CDFid id;
long varNum;
long numDims;
long dimSizes[CDF_MAX_DIMS];
long numValues;
double *buffer;

varNum = CDFgetVarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“…”);

status = CDFgetVarNumDims (id, varNum, &numDims);
if (status != CDF_OK) ….
status = CDFgetVarDimSizes (id, varNum, dimSizes);
if (status != CDF_OK) ….
numValues = 1;
for (i=1; i<numDims;++i) numValues *= dimSizes[i];
numvalue *= (109-10+1);
buffer = (double *) malloc((sizeof(double) * (size_t) numValues);
status = CDFgetVarRangeRecordsByVarName (id, “MY_VAR”, 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);

free (buffer);

6.3.13  CDFgetzVarAllocRecords

CDFstatus CDFgetzVarAllocRecords(
CDFid id,
long varNum,
long *numRecs);

CDFgetzVarAllocRecords returns the number of records allocated for the specified zVariable in a CDF. Refer to the CDF User’s Guide for a description of allocating variable records in a single-file CDF.

The arguments to CDFgetzVarAllocRecords are defined as follows:

id  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum  The zVariable number.

numRecs  The number of allocated records.

6.3.13.1. Example(s)

The following example returns the number of allocated records for zVariable “MY_VAR” in a CDF.
```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
long varNum;  /* zVariable number. */
long numRecs;  /* The allocated records. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK)  Quit ("...");

status = CDFgetzVarAllocRecords (id, varNum, &numRecs);
if (status != CDF_OK) UserStatusHandler (status);

6.3.14  CDFgetzVarAllRecordsByVarID

CDFstatus CDFgetzVarAllRecordsByVarID(
    CDFid id,  /* in -- CDF identifier. */
    long varNum,  /* in -- zVariable number. */
    void *buffer);  /* out -- Buffer for thre returned record data. */

CDFgetzVarAllRecordsByVarID reads the whole records from the specified zVariable in a CDF. This function provides an easier way of getting all data from a variable. Make sure that the buffer is big enough to hold the data. Otherwise, a segmentation fault may happen.

The arguments to CDFgetzVarAllRecordsByVarID are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum The zVariable number.

buffer The buffer that holds the returned data.

6.3.14.1. Example(s)

The following example returns the whole record data for zVariable “MY_VAR” in a CDF.

Assuming that the variable has 100 records, each record being a 1-dimensional, with 3 elements, of double type.

```
More general approach: for a variable of double type, but not knowing the total number of records, number of dimensions, etc:  

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */ long varNum; /* zVariable number. */ long numRecs; /* Number of written records. */ long numDims; /* Number of zVariable's dimensions. */ long dimSizes[CDF_MAX_DIMS]; /* zVariable's dimensionality. */ long numValues; /* Total number of values. */ double *buffer; /* The buffer holding the data. */

varNum = CDFgetVarNum (id, "MY_VAR"); if (varNum < CDF_OK) Quit ("....");

status = CDFgetzVarMaxWrittenRecNum (id, varNum, &numRecs); if (status != CDF_OK) ....
status = CDFgetzVarNumDims (id, varNum, &numDims); if (status != CDF_OK) ....
status = CDFgetzVarDimSizes (id, varNum, dimSizes); if (status != CDF_OK) ....
numValues = 1; for (i=1; i<numDims;++i) numValues *= dimSizes[i]; numvalue *= numRecs;
buffer = (double *) malloc((sizeof(double) * (size_t) numValues);
status = CDFgetzVarAllRecordsByVarID (id, varNum, buffer); if (status != CDF_OK) UserStatusHandler (status);

free (buffer);
```

### 6.3.15 CDFgetzVarBlockingFactor

CDFstatus CDFgetzVarBlockingFactor( /* out -- Completion status code. */ CDFid id, /* in -- CDF identifier. */ long varNum, /* in -- Variable number. */
CDFgetzVarBlockingFactor returns the blocking factor for the specified zVariable in a CDF. Refer to the CDF User’s Guide for a description of the blocking factor.

The arguments to CDFgetzVarBlockingFactor are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **bf**: The blocking factor. A value of zero (0) indicates that the default blocking factor will be used.

### 6.3.15.1. Example(s)

The following example returns the blocking factor for the zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long bf; /* The blocking factor. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("...");

status = CDFgetzVarBlockingFactor (id, varNum, &bf);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.16 CDFgetzVarCacheSize

CDFstatus CDFgetzVarCacheSize( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long *numBuffers); /* out -- Number of cache buffers. */

CDFgetzVarCacheSize returns the number of cache buffers being used for the specified zVariable in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User’s Guide for a description of caching scheme used by the CDF library.

The arguments to CDFgetzVarCacheSize are defined as follows:
id
The identifier of the current CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum
The zVariable number.

numBuffers
The number of cache buffers.

6.3.16.1. Example(s)

The following example returns the number of cache buffers for zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id;    /* CDF identifier. */
long varNum; /* zVariable number. */
long numBuffers; /* The number of cache buffers. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("….“);

status = CDFgetzVarCacheSize (id, varNum, &numBuffers);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.17 CDFgetzVarCompression

CDFstatus CDFgetzVarCompression(    /* out -- Completion status code. */
CDFid id,                             /* in -- CDF identifier. */
long varNum,                          /* in -- Variable number. */
long *cType,                          /* out -- Compression type. */
long cParms[],                       /* out -- Compression parameters. */
long *cPct);                          /* out -- Compression percentage. */

CDFgetzVarCompression returns the compression type/parameters and the compression percentage of the specified
zVariable in a CDF. Refer to Section 4.10 for a description of the CDF supported compression types/parameters. The
compression percentage is the result of the compressed size from all variable records divided by its original,
uncompressed variable size.

The arguments to CDFgetzVarCompression are defined as follows:

id
The identifier of the current CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum
The zVariable number.
### 6.3.17.1. Example(s)

The following example returns the compression information for zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long cType; /* The compression type. */
long cParms[CDF_MAX_PARMS]; /* The compression parameters. */
long cPct; /* The compression percentage. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("...");

status = CDFgetzVarCompression (id, varNum, &cType, cParms, &cPct);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.18 CDFgetzVarData

CDFstatus CDFgetzVarData(id, varNum, recNum, indices[], value);

CDFgetzVarData returns a data value from the specified indices, the location of the element, in the given record of the specified zVariable in a CDF.

The arguments to CDFgetzVarData are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
recNum The record number.
indices The dimension indices within the record.
value The data value.

6.3.18.1. Example(s)

The following example returns two data values, the first and the fifth element, in Record 0 from zVariable “MY_VAR”, a 2-dimensional (2 by 3) CDF_DOUBLE type variable, in a row-major CDF.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long recNum; /* The record number. */
long indices[2]; /* The dimension indices. */
double value1, value2; /* The data values. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("...");
recNum = 0L;
indices[0] = 0L;
indices[1] = 0L;
status = CDFgetzVarData (id, varNum, recNum, indices, &value1);
if (status != CDF_OK) UserStatusHandler (status);
indices[0] = 1L;
indices[1] = 1L;
status = CDFgetzVarData (id, varNum, recNum, indices, &value2);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.19 CDFgetzVarDataType

CDFgetzVarDataType( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long *dataType); /* out -- Data type. */

CDFgetzVarDataType returns the data type of the specified zVariable in a CDF. Refer to Section 4.5 for a description of the CDF data types.

The arguments to CDFgetzVarDataType are defined as follows:
id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum The zVariable number.

dataType The data type.

### 6.3.19.1. Example(s)

The following example returns the data type of zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long dataType; /* The data type. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("...");
status = CDFgetzVarDataType (id, varNum, &dataType);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.20 CDFgetzVarDimSizes

CDFstatus CDFgetzVarDimSizes( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    long varNum, /* in -- Variable number. */
    long dimSizes[]); /* out -- Dimension sizes. */

CDFgetzVarDimSizes returns the size of each dimension for the specified zVariable in a CDF. For 0-dimensional zVariables, this operation is not applicable.

The arguments to CDFgetzVarDimSizes are defined as follows:

- id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum The zVariable number
- dimSizes The dimension sizes. Each element of dimSizes receives the corresponding dimension size.
6.3.20.1. Example(s)

The following example returns the dimension sizes for zVariable “MY_VAR” in a CDF.

```
#include "cdf.h"

CDFid id;            /* CDF identifier. */
long dimSizes[CDF_MAX_DIMS]; /* The dimension sizes. */

status = CDFgetzVarDimSizes (id, CDFgetVarNum(id, "MY_VAR"), dimSizes);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.21   CDFgetzVarDimVariances

```c
CDFstatus CDFgetzVarDimVariances(
   /* out -- Completion status code. */
   CDFid id,
   /* in -- CDF identifier. */
   long varNum,
   /* in -- Variable number. */
   long dimVarys[]);
   /* out -- Dimension variances. */
```

CDFgetzVarDimVariances returns the dimension variances of the specified zVariable in a CDF. For 0-dimensional zVariable, this operation is not applicable. The dimension variances are described in section 4.9.

The arguments to CDFgetzVarDimVariances are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **dimVarys**: The dimension variances.

6.3.21.1. Example(s)

The following example returns the dimension variances of the 2-dimensional zVariable “MY_VAR” in a CDF.

```
#include "cdf.h"

CDFid id;            /* CDF identifier. */
long dimVarys[2];    /* The dimension variances. */
```
status = CDFgetzVarDimVariances (id, CDFgetVarNum (id, "MY_VAR"), dimVarys);
if (status != CDF_OK) UserStatusHandler (status);

6.3.22  CDFgetzVarMaxAllocRecNum

CDFstatus CDFgetzVarMaxAllocRecNum (CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long *maxRec); /* out -- Maximum allocated record number. */

CDFgetzVarMaxAllocRecNum returns the number of records allocated for the specified zVariable in a CDF.

The arguments to CDFgetzVarMaxAllocRecNum are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum The zVariable number.
maxRec The number of records allocated.

6.3.22.1. Example(s)

The following example returns the maximum allocated record number for the zVariable “MY_VAR” in a CDF.

#include "cdf.h"

CDFid id; /* CDF identifier. */
long maxRec; /* The maximum record number. */

status = CDFgetzVarMaxAllocRecNum (id, CDFgetVarNum (id, "MY_VAR"), &maxRec);
if (status != CDF_OK) UserStatusHandler (status);

6.3.23  CDFgetzVarMaxWrittenRecNum

CDFstatus CDFgetzVarMaxWrittenRecNum (CDFid id, /* in -- CDF identifier. */
CDFstatus CDFgetzVarMaxWrittenRecNum (CDFid id, /* in -- CDF identifier. */
CDFstatus CDFgetzVarMaxWrittenRecNum (CDFid id, /* in -- CDF identifier. */
CDFstatus CDFgetzVarMaxWrittenRecNum (CDFid id, /* in -- CDF identifier. */
long varNum,       /* in -- Variable number. */
long *maxRec);    /* out -- Maximum written record number. */

CDFgetzVarMaxWrittenRecNum returns the maximum record number written for the specified zVariable in a CDF.

The arguments to CDFgetzVarMaxWrittenRecNum are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **maxRec**: The maximum written record number.

### 6.3.23.1. Example(s)

The following example returns the maximum record number written for the zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long maxRec; /* The maximum record number. */

status = CDFgetzVarMaxWrittenRecNum (id, CDFgetVarNum (id, "MY_VAR"), &maxRec);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.24 CDFgetzVarName

CDFstatus CDFgetzVarName(       /* out -- Completion status code. */
CDFid id,                      /* in -- CDF identifier. */
long varNum,                   /* in -- Variable number. */
char *varName);                /* out -- Variable name. */

CDFgetzVarName returns the name of the specified zVariable, by its number, in a CDF.

The arguments to CDFgetzVarName are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **varName**: The name of the variable.
6.3.24.1. Example(s)

The following example returns the name of the zVariable whose variable number is 1.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
char varName[CDF_VAR_NAME_LEN256]; /* The name of the variable. */

varNum = 1L;
status = CDFgetzVarName (id, varNum, varName);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.25  CDFgetzVarNumDims

CDFstatus CDFgetzVarNumDims( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long *numDims); /* out -- Number of dimensions. */

CDFgetzVarNumDims returns the number of dimensions (dimensionality) for the specified zVariable in a CDF.

The arguments to CDFgetzVarNumDims are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number
- **numDims**: The number of dimensions.

6.3.25.1. Example(s)

The following example returns the number of dimensions for zVariable “MY_VAR” in a CDF.
CDFid id;  /* CDF identifier. */
long numDims;  /* The dimensionality of the variable. */

status = CDFgetzVarNumDims (id, CDFgetVarNum(id, “MY_VAR”), &numDims);
if (status != CDF_OK) UserStatusHandler (status);

6.3.26 CDFgetzVarNumElements

CDFstatus CDFgetzVarNumElements( /* out -- Completion status code. */
CDFid  id,  /* in -- CDF identifier. */
long  varNum,  /* in -- Variable number. */
long *numElems);  /* out -- Number of elements. */

CDFgetzVarNumElements returns the number of elements for each data value of the specified zVariable in a CDF. For character data type (CDF_CHAR and CDF_UCHAR), the number of elements is the number of characters in the string. For other data types, the number of elements will always be one (1).

The arguments to CDFgetzVarNumElements are defined as follows:

id  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum  The zVariable number.
numElems  The number of elements.

6.3.26.1. Example(s)

The following example returns the number of elements for the data type from zVariable “MY_VAR” in a CDF.

#include "cdf.h"

CDFid id;  /* CDF identifier. */
long numElems;  /* The number of elements. */

status = CDFgetzVarNumElements (id, CDFgetVarNum (id, “MY_VAR”), &numElems);
if (status != CDF_OK) UserStatusHandler (status);
6.3.27  CDFgetzVarNumRecsWritten

CDFstatus CDFgetzVarNumRecsWritten( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long *numRecs); /* out -- Number of written records. */

CDFgetzVarNumRecsWritten returns the number of records written for the specified zVariable in a CDF. This number may not correspond to the maximum record written if the zVariable has sparse records.

The arguments to CDFgetzVarNumRecsWritten are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum The zVariable number.

numRecs The number of written records.

6.3.27.1. Example(s)

The following example returns the number of written records from zVariable “MY_VAR” in a CDF.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
long numRecs; /* The number of written records. */

status = CDFgetzVarNumRecsWritten (id, CDFgetVarNum (id, "MY_VAR"), &numRecs);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.28  CDFgetzVarPadValue

CDFstatus CDFgetzVarPadValue( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
void *value); /* out -- Pad value. */

CDFgetzVarPadValue returns the pad value of the specified zVariable in a CDF. If a pad value has not been explicitly specified for the zVariable through CDFsetzVarPadValue or something similar from the Internal Interface function, the informational status code NO_PADVALUE_SPECIFIED will be returned and the default pad value for the variable’s data type will be placed in the pad value buffer provided.
The arguments to CDFgetzVarPadValue are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **varNum**: The zVariable number.

- **value**: The pad value.

### 6.3.28.1. Example(s)

The following example returns the pad value from zVariable “MY_VAR”, a CDF_INT4 type variable, in a CDF.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
int padValue;  /* The pad value. */

status = CDFgetzVarPadValue (id, CDFgetVarNum (id, "MY_VAR"), &padValue);
if (status != NO_PADVALUE_SPECIFIED) {

}
```
startRec  The zero-based starting record number.

stopRec  The zero-based stopping record number.

buffer  The buffer that holds the returned data.

6.3.29.1. Example(s)

The following example reads the 100 record data, from record number 10 to 109 for variable “MY_VAR” in a CDF. Assuming each record is a 1-dimensional, with 3 elements, of double type.

```c
#include "cdf.h"

CDFid id;    /* CDF identifier. */
long varNum; /* zVariable number. */
double buffer[100][3]; /* The buffer holding the data. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("….");

status = CDFgetzVarRangeRecordsByVarID (id, varNum, 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);
```

More general approach: for a variable of double type:

```c
#include "cdf.h"

CDFid id;    /* CDF identifier. */
long varNum; /* zVariable number. */
long numDims; /* Number of zVariable’s dimensions. */
long dimSizes[CDF_MAX_DIMS]; /* zVariable’s dimensionality. */
long numValues; /* Total number of values. */
double *buffer; /* The buffer holding the data. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("….");

status = CDFgetzVarNumDims (id, varNum, &numDims);
if (status != CDF_OK) ….
status = CDFgetzVarDimSizes (id, varNum, dimSizes);
if (status != CDF_OK) ….
numValues = 1;
for (i=1; i<numDims;++i) numValues *= dimSizes[i];
```
numvalue *= (109 - 10 + 1);
buffer = (double *) malloc((sizeof(double) * (size_t) numValues);
status = CDFgetzVarRangeRecordsByVarID (id, varNum, 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);
.
free (buffer);

6.3.30  CDFgetzVarRecordData

CDFstatus CDFgetzVarRecordData(/* out -- Completion status code. */
CDFid id,            /* in -- CDF identifier. */
long varNum,        /* in -- Variable number. */
long recNum,        /* in -- Record number. */
void *buffer);      /* out -- Record data. */

CDFgetzVarRecordData returns an entire record at a given record number for the specified zVariable in a CDF. The buffer should be large enough to hold the entire data values form the variable.

The arguments to CDFgetzVarRecordData are defined as follows:

id  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum  The zVariable number.
recNum  The record number.
buffer  The buffer holding the entire record data.

6.3.30.1. Example(s)

The following example will read two full records (record numbers 2 and 5) from zVariable “MY_VAR”, a 2-dimension (2 by 3), CDF_INT4 type variable, in a CDF. The variable’s dimension variances are all VARY.

```
#include "cdf.h"
.

CDFid id;          /* CDF identifier. */
long varNum;       /* zVariable number. */
int *buffer1;      /* The data holding buffer – dynamical allocation. */
int buffer2[2][3]; /* The data holding buffer – static allocation. */
long size;
.
varNum = CDFgetVarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“…..”);
status = CDFgetDataTypeInfo (CDF_INT4, &size);
buffer1 = (int *) malloc(2*3*(int)size);
status = CDFgetzVarRecordData (id, varNum, 2L, buffer1);
if (status != CDF_OK) UserStatusHandler (status);
status = CDFgetzVarRecordData (id, varNum, 5L, buffer2);
if (status != CDF_OK) UserStatusHandler (status);
.
.
free (buffer1);

6.3.31 CDFgetzVarRecVariance

CDFstatus CDFgetzVarRecVariance(  /* out -- Completion status code. */
  CDFid id,  /* in -- CDF identifier. */
  long varNum,  /* in -- Variable number. */
  long *recVary);  /* out -- Record variance. */

CDFgetzVarRecVariance returns the record variance of the specified zVariable in a CDF. The record variances are described in Section 4.9.

The arguments to CDFgetzVarRecVariance are defined as follows:

id  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum  The zVariable number.
recVary  The record variance.

6.3.31.1. Example(s)

The following example returns the record variance for the zVariable “MY_VAR” in a CDF.
.
.
#include "cdf.h"
.
.
CDFid id;  /* CDF identifier. */
long recVary;  /* The record variance. */
.

status = CDFgetzVarRecVariance (id, CDFgetVarNum (id, “MY_VAR”), &recVary);
if (status != CDF_OK) UserStatusHandler (status);
.
.
6.3.32 CDFgetzVarReservePercent

CDFstatus CDFgetzVarReservePercent(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long varNum,  /* in -- Variable number. */
long *percent);  /* out -- Reserve percentage. */

CDFgetzVarReservePercent returns the compression reserve percentage being used for the specified zVariable in a CDF. This operation only applies to compressed zVariables. Refer to the CDF User’s Guide for a description of the reserve scheme used by the CDF library.

The arguments to CDFgetzVarReservePercent are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **percent**: The reserve percentage.

6.3.32.1. Example(s)

The following example returns the compression reserve percentage from the compressed zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long percent; /* The compression reserve percentage. */

status = CDFgetzVarReservePercent (id, CDFgetVarNum (id, "MY_VAR"), &percent);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.33 CDFgetzVarSeqData

CDFstatus CDFgetzVarSeqData(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long varNum,  /* in -- Variable number. */
void *value);  /* out -- Data value. */

CDFgetzVarSeqData reads one value from the specified zVariable in a CDF at the current sequential value (position). After the read, the current sequential value is automatically incremented to the next value. An error is returned if the
current sequential value is past the last record of the zVariable. Use CDFsetzVarSeqPos function to set the current sequential value (position).

The arguments to CDFgetzVarSeqData are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number from which to read data.
- **value**: The buffer to store the value.

### 6.3.33.1. Example(s)

The following example will read the first two data values from the beginning of record number 2 (from a 2-dimensional zVariable whose data type is CDF_INT4) in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* The variable number from which to read data */
int value1, value2; /* The data value. */
long indices[2]; /* The indices in a record. */
long recNum; /* The record number. */

recNum = 2L;
indices[0] = 0L;
indices[1] = 0L;
status = CDFsetzVarSeqPos (id, varNum, recNum, indices);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFgetzVarSeqData (id, varNum, &value1);
if (status != CDF_OK) UserStatusHandler (status);
status = CDFgetzVarSeqData (id, varNum, &value2);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.34 CDFgetzVarSeqPos

CDFstatus CDFgetzVarSeqPos(
    /* out -- Completion status code. */
    CDFid id,
    /* in -- CDF identifier. */
    long varNum,
    /* in -- Variable number. */
    long *recNum,
    /* out -- Record number. */
    long indices[]);
    /* out -- Indices in a record. */
CDFgetzVarSeqPos returns the current sequential value (position) for sequential access for the specified zVariable in a CDF. Note that a current sequential value is maintained for each zVariable individually. Use CDFsetzVarSeqPos function to set the current sequential value.

The arguments to CDFgetzVarSeqPos are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **recNum**: The zVariable record number.
- **indices**: The dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional zVariable, this argument is ignored, but must be presented.

### 6.3.34.1. Example(s)

The following example returns the location for the current sequential value (position), the record number and indices within it, from a 2-dimensional zVariable named MY_VAR in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long recNum; /* The record number. */
long indices[2]; /* The indices. */

status = CDFgetzVarSeqPos (id, CDFgetVarNum(id, "MY_VAR"), &recNum, indices);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.35 CDFgetzVarsMaxWrittenRecNum

CDFgetzVarsMaxWrittenRecNum returns the maximum record number among all of the zVariables in a CDF. Note that this is not the number of written records but rather the maximum written record number (that is one less than the number of records). A value of negative one (-1) indicates that zVariables contain no records. The maximum record number for an individual zVariable may be acquired using the CDFgetzVarMaxWrittenRecNum function call.

Suppose there are three zVariables in a CDF: Var1, Var2, and Var3. If Var1 contains 15 records, Var2 contains 10 records, and Var3 contains 95 records, then the value returned from CDFgetzVarsMaxWrittenRecNum would be 95.

The arguments to CDFgetzVarsMaxWrittenRecNum are defined as follows:
id

The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

recNum

The maximum written record number.

6.3.35.1. Example(s)

The following example returns the maximum record number for all of the zVariables in a CDF.

```c
#include "cdf.h"

CDFid id;   /* CDF identifier. */
long recNum; /* The maximum record number. */

status = CDFgetzVarsMaxWrittenRecNum (id, &recNum);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.36  CDFgetzVarSparseRecords

CDFstatus CDFgetzVarSparseRecords(
    CDFid id,   /* in -- CDF identifier. */
    long varNum, /* in -- The variable number. */
    long *sRecordsType); /* out -- The sparse records type. */

CDFgetzVarSparseRecords returns the sparse records type of the zVariable in a CDF. Refer to Section 4.11.1 for the description of sparse records.

The arguments to CDFgetzVarSparseRecords are defined as follows:

- **id**
  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **varNum**
  The variable number.

- **sRecordsType**
  The sparse records type.

6.3.36.1. Example(s)

The following example returns the sparse records type of the zVariable “MY_VAR” in a CDF.


```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long sRecordsType; /* The sparse records type. */

status = CDFgetzVarSparseRecords (id, CDFgetVarNum(id, "MY_VAR"), &sRecordsType);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.37 CDFgetzVarsRecordDatabyNumbers

CDFstatus CDFgetzVarsRecordDatabyNumbers( /* out -- Completion status code. */
  CDFid id, /* in -- CDF identifier. */
  long numVars, /* in -- Number of zVariables. */
  long varNums[], /* in -- zVariables’ numbers. */
  long varRecNum, /* in -- Number of record. */
  void *buffer; /* out -- Buffer for holding data. */
)

CDFgetzVarsRecordDatabyNumbers reads an entire record of the specified record number from the specified zVariable numbers in a CDF. This function provides an easier and higher level interface to acquire data for a group of variables, instead of doing it one variable at a time if calling the lower-level function. The retrieved record data from the zVariable group is added to the buffer. The specified variables are identified by their variable numbers. Use the CDFgetzVarsRecordData function to perform the same operation by providing the variable names, instead of the variable numbers.

The arguments to CDFgetzVarsRecordDatabyNumbers are defined as follows:

- **id** The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDFopenCDF or a similar CDF creation or opening functionality from the Internal Interface.
- **numVars** The number of the zVariables in the group involved this read operation.
- **varNums** The zVariables’ numbers from which to read data.
- **varRecNum** The record number at which to read data.
- **buffer** Buffer that holds the retrieved data for the given zVariables. It should be big enough to allow full physical record data from all variables to fill.

### 6.3.37.1. Example(s)

The following example will read an entire single record data for a group of zVariables: Time, Longitude, Delta and Name. The record to be read is the sixth record that is record number 5 (record number starts at 0). For Longitude, a 1-dimensional array of type short (size [3]) is given based on its dimension variance [VARY] and data type CDF_INT2. For Delta, it is 2-dimensional of type int (sizes [3,2]) for its dimension variances [VARY,VARY] and data type

---

129
CDF_INT4. For zVariable Time, a 2-dimensional array of type unsigned int (size [3,2]) is needed. It has dimension variances [VARY,VARY] and data type CDF_UINT4. For Name, a 2-dimensional array of type char (size [2,10]) is allocated for its [VARY] dimension variances and CDF_CHAR data type with the number of element 10.

```
#include "cdf.h"

CDFId       id; /* CDF identifier. */
CDFstatus   status; /* Returned status code. */
long        numVars = 4; /* Number of zVariables to read. */
long        varRecNum = 5; /* The record number to read data. */
char *zVar1 = "Longitude", *zVar2 = "Delta", *zVar3 = "Time", *zVar4 = "Name";
long        varNums[4];
void        *buffer, *bufferptr; /* Buffer for holding retrieved data. */
unsigned int time[3][2]; /* zVariable: Time; Datatype: UINT4. */
short       longitude[3]; /* zVariable: Longitude; Datatype: INT2. */
int         delta[3][2]; /* zVariable: Delta; Datatype: INT4. */
char        name[2][10]; /* zVariable: Name; Datatype: CHAR/10. */

varNums[0] = CDFgetVarNum(id, zVar1); /* Number of each zVariable. */
varNums[1] = CDFgetVarNum(id, zVar2);
varNums[2] = CDFgetVarNum(id, zVar3);
varNums[3] = CDFgetVarNum(id, zVar4);

buffer = (void *) malloc(sizeof(longitude) + sizeof(delta) + sizeof(time) + sizeof(name));

status = CDFgetzVarsRecordDatabyNumbers(id, numVars, varNums, varRecNum, buffer);
if (status != CDF_OK) UserStatusHandler (status);

  bufferptr = buffer;
  memcpy(time, bufferptr, sizeof(time));
  bufferptr += sizeof(time);
  memcpy(longitude, bufferptr, sizeof(longitude));
  bufferptr += sizeof(longitude);
  memcpy(delta, bufferptr, sizeof(delta));
  bufferptr += sizeof(delta);
  memcpy(name, bufferptr, sizeof(name));

free (buffer);
```
CDFhyperGetzVarData is used to read one or more values for the specified zVariable. It is important to know the variable majority of the CDF before using this function because the values placed into the data buffer will be in that majority. CDFinquireCDF can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The record number starts at 0, not 1. For example, if you want to read the first 5 records, the starting record number (recStart), the number of records to read (recCount), and the record interval (recInterval) should be 0, 5, and 1, respectively.

The arguments to CDFhyperGetzVarData are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number from which to read data. This number may be determined with a call to CDFgetVarNum.
- **recStart**: The record number at which to start reading.
- **recCount**: The number of records to read.
- **recInterval**: The reading interval between records (e.g., an interval of 2 means read every other record).
- **indices**: The dimension indices (within each record) at which to start reading. Each element of indices specifies the corresponding dimension index. For 0-dimensional zVariable, this argument is ignored (but must be present).
- **counts**: The number of values along each dimension to read. Each element of counts specifies the corresponding dimension count. For 0-dimensional zVariable, this argument is ignored (but must be present).
- **intervals**: For each dimension, the dimension interval between reading (e.g., an interval of 2 means read every other value). Each element of intervals specifies the corresponding dimension interval. For 0-dimensional zVariable, this argument is ignored (but must be present).
- **buffer**: The data holding buffer for the read values. 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. CDFinquirezVar can be used to determine the zVariable's data type and number of elements (of that data type) at each value.
6.3.38.1. Example(s)

The following example will read 3 records of data, starting at record number 13 (14th record), from a zVariable named Temperature. The variable is a 3-dimensional array with sizes [180,91,10] and the CDF’s variable majority is ROW_MAJOR. 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 CDFgetzVarData example except that it uses a single call to CDFhyperGetzVarData (rather than numerous calls to CDFgetzVarData).

```c
#include "cdf.h"

CDFid id;                 /* CDF identifier. */
CDFstatus status;         /* Returned status code. */
float tmp[3][180][91][10]; /* Temperature values. */
long varN;                /* zVariable number. */
long recStart = 13;       /* Start record number. */
long recCount = 3;        /* Number of records to read */
long recInterval = 1;     /* Record interval – read every record */
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 – read every value*/

varN = CDFgetVarNum(id, "Temperature");
if (varN < CDF_OK) UserStatusHandle(varN);
status = CDFhyperGetzVarData(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 COLUMN_MAJOR, the tmp array would have been declared float tmp[10][91][180][3] for proper indexing.

6.3.39 CDFhyperPutzVarData

CDFstatus CDFhyperPutzVarData(/* out -- Completion status code. */
CDFid id,                  /* in -- CDF identifier. */
long varNum,               /* in -- zVariable number. */
long recStart,             /* in -- Starting record number. */
long recCount,             /* in -- Number of records. */
long recInterval,          /* in -- Writing interval between records. */
long indices[],            /* in -- Dimension indices of starting value. */
long counts[],             /* in -- Number of values along each dimension. */
long intervals[],          /* in -- Writing intervals along each dimension. */
void *buffer);             /* in -- Buffer of values. */

CDFhyperPutzVarData is used to write one or more values from the data holding buffer to the specified zVariable. It is important to know the variable majority of the CDF before using this function because the values in the data buffer will be written using that majority. CDFinquireCDF can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.
The record number starts at 0, not 1. For example, if you want to write 2 records (10th and 11th record), the starting record number (recStart), the number of records to write (recCount), and the record interval (recInterval) should be 9, 2, and 1, respectively.

The arguments to CDFhyperPutzVarData are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number to which write data. This number may be determined with a call to CDFgetVarNum.
- **recStart**: The record number at which to start writing.
- **recCount**: The number of records to write.
- **recInterval**: The interval between records for writing (e.g., an interval of 2 means write every other record).
- **indices**: The indices (within each record) at which to start writing. Each element of indices specifies the corresponding dimension index. For 0-dimensional zVariable this argument is ignored (but must be present).
- **counts**: The number of values along each dimension to write. Each element of counts specifies the corresponding dimension count. For 0-dimensional zVariable this argument is ignored (but must be present).
- **intervals**: For each dimension, the interval between values for writing (e.g., an interval of 2 means write every other value). Each element of intervals specifies the corresponding dimension interval. For 0-dimensional zVariable this argument is ignored (but must be present).
- **buffer**: The data holding 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.

### 6.3.39.1. Example(s)

The following example writes 2 records to a zVariable named LATITUDE that is a 1-dimensional array with dimension sizes [181]. The dimension variances are [VARY], and the data type is CDF_INT2. This example is similar to the CDFputzVarData example except that it uses a single call to CDFhyperPutzVarData rather than numerous calls to CDFputzVarData.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
short lat; /* Latitude value. */
short i, lats[2][181]; /* Buffer of latitude values. */
long varN; /* zVariable number. */
long recStart = 0; /* Record number. */
long recCount = 2; /* Record counts. */
```
long recInterval = 1; /* Record interval. */
static long indices[] = {0}; /* Dimension indices. */
static long counts[] = {181}; /* Dimension counts. */
static long intervals[] = {1}; /* Dimension intervals. */

varN = CDFgetVarNum (id, "LATITUDE");
if (varN < CDF_OK) UserStatusHandler (varN); /* If less than zero (0), not a zVariable number but
rather a warning/error code. */

for (i= 0; i < 2; i++)
  for (lat = -90; lat <= 90; lat++)
    lats[i][90+lat] = lat;

status = CDFhyperPutzVarData (id, varN, recStart, recCount, recInterval, indices, counts, intervals, lats);
if (status != CDF_OK) UserStatusHandler (status);

6.3.40 CDFinquirezVar

CDFstatus CDFinquirezVar( /* out -- Completion status code. */
  CDFid id, /* in -- CDF identifier */
  long varNum, /* in -- zVariable number */
  char varName, /* out -- zVariable name */
  long *dataType, /* out -- Data type */
  long *numElements, /* out -- Number of elements (of the data type) */
  long *numDims, /* out -- Number of dimensions */
  long *dimSizes[], /* out -- Dimension sizes */
  long *recVariance, /* out -- Record variance */
  long *dimVariances[]); /* out -- Dimension variances */

CDFinquirezVar is used to inquire about the specified zVariable. This function would normally be used before reading
zVariable values (with CDFgetzVarData or CDFhyperGetzVarData) to determine the data type and number of elements
of that data type.

The arguments to CDFinquirezVar are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to
    CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum The number of the zVariable to inquire. This number may be determined with a call to
    CDFgetVarNum (see Section 6.3.10).

varName The zVariable's name. This character string must be large enough to hold
    CDF_VAR_NAME_LEN256 + 1 characters (including the NUL terminator).

dataType The data type of the zVariable. The data types are defined in Section 4.5.

numElements The number of elements of the data type at each zVariable value. For character 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 data types.

numDims The number of dimensions.

dimSizes The dimension sizes. It is a 1-dimensional array, containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional zVariables this argument is ignored (but must be present).

dimVariances The dimension variances. Each element of dimVariances receives the corresponding dimension variance. The dimension variances are described in Section 4.9. For 0-dimensional zVariables this argument is ignored (but a placeholder is necessary).

6.3.40.1. Example(s)

The following example returns information about an zVariable named HEAT_FLUX in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
char varName[CDF_VAR_NAME_LEN256+1]; /* zVariable name, +1 for NUL terminator. */
long dataType; /* Data type of the zVariable. */
long numElems; /* Number of elements (of data type). */
long recVary; /* Record variance. */
long numDims; /* Number of dimensions. */
long dimSizes[CDF_MAX_DIMS]; /* Dimension sizes (allocate to allow the maximum number of dimensions). */
long dimVariances[CDF_MAX_DIMS]; /* Dimension variances (allocate to allow the maximum number of dimensions). */

status = CDFinquirezVar(id, CDFgetVarNum(id,"HEAT_FLUX"), varName, &dataType, &numElems, &numDims, dimSizes, &recVary, dimVariances);
if (status != CDF_OK) UserStatusHandler (status);

varNum The zVariable number.

6.3.41 CDFinsertrVarRecordsByVarID

CDFstatus CDFinsertrVarRecordsByVarID( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier */
    long varNum, /* in -- rVariable number. */
    ......
CDFinsertVarRecordsByVarID inserts a number of records for the specified rVariable in a CDF. This function will move down the existing records in range by the number of inserted records, as passed numRecs. The data buffer should be big enough to hold all data values in the records. Segmentation could occur if the buffer does not have enough data. The function is only applicable to rVariables defined as non-sparse records.

The arguments to CDFinsertVarRecordsByVarID are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The rVariable number.
- **startRec**: The starting record to insert.
- **numRecs**: The number of records to insert.
- **buffer**: The buffer that holds the full data values for the inserted records.

### 6.3.41.1. Example(s)

The following example shows how 10 records, from (zero-based) record number 5, are inserted for an rVariable “Test”, a scalar of CDF_INT4 type, in a CDF.

```c
#include "cdf.h"

CDFid  id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long  varNum; /* rVariable number. */
long  startRec; /* Starting record to insert. */
long  numRecs; /* Number of records to insert. */
int  buffer[10]; /* Data buffer for inserted records. */

varNum = CDFvarNum (id, "Test");
startRec = 5L;
numRecs = 10L;

.. fill buffer

status = CDFinsertVarRecordsByVarID (id, varNum, startRec, numRecs, buffer);
if (status != CDF_OK) UserStatusHandler (status);
```
CDFstatus CDFinsertVarRecordsByVarName( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier */
    char *varName, /* in -- r/zVariable name. */
    long startRec, /* in -- Starting record number to insert. */
    long numRecs, /* in -- Number of records to insert. */
    void *buffer); /* in -- Data holding buffer. */

CDFinsertVarRecordsByVarName inserts a number of records for the specified r/zVariable in a CDF. As a variable name is unique in a CDF, this function can be used for both rVariables and zVariables. This function will move down the existing records in range by the number of inserted records, as passed numRecs. The data buffer should be big enough to hold all data values in the records. Segmentation could occur if the buffer does not have enough data. The function is only applicable to variables defined as non-spared records.

The arguments to CDFinsertVarRecordsByVarName are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varName**: The r/zVariable name.
- **startRec**: The starting record to insert.
- **numRecs**: The number of records to insert.
- **buffer**: The buffer that holds the full data values for the inserted records.

### 6.3.42.1. Example(s)

The following example shows how 10 records, from (zero-based) record number 5, are inserted for a zVariable “Test”, a scalar of CDF_INT4 type, in a CDF.

```c
#include "cdf.h"

CDFid id;
CDFstatus status; /* CDF identifier. */
/* Returned status code. */
long startRec; /* Starting record to insert. */
long numRecs; /* Number of records to insert. */
int buffer[10]; /* Data buffer for inserted records. */

startRec = 5L;
numRecs = 10L;
... fill buffer
...
status = CDFinsertVarRecordsByVarName (id, “Test”, startRec, numRecs, buffer);
if (status != CDF_OK) UserStatusHandler (status);

6.3.43  CDFinsertzVarRecordsByVarID

CDFstatus CDFinsertzVarRecordsByVarID(
    CDFid id, /* in -- CDF identifier */
    long varNum, /* in -- zVariable number. */
    long startRec, /* in -- Starting record number to insert. */
    long numRecs, /* in -- Number of records to insert. */
    void *buffer); /* in -- Data holding buffer. */

CDFinsertzVarRecordsByVarID inserts a number of records for the specified zVariable in a CDF. This function will move down the existing records in range by the number of inserted records, as passed numRecs. The data buffer should be big enough to hold all data values in the records. Segmentation could occur if the buffer does not have enough data. The function is only applicable to zVariables defined as non-sparse records.

The arguments to CDFinsertzVarRecordsByVarID are defined as follows:

- **id**
  - The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **varNum**
  - The zVariable number.

- **startRec**
  - The starting record to insert

- **numRecs**
  - The number of records to insert.

- **buffer**
  - The buffer that holds the full data values for the inserted records.

6.3.43.1. Example(s)

The following example shows how 10 records, from (zero-based) record number 5, are inserted for an zVariable "Test", a scalar of CDF_INT4 type, in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long varNum; /* zVariable number. */
long startRec; /* Starting record to insert. */
long numRecs; /* Number of records to insert. */
int buffer[10]; /* Data buffer for inserted records. */
```
varNum = CDFvarNum (id, "Test");
startRec = 5L;
numRecs = 10L;

....
.... fill buffer
....
status = CDFinsertzVarRecordsByVarID (id, varNum, startRec, numRecs, buffer);
if (status != CDF_OK) UserStatusHandler (status);

6.3.44 CDFputVarAllRecordsByVarName

CDFstatus CDFputVarAllRecordsByVarName(  /**< out -- Completion status code. */
CDFid id, /**< in -- CDF identifier. */
char *varName, /**< in -- Variable name. */
long numRecs, /**< in -- The total number of records to write. */
void *buffer); /**< in -- Buffer for the written record data. */

CDFputVarAllRecordsByVarName writes/updates the whole data records from the specified variable in a CDF. This function provides an easier way of writing data from a variable. Since a variable name is unique in a CDF, this name can be either a zVariable or rVariable. The variable shall be created before this function can be called.

The arguments to CDFputVarAllRecordsByVarName are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varName: The Variable name.
- numRecs: The total number of records to write.
- buffer: The buffer that holds the written data.

6.3.44.1. Example(s)

The following example writes out a total of 100 records, for zVariable “MY_VAR” in a CDF. Assuming each record is a 1-dimensional, with 3 elements, of double type.

```c
#include "cdf.h"

CDFid id; /**< CDF identifier. */
double buffer[100][3]; /**< The buffer holding the data. */
```

30 If the variable already has more records than the numRecs in this function call, those records out of the range will stay after the call. If you want to remove those records, you can delete all records before calling this function.
6.3.45  CDFputVarRangeRecordsByVarName

CDFstatus CDFputVarRangeRecordsByVarName(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
char *varName,  /* in -- Variable name. */
long startRec,  /* in -- The starting record to write. */
long stopRec,  /* in -- The stopping record to write. */
void *buffer);  /* in -- Buffer for the written record data. */

CDFputVarRangeRecordsByVarName writes the whole data records from the specified variable in a CDF. This function provides an easier way of writing data from a variable. Since the variable name is unique in a CDF, this name can be either a zVariable or rVariable. The variable shall be created before this function can be called.

The arguments to CDFputVarRangeRecordsByVarName are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varName**: The variable name.
- **startRec**: The starting record number to write.
- **stopRec**: The stopping record number to write.
- **buffer**: The buffer that holds the written data.

6.3.45.1. Example(s)

The following example writes out a range of record data, from record 10 to 109, for zVariable “MY_VAR” in a CDF. Assuming each record is a 1-dimensional, with 3 elements, of double type.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
double buffer[100][3];  /* The buffer holding the data. */

... fill the buffer
```
... status = CDFputVarRangeRecordsByVarName (id, "MY_VAR", 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);

6.3.46 CDFputzVarAllRecordsByVarID

CDFstatus CDFputzVarAllRecordsByVarID( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- zVariable number. */
long numRecs, /* in -- Number of records in total to write. */
void *buffer); /* in -- Buffer for the written record data. */

CDFputzVarAllRecordsByVarID writes/updats\(^3\) the whole records from the specified zVariable in a CDF. This function provides an easier way of writing all data from a variable. Make sure that the buffer has the enough data to cover the records to be written. The zVariable shall be created before this function can be called.

The arguments to CDFputzVarAllRecordsByVarID are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum: The zVariable number.
- numRecs: The total number of records to write.
- buffer: The buffer that holds the written data.

6.3.46.1. Example(s)

The following example writes out the whole record data for zVariable “MY_VAR” in a CDF.

Assuming that the variable has 100 records, each record being a 1-dimensional, with 3 elements, of double type.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
double buffer[100][3]; /* The buffer holding the data. */

varNum = CDFgetVarNum (id, "MY_VAR");
```

\(^{3}\) If the variable already has more records than the numRecs in this function call, those records out of the range will stay after the call. If you want to remove those records, you can delete all records before calling this function.
if (varNum < CDF_OK) Quit ("....");
.
... fill the buffer
...
status = CDFputzVarAllRecordsByVarID (id, varNum, 100L, buffer);
if (status != CDF_OK) UserStatusHandler (status);
.
.
6.3.47 CDFputzVarData

CDFstatus CDFputzVarData(
CDFid id, /* out -- Completion status code. */
long varNum, /* in -- CDF identifier. */
long recNum, /* in -- Variable number. */
long indices[], /* in -- Record number. */
void *value); /* in -- Dimension indices. */

CDFputzVarData writes a single data value to the specified index, the location of the element, in the given record of the specified zVariable in a CDF.

The arguments to CDFputzVarData are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum The zVariable number.

recNum The record number.

indices The dimension indices within the record.

value The data value.

6.3.47.1. Example(s)

The following example will write two data values, the first and the fifth element, in Record 0 from zVariable “MY_VAR”, a 2-dimensional (2 by 3), CDF_DOUBLE type variable, in a row-major CDF.

.
.
#include "cdf.h"
.
.
CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long recNum; /* The record number. */
long indices[2]; /* The dimension indices. */
double value1, value2; /* The data values. */
.
.
142
varNum = CDFgetVarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“....”);
recNum = 0L;
indices[0] = 0L;
indices[1] = 0L;
value1 = 10.1;
status = CDFputzVarData (id, varNum, recNum, indices, &value1);
if (status != CDF_OK) UserStatusHandler (status);
indices[0] = 1L;
indices[1] = 1L;
value2 = 20.2;
status = CDFputzVarData (id, varNum, recNum, indices, &value2);
if (status != CDF_OK) UserStatusHandler (status);

6.3.48  CDFputzVarRangeRecordsByVarID

CDFstatus CDFputzVarRangeRecordsByVarID(
        CDFid id,
        long varNum,
        long startRec,
        long stopRec,
        void *buffer);

CDFputzVarRangeRecordsByVarID writes/updates a range of records from the specified zVariable in a CDF. This
function provides an easier way of writing data from a variable. The zVariable shall be created before this function can
be called.

The arguments to CDFputzVarRangeRecordsByVarID are defined as follows:

id
The identifier of the current CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum
The zVariable number.

startRec
The starting record number to write.

stopRec
The stopping record number to write.

buffer
The buffer that holds the written data.

6.3.48.1. Example(s)

The following example writes out a range of record data, from record 10 to 109, for zVariable “MY_VAR” in a CDF.
Assuming each record is a 1-dimensional, with 3 elements, of double type.
#include "cdf.h"
.

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
double buffer[100][3]; /* The buffer holding the data. */
.

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("....");
.
... fill the buffer
...
status = CDFputzVarRangeRecordsByVarID (id, varNum, 10L, 109L, buffer);
if (status != CDF_OK) UserStatusHandler (status);
.
.

6.3.49 CDFputzVarRecordData

CDFstatus CDFputzVarRecordData(
    /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    long varNum, /* in -- Variable number. */
    long recNum, /* in -- Record number. */
    void *buffer); /* in -- Record data. */

CDFputzVarRecordData writes an entire record at a given record number for the specified zVariable in a CDF. The buffer should hold the entire data values for the variable. The data values in the buffer should be in the order that corresponds to the variable majority defined for the CDF.

The arguments to CDFputzVarRecordData are defined as follows:

- **id** The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum** The zVariable number.
- **recNum** The record number.
- **buffer** The buffer holding the entire record values.

6.3.49.1. Example(s)

The following example will write two full records (numbered 2 and 5) from zVariable “MY_VAR”, a 2-dimension (2 by 3), CDF_INT4 type variable, in a CDF. The variable’s dimension variances are all VARY.

...
CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
int *buffer1; /* The data holding buffer – dynamical allocation. */
int buffer2[2][3]; /* The data holding buffer – static allocation. */
long size;
int i,j;

varNum = CDFgetVarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“….”);
status = CDFgetDataTypeSize (CDF_INT4, &size);
buffer1 = (int *) malloc(2*3*(int)size);
for (i=0; i<6; i++) *((int *) buffer1)+i) = i;
status = CDFputzVarRecordData (id, varNum, 2L, buffer1);
if (status != CDF_OK) UserStatusHandler (status);
for (i=0; i<2; I++)
  for (j=0; j<3; j++)
    buffer2[i][j] = i*j;
status = CDFputzVarRecordData (id, varNum, 5L, buffer2);
if (status != CDF_OK) UserStatusHandler (status);

free (buffer1);

6.3.50 CDFputzVarSeqData

CDFstatus CDFputzVarSeqData( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
void *value); /* in -- Data value. */

CDFputzVarSeqData writes one value to the specified zVariable in a CDF at the current sequential value (position) for that variable. After the write, the current sequential value is automatically incremented to the next value. Use CDFsetzVarSeqPos function to set the current sequential value (position).

The arguments to CDFputzVarSeqData are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum The zVariable number.
value The buffer holding the data value.

6.3.50.1. Example(s)

The following example will write two data values starting at record number 2 from a 2-dimensional zVariable whose data type is CDF_INT4.
```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* The variable number. */
int value1, value2; /* The data value. */
long indices[2]; /* The indices in a record. */
long recNum; /* The record number. */

recNum = 2L;
indices[0] = 0L;
indices[1] = 0L;
status = CDFsetzVarSeqPos (id, varNum, recNum, indices);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFputzVarSeqData (id, varNum, &value1);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFputzVarSeqData (id, varNum, &value2);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.51 CDFputzVarsRecordDatabyNumbers

CDFstatus CDFputzVarsRecordDatabyNumbers( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    long numVars, /* in -- Number of zVariables. */
    long varNums[], /* in -- zVariables’s numbers. */
    long varRecNum, /* in -- Record number. */
    void *buffer; /* in -- Buffer for input data. */
)

CDFputzVarsRecordDatabyNumbers is used to write a whole record data at a specific record number for a group of zVariables in a CDF. It expects that the data buffer matches up to the total full physical record size of all requested zVariables. Passed record data is filled into its respective zVariable. This function provides an easier and higher level interface to write data for a group of variables, instead of doing it one variable at a time if calling the lower-level function. The specified variables are identified by their variable numbers. Use CDFputzVarsRecordData function to perform the similar operation by providing the variable names, instead of the numbers.

The arguments to CDFputzVarsRecordDatabyNumbers are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDFopenCDF or a similar CDF creation or opening functionality from the Internal Interface.
- **numVars**: The number of the zVariables in the group involved this write operation.
- **varNums**: The zVariables’s numbers in the group involved this write operation.
- **varRecNum**: The record number at which to write the whole record data for the group of zVariables.
buffer  A buffer that holds the output data for the given zVariables.

6.3.51.1. Example(s)

The following example will write an entire single record data for a group of zVariables. The CDF’s zVariables are 2-dimensional with sizes [2,2]. The zVariables involved in the write are Time, Longitude, Latitude and Temperature. The record to be written is 4. Since the dimension variances for Time are [NONVARY,NONVARY], a scalar variable of type int is allocated for its data type CDF_INT4. For Longitude, a 1-dimensional array of type float (size [2]) is allocated as its dimension variances are [VARY,NONVARY] with data type CDF_REAL4. A similar 1-dimensional array is provided for Latitude for its [NONVARY,VARY] dimension variances and CDF_REAL4 data type. For Temperature, since its [VARY,VARY] dimension variances and CDF_REAL4 data type, a 2-dimensional array of type float is provided. For NAME, a 2-dimensional array of type char (size [2,10]) is allocated due to its [VARY, NONVARY] dimension variances and CDF_CHAR data type with the number of element 10.

```c
#include "cdf.h"

/* Dim/Rec Variances: T/TF. */
CDFid         id;   /* CDF identifier. */
CDFstatus   status; /* Returned status code. */
long             numVars = 5; /* Number of zVariables to write. */
long             varRecNum = 4; /* The record number to write data. */
char             *zVar1 = "Time", /* Names of the zVariables to write. */
char             *zVar2 = "Longitude",
char             *zVar3 = "Latitude",
char             *zVar4 = "Temperature",
char             *zVar5 = "NAME";
long varNums[5];
void             *buffer;   /* Buffer for holding the output data */
void             *bufferptr; /* Buffer place keeper */
int              time = {123}; /* zVariable: Time; Datatype: INT4. */
float           longitude[2] = /* zVariable: Longitude; Datatype: REAL4. */
{11.1, 22.2};     /* Dim/Rec Variances: T/FF. */
float           latitude[2] = /* zVariable: Latitude; Datatype: REAL4. */
{-11.1, -22.2};   /* Dim/Rec Variances: T/FT. */
float           temperature[2][2] = /* zVariable: Temperature; Datatype: REAL4. */
{100.0, 200.0, 300.0, 400.0}; /* Dim/Rec Variances: T/TT. */
char             name[2][10] = /* zVariable: NAME; Datatype: CHAR/10. */
{1', 3', 5', 7', 9', 2', 4', 6', 8', 0',
'z', 'Z', 'y', 'Y', 'x', 'X', 'w', 'W', 'v', 'V'};
varNums[0] = CDFgetVarNum(id, zVar1); /* Number of each zVariable. */
varNums[1] = CDFgetVarNum(id, zVar2);
varNums[2] = CDFgetVarNum(id, zVar3);
varNums[3] = CDFgetVarNum(id, zVar4);
varNums[4] = CDFgetVarNum(id, zVar5);

buffer = (void *) malloc(sizeof(time) + sizeof(longitude) + sizeof(latitude) + sizeof(temperature) + sizeof(name));
bufferptr = buffer;
memcpy(bufferptr, (void *) time, sizeof(time));
```
bufferptr += sizeof(time);
memcpy(bufferptr, (void *) longitude, sizeof(longitude));
bufferptr += sizeof(longitude);
memcpy(bufferptr, (void *) latitude, sizeof(latitude));
bufferptr += sizeof(latitude);
memcpy(bufferptr, (void *) temperature, sizeof(temperature));
bufferptr += sizeof(temperature);
memcpy(bufferptr, (void *) name, sizeof(name));

status = CDFputzVarsRecordDatabyNumbers(id, numVars, varNums, varRecNum, buffer);
if (status != CDF_OK) UserStatusHandler (status);
...
...
free (buffer);

6.3.52  CDFrenamezVar

CDFstatus CDFrenamezVar(id, varNum, varName);

CDFrenamezVar is used to rename an existing zVariable. A variable (rVariable or zVariable) with the same name
must not already exist in the CDF.

The arguments to CDFrenamezVar are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate
  (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The number of the zVariable to rename. This number may be determined with a call to
  CDFgetVarNum.
- **varName**: The new zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters
  (excluding the NUL terminator). Variable names are case-sensitive.

6.3.52.1. Example(s)

In the following example the zVariable named TEMPERATURE is renamed to TMP (if it exists). Note that if
CDFgetVarNum returns a value less than zero (0) then that value is not an zVariable number but rather a warning/error
code.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long varNum;  /* zVariable number. */
```
```c
varNum = CDFgetVarNum (id, "TEMPERATURE");
if (varNum < CDF_OK) {
    if (varNum != NO_SUCH_VAR) UserStatusHandler (varNum);
} else {
    status = CDFrenamezVar (id, varNum, "TMP");
    if (status != CDF_OK) UserStatusHandler (status);
}

6.3.53  CDFsetzVarAllocBlockRecords

CDFstatus CDFsetzVarAllocBlockRecords( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    long varNum, /* in -- Variable number. */
    long firstRec, /* in -- First record number. */
    long lastRec); /* in -- Last record number. */

CDFsetzVarAllocBlockRecords specifies a range of records to be allocated (not written) for the specified zVariable in a CDF. This operation is only applicable to uncompressed zVariable in single-file CDFs. Refer to the CDF User’s Guide for the descriptions of allocating variable records.

The arguments to CDFsetzVarAllocBlockRecords are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum: The zVariable number.
- firstRec: The first record number to allocate.
- lastRec: The last record number to allocate.

6.3.53.1. Example(s)

The following example allocates 10 records, from record numbered 10 to 19, for zVariable “MY_VAR” in a CDF.

```
status = CDFsetzVarAllocBlockRecords (id, CDFgetVarNum(id, "MY_VAR"), firstRec, lastRec);
if (status != CDF_OK) UserStatusHandler (status);

6.3.54 CDFsetzVarAllocRecords

CDFstatus CDFsetzVarAllocRecords(
    CDFid id,                      /* in -- CDF identifier. */
    long varNum,                    /* in -- Variable number. */
    long numRecs);                  /* in -- Number of records. */

CDFsetzVarAllocRecords specifies a number of records to be allocated (not written) for the specified zVariable in a CDF. The records are allocated beginning at record number zero (0). This operation is only applicable to uncompressed zVariable in single-file CDFs. Refer to the CDF User’s Guide for the descriptions of allocating variable records.

The arguments to CDFsetzVarAllocRecords are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum The zVariable number.
numRecs The number of records to allocate.

6.3.54.1. Example(s)

The following example allocates 100 records, from record numbered 0 to 99, for zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id;              /* CDF identifier. */
long numRecs;          /* The number of records. */

numRecs = 100L;
status = CDFsetzVarAllocRecords (id, CDFgetVarNum(id, "MY_VAR"), numRecs);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.55 CDFsetzVarBlockingFactor
CDFstatus CDFsetzVarBlockingFactor(  /* out -- Completion status code. */
CxDFid id,  /* in -- CDF identifier. */
long varNum,  /* in -- Variable number. */
long bf);  /* in -- Blocking factor. */

CDFsetzVarBlockingFactor specifies the blocking factor (number of records allocated) for the specified zVariable in a CDF. Refer to the CDF User’s Guide for a description of the blocking factor.

The arguments to CDFsetzVarBlockingFactor are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum: The zVariable number.
- bf: The blocking factor. A value of zero (0) indicates that the default blocking factor is being used.

6.3.55.1. Example(s)

The following example sets the blocking factor to 100 records for zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CxDFid id; /* CDF identifier. */
long bf; /* The blocking factor. */

bf = 100L;
status = CDFsetzVarBlockingFactor (id, CDFgetVarNum(id, "MY_VAR"), bf);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.56 CDFsetzVarCacheSize

CDFstatus CDFsetzVarCacheSize(  /* out -- Completion status code. */
CxDFid id,  /* in -- CDF identifier. */
long varNum,  /* in -- Variable number. */
long numBuffers);  /* in -- Number of cache buffers. */

CDFsetzVarCacheSize specifies the number of cache buffers being for the zVariable in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User’s Guide for description about caching scheme used by the CDF library.

The arguments to CDFsetzVarCacheSize are defined as follows:
id               The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum           The zVariable number.

numBuffers       The number of cache buffers.

6.3.56.1. Example(s)

The following example sets the number of cache buffers to 10 for zVariable “MY_VAR” in a CDF.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
long numBuffers; /* The number of cache buffers. */

numBuffers = 10L;
status = CDFsetzVarCacheSize (id, CDFgetVarNum(id, "MY_VAR"), numBuffers);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.57 CDFsetzVarCompression

CDFstatus CDFsetzVarCompression( /* out -- Completion status code. */
  CDFid id, /* in -- CDF identifier. */
  long varNum, /* in -- Variable number. */
  long cType, /* in -- Compression type. */
  long cParms[]); /* in -- Compression parameters. */

CDFsetzVarCompression specifies the compression type/parameters for the specified zVariable in a CDF. Refer to Section 4.10 for a description of the CDF supported compression types/parameters.

The arguments to CDFsetzVarCompression are defined as follows:

- **id**               The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**           The zVariable number.
- **cType**            The compression type.
- **cParms**           The compression parameters.
6.3.57.1. Example(s)

The following example sets the compression to GZIP.9 for zVariable “MY_VAR” in a CDF.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
long cType; /* The compression type. */
long cParms[CDF_MAX_PARMS]; /* The compression parameters. */


cType = GZIP_COMPRESSION;
cParms[0] = 9L;
status = CDFsetzVarCompression (id, CDFgetVarNum (id, "MY_VAR"), cType, cParms);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.58  CDFsetzVarDataSpec

CDFstatus CDFsetzVarDataSpec( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long dataType) /* in -- Data type. */

CDFsetzVarDataSpec respecifies the data type of the specified zVariable in a CDF. The variable’s data type cannot be changed if the new data type is not equivalent to the old data type and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent. Refer to the CDF User’s Guide for equivalent data types.

The arguments to CDFsetzVarDataSpec are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum: The zVariable number.
- dataType: The new data type.

6.3.58.1. Example(s)

The following example respecifies the data type to CDF_INT2 (from its original CDF_UINT2) for zVariable “MY_VAR” in a CDF.
```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long dataType; /* The data type. */

dataType = CDF_INT2;
status = CDFsetzVarDataSpec (id, CDFgetVarNum (id, "MY_VAR"), dataType);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.59 CDFsetzVarDimVariances

```c
CDFstatus CDFsetzVarDimVariances(
    /* out -- Completion status code. */
    CDFid id,
    /* in -- CDF identifier. */
    long varNum,
    /* in -- Variable number. */
    long dimVarys[]);
    /* in -- Dimension variances. */
```

CDFsetzVarDimVariances respecifies the dimension variances of the specified zVariable in a CDF. For 0-dimensional zVariable, this operation is not applicable. The dimension variances are described in Section 4.9.

The arguments to CDFsetzVarDimVariances are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **dimVarys**: The dimension variances.

### 6.3.59.1. Example(s)

The following example resets the dimension variances to true (VARY) and false (NOVARY) for zVariable "MY_VAR", a 2-dimensional variable, in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long dimVarys[2]; /* The dimension variances. */

varNum = CDFgetVarNum (id, "MY_VAR");
if (varNum < CDF_OK) Quit ("...");
```
dimVarys[0] = VARY;
dimVarys[1] = NOVARY;
status = CDFsetzVarDimVariances (id, varNum, dimVarys);
if (status != CDF_OK) UserStatusHandler (status);

6.3.60 CDFsetzVarInitialRecs

CDFstatus CDFsetzVarInitialRecs( /* out */ Completion status code. */
    CDFid id, /* in */ CDF identifier. */
    long varNum, /* in */ Variable number. */
    long numRecs); /* in */ Number of records. */

CDFsetzVarInitialRecs specifies a number of records to initially write to the specified zVariable in a CDF. The records are written beginning at record number 0 (zero). 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 the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records.

The arguments to CDFsetzVarInitialRecs are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum The zVariable number.

numRecs The initially written records.

6.3.60.1. Example(s)

The following example writes the initial 100 records to zVariable “MY_VAR” in a CDF.

#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* zVariable number. */
long numRecs /* The number of records. */

varNum = CDFgetVarNum (id, “MY_VAR”);
if (varNum < CDF_OK) Quit (“….”);
numRecs = 100L;
status = CDFsetzVarInitialRecs (id, varNum, numRecs);
if (status != CDF_OK) UserStatusHandler (status);

155
6.3.61  CDFsetzVarPadValue

CDFstatus CDFsetzVarPadValue( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
void *value); /* in -- Pad value. */

CDFsetzVarPadValue specifies the pad value for the specified zVariable in a 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). The Concepts chapter in the CDF User's Guide describes variable pad values.

The arguments to CDFsetzVarPadValue are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **value**: The pad value.

6.3.61.1. Example(s)

The following example sets the pad value to –9999 for zVariable “MY_VAR”, a CDF_INT4 type variable, in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
int padValue; /* The pad value. */

padValue = -9999L;
status = CDFsetzVarPadValue (id, CDFgetVarNum (id, "MY_VAR"), &padValue);
if (status != CDF_OK) UserStatusHandler (status);
```

6.3.62  CDFsetzVarRecVariance

CDFstatus CDFsetzVarRecVariance( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long varNum, /* in -- Variable number. */
long recVary); /* in -- Record variance. */
CDFsetzVarRecVariance specifies the record variance of the specified zVariable in a CDF. The record variances are described in Section 4.9.

The arguments to CDFsetzVarRecVariance are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
- **recVary**: The record variance.

### 6.3.62.1. Example(s)

The following example sets the record variance to VARY (from NOVARY) for zVariable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long recVary; /* The record variance. */

recVary = VARY;
status = CDFsetzVarRecVariance (id, CDFgetVarNum (id, "MY_VAR"), recVary);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.3.63  **CDFsetzVarReservePercent**

CDFstatus CDFsetzVarReservePercent(  /* out -- Completion status code. */
    CDFid id,  /* in -- CDF identifier. */
    long varNum,  /* in -- Variable number. */
    long percent);  /* in -- Reserve percentage. */

CDFsetzVarReservePercent specifies the compression reserve percentage being used for the specified zVariable in a CDF. This operation only applies to compressed zVariables. Refer to the CDF User’s Guide for a description of the reserve scheme used by the CDF library.

The arguments to CDFsetzVarReservePercent are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **varNum**: The zVariable number.
percent The reserve percentage.

6.3.63.1. Example(s)

The following example sets the reserve percentage to 10 for variable “MY_VAR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long percent; /* The reserve percentage. */

percent = 10L;
status = CDFsetzVarReser

6.3.64 CDFsetzVarsCacheSize

CDFstatus CDFsetzVarsCacheSize( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long numBuffers); /* in -- Number of cache buffers. */

CDFsetzVarsCacheSize specifies the number of cache buffers to be used for all of the variables. This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library.

The arguments to CDFsetzVarsCacheSize are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numBuffers The number of buffers.

6.3.64.1. Example(s)

The following example sets the number of cache buffers to 10 for all variables in a CDF.

```c
#include "cdf.h"
```
CDFid id; /* CDF identifier. */
long numBuffers; /* The number of cache buffers. */

numBuffers = 10L;
status = CDFsetzVarsCacheSize (id, numBuffers);
if (status != CDF_OK) UserStatusHandler (status);

6.3.65  CDFsetzVarSeqPos

CDFstatus CDFsetzVarSeqPos(
/* out -- Completion status code. */
CDFid id,
/* in -- CDF identifier. */
long varNum,
/* in -- Variable number. */
long recNum,
/* in -- Record number. */
long indices[]);
/* in -- Indices in a record. */

CDFsetzVarSeqPos specifies the current sequential value (position) for sequential access for the specified zVariable in a CDF. Note that a current sequential value is maintained for each zVariable individually. Use CDFgetzVarSeqPos function to get the current sequential value.

The arguments to CDFsetzVarSeqPos are defined as follows:

id  The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

varNum  The zVariable number.

recNum  The zVariable record number.

indices  The dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional zVariable, this argument is ignored, but must be presented.

6.3.65.1. Example(s)

The following example sets the current sequential value to the first value element in record number 2 for a zVariable, a 2-dimensional variable, in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
long varNum; /* The variable number. */
long recNum; /* The record number. */
long indices[2]; /* The indices. */
```
recNum = 2L;
indices[0] = 0L;
indices[1] = 0L;
status = CDFsetzVarSeqPos (id, varNum, recNum, indices);
if (status != CDF_OK) UserStatusHandler (status);

6.3.66  CDFsetzVarSparseRecords

CDFstatus CDFsetzVarSparseRecords(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long varNum,  /* in -- The variable number. */
long sRecordsType);  /* in -- The sparse records type. */

CDFsetzVarSparseRecords specifies the sparse records type of the specified zVariable in a CDF. Refer to Section 4.11.1 for the description of sparse records.

The arguments to CDFsetzVarSparseRecords are defined as follows:

- id: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- varNum: The zVariable number.
- sRecordsType: The sparse records type.

6.3.66.1. Example(s)

The following example sets the sparse records type to PAD_SPARSERECords from its original type for zVariable "MY_VAR" in a CDF.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
long sRecordsType;  /* The sparse records type. */

sRecordsType = PAD_SPARSERECords;
status = CDFsetzVarSparseRecords (id, CDFgetVarNum(id, "MY_VAR"), sRecordsType);
if (status != CDF_OK) UserStatusHandler (status);
```
6.4 Attributes/Entries

This section provides functions that are related to CDF attributes or attribute entries. An attribute is identified by its name or an number in the CDF. Before you can perform any operation on an attribute or attribute entry, the CDF in which it resides must be opened.

6.4.1 CDFconfirmAttrExistence

CDFstatus CDFconfirmAttrExistence( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
char * attrName) /* in -- Attribute name. */

CDFconfirmAttrExistence confirms whether an attribute exists for the given attribute name in a CDF. If the attribute doesn’t exist, an error is returned.

The arguments to CDFconfirmAttrExistence are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **attrName**: The attribute name to check.

6.4.1.1. Example(s)

The following example checks whether the attribute by the name of “ATTR_NAME1” is in a CDF:

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */

status = CDFconfirmAttrExistence (id, "ATTR_NAME1");
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.2 CDFconfirmgEntryExistence

CDFstatus CDFconfirmgEntryExistence( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute number. */
long entryNum); /* in -- gEntry number. */

CDFconfirmgEntryExistence confirms whether an attribute entry exists for the given attribute number and entry number in a CDF. If the attribute entry doesn’t exist, an error is returned.

The arguments to CDFconfirmgEntryExistence are defined as follows:

- **id**: The identifier of the CDF.

- **attrNum**: The attribute number.

- **entryNum**: The entry number.
CDFconfirmEntryExistence confirms the existence of the specified entry (gEntry), in a global attribute from a CDF. If the gEntry does not exist, the informational status code NO_SUCH_ENTRY will be returned.

The arguments to CDFconfirmEntryExistence are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum**: The (global) attribute number.
- **entryNum**: The gEntry number.

### 6.4.2.1. Example(s)

The following example checks the existence of gEntry numbered 1 for attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id;       /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum;   /* Attribute number. */
long entryNum;  /* gEntry number. */

attrNum = CDFgetAttrNum(id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
entryNum = 1L;
status = CDFconfirmEntryExistence (id, attrNum, entryNum);
if (status == NO_SUCH_ENTRY) UserStatusHandler (status);
```

### 6.4.3 CDFconfirmrEntryExistence

CDFconfirmrEntryExistence confirms the existence of the specified entry (rEntry), corresponding to an rVariable, in a variable attribute from a CDF. If the rEntry does not exist, the informational status code NO_SUCH_ENTRY will be returned.

The arguments to CDFconfirmrEntryExistence are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum  The variable attribute number.

entryNum  The rEntry number.

6.4.3.1. Example(s)

The following example checks the existence of an rEntry, corresponding to rVariable “MY_VAR”, for attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid  id;  /*  CDF identifier. */
CDFstatus status;  /*  Returned status code. */
long attrNum;  /*  Attribute number. */
long entryNum;  /*  rEntry number. */

attrNum = CDFgetAttrNum(id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(....);
entryNum = CDFgetVarNum(id, "MY_VAR");
if (entryNum < CDF_OK) QuitError(....);
status = CDFconfirmrEntryExistence (id, attrNum, entryNum);
if (status == NO_SUCH_ENTRY) UserStatusHandler (status);
```

6.4.4 CDFconfirmzEntryExistence

```c
CDFstatus CDFconfirmzEntryExistence(  /*  out -- Completion status code. */
    CDFid  id,  /*  in -- CDF identifier. */
    long attrNum,  /*  in -- Attribute number. */
    long entryNum);  /*  in -- zEntry number. */
```

CDFconfirmzEntryExistence confirms the existence of the specified entry (zEntry), corresponding to a zVariable, in a variable attribute from a CDF. If the zEntry does not exist, the informational status code NO_SUCH_ENTRY will be returned.

The arguments to CDFconfirmzEntryExistence are defined as follows:

- **id**  The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum**  The (variable) attribute number.
- **entryNum**  The zEntry number.
6.4.4.1. Example(s)

The following example checks the existence of the zEntry corresponding to zVariable “MY_VAR” for the variable attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long entryNum; /* zEntry number. */

attrNum = CDFgetAttrNum(id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
entryNum = CDFgetVarNum(id, "MY_VAR");
if (entryNum < CDF_OK) QuitError(…);
status = CDFconfirmzEntryExistence (id, attrNum, entryNum);
if (status == NO_SUCH_ENTRY) UserStatusHandler (status);
```

6.4.5 CDFcreateAttr

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

CDFcreateAttr creates an attribute with the specified scope in a CDF. It is identical to the original Standard Interface function CDFattrCreate. An attribute with the same name must not already exist in the CDF.

The arguments to CDFcreateAttr are defined as follows:

- `id`: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- `attrName`: The name of the attribute to create. This may be at most CDF_ATTR_NAME_LEN256 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 4.12.
- `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 CDFgetAttrNum function.
6.4.5.1. 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 = CDFcreateAttr (id, "TITLE", TITLEattrScope, &TITLEattrNum);
if (status != CDF_OK) UserStatusHandler (status);
status = CDFcreateAttr (id, UNITSattrName, VARIABLE_SCOPE, &UNITSattrnum);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.6 CDFdeleteAttr

CDFstatus CDFdeleteAttr( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum); /* in -- Attribute identifier. */

CDFdeleteAttr deletes the specified attribute from a CDF.

The arguments to CDFdeleteAttr are defined as follows:

- **id** The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum** The attribute number to be deleted.

6.4.6.1. Example(s)

The following example deletes an existing attribute named MY_ATTR from a CDF.

```
#include "cdf.h"
```
6.4.7 CDFdeleteAttrgEntry

CDFdeleteAttrgEntry deletes the specified entry (gEntry) in a global attribute from a CDF.

The arguments to CDFdeleteAttrgEntry are defined as follows:

- **id**     The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum** The global attribute number from which to delete an attribute entry.
- **entryNum** The gEntry number to delete.

6.4.7.1. Example(s)

The following example deletes the entry number 5 from an existing global attribute MY_ATTR in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long entryNum; /* gEntry number. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(....);
entryNum = 5L;
status = CDFdeleteAttrgEntry (id, attrNum, entryNum);
```
if (status != CDF_OK) UserStatusHandler (status);

6.4.8  CDFdeleteAttrrEntry

CDFstatus CDFdeleteAttrrEntry(  /* out -- Completion status code. */
CDFid id,  /* in -- CDF identifier. */
long attrNum,  /* in -- Attribute identifier. */
long entryNum);  /* in -- rEntry identifier. */

CDFdeleteAttrrEntry deletes the specified entry (rEntry), corresponding to an rVariable, in an (variable) attribute from a CDF.

The arguments to CDFdeleteAttrrEntry are defined as follows:

id  The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum  The (variable) attribute number.

entryNum  The rEntry number.

6.4.8.1.  Example(s)

The following example deletes the entry corresponding to rVariable “MY_VAR1” from the variable attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long attrNum;  /* Attribute number. */
long entryNum;  /* rEntry number. */

attrNum = CDFgetAttrNum (id, “MY_ATTR”);
if (attrNum < CDF_OK) QuitError(…);
entryNum = CDFgetVarNum(id, “MY_VAR1”);
if (entryNum < CDF_OK) QuitError(…);
status = CDFdeleteAttrrEntry (id, attrNum, entryNum);
if (status != CDF_OK) UserStatusHandler (status);
```
6.4.9  CDFdeleteAttrzEntry

CDFstatus CDFdeleteAttrzEntry( /* out -- Completion status code. */
  CDFid id,          /* in -- CDF identifier. */
  long attrNum,      /* in -- Attribute identifier. */
  long entryNum);    /* in -- zEntry identifier. */

CDFdeleteAttrzEntry deletes the specified entry (zEntry), corresponding to a zVariable, in an (variable) attribute from a CDF.

The arguments to CDFdeleteAttrzEntry are defined as follows:

  id    The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

  attrNum The identifier of the variable attribute.

  entryNum The zEntry number to be deleted that is the zVariable number.

6.4.9.1.  Example(s)

The following example deletes the variable attribute entry named MY_ATTR that is attached to the zVariable MY_VAR1.

```
#include "cdf.h"

CDFid id;          /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long attrNum;      /* Attribute number. */
long entryNum;     /* zEntry number. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(….);
entryNum = CDFgetVarNum(id, "MY_VAR1");
if (entryNum < CDF_OK) QuitError(….);
status = CDFdeleteAttrzEntry (id, attrNum, entryNum);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.10  CDFgetAttrgEntry
CDFstatus CDFgetAttrgEntry (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long entryNum, /* in -- gEntry number. */
void *value); /* out -- gEntry data. */

This function is identical to the original Standard Interface function CDFattrGet. CDFgetAttrgEntry is used to read a
global attribute entry from a CDF. In most cases it will be necessary to call CDFinquireAttrgEntry before calling
CDFgetAttrgEntry in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to CDFgetAttrgEntry are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate
(or CDFcreateCDF) or CDFopenCDF.

attrNum The attribute number. This number may be determined with a call to CDFgetAttrNum.

entryNum The global attribute entry number.

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.

6.4.10.1. Example(s)

The following example displays the value of the global attribute called HISTORY. 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).

```c
#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, "HISTORY");
if (attrN < CDF_OK) UserStatusHandler (attrN); /* If less than zero (0), then it must be a warning/error
code. */

entryN = 0;

status = CDFinquireAttrgEntry (id, attrN, entryN, &dataType, &numElems);
if (status != CDF_OK) UserStatusHandler (status);
```
if (dataType == CDF_CHAR) {
    buffer = (char *) malloc (numElems + 1);
    if (buffer == NULL)...

    status = CDFgetAttrgEntry (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);
}.
.
.
6.4.11 CDFgetAttrgEntryDataType

CDFstatus CDFgetAttrgEntryDataType ( /* out -- Completion status code. */
    CDFid id, /* in -- CDF identifier. */
    long attrNum, /* in -- Attribute identifier. */
    long entryNum, /* in -- gEntry number. */
    long *dataType); /* out -- gEntry data type. */

CDFgetAttrgEntryDataType returns the data type of the specified global attribute and gEntry number in a CDF. The data types are described in Section 4.5.

The arguments to CDFgetAttrgEntryDataType are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum The global attribute number.
entryNum The gEntry number.
datatype The data type of the gEntry.

6.4.11.1. Example(s)

The following example gets the data type for the gEntry numbered 2 from the global attribute “MY_ATTRIB” in a CDF.

.
.
#include "cdf.h"
.
.
CDFid id; /* CDF identifier. */
6.4.12 CDFgetAttrgEntryNumElements

CDFgetAttrgEntryNumElements returns the number of elements of the specified global attribute and gentry number in a CDF.

The arguments to CDFgetAttrgEntryNumElements are defined as follows:

- id: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- attrNum: The identifier of the global attribute.
- entryNum: The gEntry number.
- numElems: The number of elements of the gEntry.

6.4.12.1. Example(s)

The following example gets the number of elements from the gEntry numbered 2 from the global attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long entryNum; /* gEntry number. */

attrNum = CDFgetAttrNum (id, “MY_ATTR”);
if (attrNum < CDF_OK) QuitError(…);
entryNum = 2L;
status = CDFgetAttrgEntryDataType (id, attrNum, entryNum, &dataType);
if (status != CDF_OK) UserStatusHandler (status);
```
attrNum = CDFgetAttrNum (id, “MY_ATTR”);  
if (attrNum < CDF_OK) QuitError(….);  
entryNum = 2L;  
status = CDFgetAttrEntryNumElements (id, attrNum, entryNum, &numElements);  
if (status != CDF_OK) UserStatusHandle (status);  

6.4.13  CDFgetAttrEntry

CDFstatus CDFgetAttrEntry (  
CDFid id,  
long attrNum,  
long entryNum,  
void *value);  

This function is identical to the original Standard Interface function CDFattrGet.  CDFgetAttrEntry is used to read an 
variable attribut

e entry from a CDF.  In most cases it will be necessary to call CDFattrEntryInquire before calling 

equireAttrEntry in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to CDFgetAttrEntry are defined as follows:

id  The identifier of the CDF.  This identifier must have been initialized by a call to CDFcreate 
(or CDFcreateCDF) or CDFopenCDF.

attrNum  The attribute number.  This number may be determined with a call to CDFgetAttrNum.

entryNum  The variable attribute entry number that is the variable number from which the attribute is 
read.

value  The entry 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.

6.4.13.1.  Example(s)

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

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

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

status = CDFinquireAttrrEntry (id, attrN, entryN, &dataType, &numElems);

if (status != CDF_OK) UserStatusHandler (status);
if (dataType == CDF_CHAR) {
    buffer = (char *) malloc (numElems + 1);
    if (buffer == NULL)...
    status = CDFgetAttrrEntry (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);
}

6.4.14  CDFgetAttrMaxgEntry

CDFstatus CDFgetAttrMaxgEntry (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long *maxEntry); /* out -- The last gEntry number. */

CDFgetAttrMaxgEntry returns the last entry number of the specified global attribute in a CDF.

The arguments to CDFgetAttrMaxgEntry are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum The identifier of the global attribute.
maxEntry  The last gEntry number.

6.4.14.1. Example(s)

The following example gets the last entry number from the global attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long maxEntry; /* The last gEntry number. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
status = CDFgetAttrMaxgEntry (id, attrNum, &maxEntry);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.15  CDFgetAttrMaxrEntry

CDFstatus CDFgetAttrMaxrEntry (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long *maxEntry); /* out -- The maximum rEntry number. */

CDFgetAttrMaxrEntry returns the last rEntry number (rVariable number) to which the given variable attribute is attached.

The arguments to CDFgetAttrMaxrEntry are defined as follows:

- id  The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- attrNum  The identifier of the variable attribute.
- maxEntry  The last rEntry number (rVariable number) to which attrNum is attached.

6.4.15.1. Example(s)

The following example gets the last entry, corresponding to the last rVariable number, from the variable attribute “MY_ATTR” in a CDF.
#include "cdf.h"

cdfid id; /* CDF identifier. */
cdfstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long maxEntry; /* The last rEntry number. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
status = CDFgetAttrMaxrEntry (id, attrNum, &maxEntry);
if (status != CDF_OK) UserStatusHandler (status);

6.4.16 CDFgetAttrMaxzEntry

CDFstatus CDFgetAttrMaxzEntry (/* out -- Completion status code. */
cdfid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long *maxEntry); /* out -- The maximum zEntry number. */

CDFgetAttrMaxzEntry returns the last entry number, corresponding to the last zVariable number, to which the given variable attribute is attached.

The arguments to CDFgetAttrMaxzEntry are defined as follows:

- id: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- attrNum: The identifier of the variable attribute.
- maxEntry: The last zEntry number (zVariable number) to which attrNum is attached.

6.4.16.1. Example(s)

The following example gets the last entry, corresponding to the last zVariable number, attached to the variable attribute MY_ATTR in a CDF.

#include "cdf.h"
cdfid id; /* CDF identifier. */
cdfstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long maxEntry; /* The last zEntry number that is the last zVariable added */

attrNum = CDFgetAttrNum (id, “MY_ATTR”);
if (attrNum < CDF_OK) QuitError(…);
status = CDFgetAttrMaxzEntry (id, attrNum, &maxEntry);
if (status != CDF_OK) UserStatusHandler (status);

6.4.17 CDFgetAttrName

CDFstatus CDFgetAttrName (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
char *attrName); /* out -- The attribute name. */

CDFgetAttrName gets the name of the specified attribute (by its number) in a CDF.

The arguments to CDFgetAttrName are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum The identifier of the attribute.

attrName The name of the attribute.

6.4.17.1. Example(s)

The following example retrieves the name of the attribute number 2, if it exists, in a CDF.

#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
char attrName[CDF_ATTR_NAME_LEN256]; /* The attribute name. */

attrNum = 2L;
status = CDFgetAttrName (id, attrNum, attrName);
if (status != CDF_OK) UserStatusHandler (status);
6.4.18  CDFgetAttrNum

```c
long CDFgetAttrNum ( /* out -- Attribute number. */
CDFid id, /* in -- CDF identifier. */
char *attrName); /* in -- The attribute name. */
```

CDFgetAttrNum is used to determine the attribute number associated with a given attribute name. If the attribute is found, CDFgetAttrNum 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 CDFgetAttrNum are defined as follows:

- `id` The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- `attrName` The name of the attribute for which to search. This may be at most CDF_ATTR_NAME_LEN256 characters (excluding the NUL terminator). Attribute names are case-sensitive.

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

6.4.18.1. Example(s)

In the following example the attribute named pressure will be renamed to PRESSURE with CDFgetAttrNum being used as an embedded function call. Note that if the attribute pressure did not exist in the CDF, the call to CDFgetAttrNum 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.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */

status = CDFrenameAttr (id, CDFgetAttrNum(id,"pressure"), "PRESSURE");
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.19  CDFgetAttrEntryDataType

```c
CDFstatus CDFgetAttrEntryDataType ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long entryNum, /* in -- rEntry number. */
long *dataType); /* out -- rEntry data type. */
```
CDFgetAttrEntryDataType returns the data type of the rEntry from an (variable) attribute in a CDF. The data types are described in Section 4.5.

The arguments to CDFgetAttrEntryDataType are defined as follows:

- **id** The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum** The identifier of the variable attribute.
- **entryNum** The rEntry number.
- **dataType** The data type of the rEntry.

### 6.4.19.1. Example(s)

The following example gets the data type for the entry of rVariable “MY_VAR1” in the (variable) attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long entryNum; /* rEntry number. */
long dataType; /* rEntry data type. */

attrNum = CDFgetAttrNum(id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
entryNum = CDFgetVarNum(id, "MY_VAR1");
if (entryNum < CDF_OK) QuitError(…);
status = CDFgetAttrEntryDataType (id, attrNum, entryNum, &dataType);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.4.20 CDFgetAttrEntryNumElements

CDFstatus CDFgetAttrEntryNumElements (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long startRec, /* in -- rEntry number. */
long *numElems); /* out -- rEntry’s number of elements. */

CDFgetAttrEntryNumElements returns the number of elements of the rEntry from an (variable) attribute in a CDF.
The arguments to CDFgetAttrEntryNumElements are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum**: The identifier of the variable attribute.
- **entryNum**: The rEntry number.
- **numElems**: The number of elements of the rEntry.

### 6.4.20.1. Example(s)

The following example gets the number of elements for the entry of rVariable “MY_VAR1” in the (variable) attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long entryNum; /* rEntry number. */
long numElements; /* rEntry’s number of elements. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
entryNum = CDFgetVarNum(id, "MY_VAR1");
if (entryNum < CDF_OK) QuitError(…);
status = CDFgetAttrEntryNumElements (id, attrNum, entryNum, &numElements);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.4.21 CDFgetAttrScope

CDFgetAttrScope returns the attribute scope (GLOBAL_SCOPE or VARIABLE_SCOPE) of the specified attribute in a CDF. Refer to Section 4.12 for the description of the attribute scopes.

The arguments to CDFgetAttrScope are defined as follows:
The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum  The attribute number.

attrScope  The scope of the attribute.

6.4.21.1. Example(s)

The following example gets the scope of the attribute “MY_ATTR” in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long attrScope; /* Attribute scope. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
status = CDFgetAttrScope (id, attrNum, &attrScope);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.22  CDFgetAttrzEntry

CDFstatus CDFgetAttrzEntry(id, /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Variable attribute number. */
long entryNum, /* in -- Entry number. */
void *value); /* out -- Entry value. */

CDFgetAttrzEntry is used to read zVariable’s attribute entry. In most cases it will be necessary to call CDFinquireAttrzEntry before calling this function in order to determine the data type and number of elements (of that data type) for dynamical space allocation for the entry.

The arguments to CDFgetAttrzEntry are defined as follows:

id  The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum  The variable attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum  The variable attribute entry number that is the zVariable number from which the attribute entry is read

value  The entry 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.

6.4.22.1. Example(s)

The following example displays the value of the UNITS attribute for the PRES_LVL zVariable (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).

```c
#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 = CDFgetAttrNum (id, "UNITS");
if (attrN < CDF_OK) UserStatusHandler (attrN);

entryN = CDFgetVarNum (id, "PRES_LVL");  /*  The zEntry number is the zVariable number. */
if (entryN < CDF_OK) UserStatusHandler (entryN);  /*  If less than zero (0), then it must be a warning/error code. */
status = CDFinquireAttrzEntry (id, attrN, entryN, &dataType, &numElems);

if (status != CDF_OK) UserStatusHandler (status);
if (dataType == CDF_CHAR) {
    buffer = (char *) malloc (numElems + 1);
    if (buffer == NULL)...
    status = CDFgetAttrzEntry (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);
}
```
6.4.23  CDFgetAttrzEntryDataType

CDFstatus CDFgetAttrzEntryDataType ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute identifier. */
long entryNum, /* in -- zEntry number. */
long *dataType); /* out -- zEntry data type. */

CDFgetAttrzEntryDataType returns the data type of the zEntry for the specified variable attribute in a CDF. The data
types are described in Section 4.5.

The arguments to CDFgetAttrzEntryDataType are defined as follows:

id  The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or
    CDFcreateCDF) or CDFopenCDF.
attrNum The identifier of the variable attribute.
entryNum The zEntry number that is the zVariable number.
dataType The data type of the zEntry.

6.4.23.1. Example(s)

The following example gets the data type of the attribute named MY_ATTR for the zVariable MY_VAR1 in a CDF.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrNum; /* Attribute number. */
long entryNum; /* zEntry number. */
long dataType; /* zEntry data type. */

attrNum = CDFgetAttrNum (id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
entryNum = CDFgetVarNum(id, "MY_VAR1");
if (entryNum < CDF_OK) QuitError(…);
status = CDFgetAttrzEntryDataType (id, attrNum, entryNum, &dataType);
if (status != CDF_OK) UserStatusHandler (status);
```

182
6.4.24  CDFgetAttrzEntryNumElements

CDFstatus CDFgetAttrzEntryNumElements (/*  out --  Completion status code. */
CDFid  id,         /*  in --  CDF identifier. */
long attrNum,     /*  in --  Attribute identifier. */
long entryNum,    /*  in --  zEntry number. */
long *numElems);  /*  out --  zEntry’s number of elements. */

CDFgetAttrzEntryNumElements returns the number of elements of the zEntry for the specified variable attribute in a CDF.

The arguments to CDFgetAttrzEntryNumElements are defined as follows:

id        The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum    The identifier of the variable attribute.
entryNum   The zEntry number that is the zVariable number.
numElems   The number of elements of the zEntry.

6.4.24.1. Example(s)

The following example returns the number of elements for attribute named MY_ATTR for the zVariable MY_VAR1 in a CDF.

```
#include "cdf.h"

CDFid  id;         /*  CDF identifier. */
CDFstatus  status; /*  Returned status code. */
long  attrNum;    /*  Attribute number. */
long  entryNum;   /*  zEntry number. */
long  numElements; /*  zEntry’s number of elements. */

attrNum = CDFgetAttrNum (id, “MY_ATTR”);
if (attrNum < CDF_OK) QuitError(…..);
entryNum = CDFgetVarNum(id, “MY_VAR1”);
if (entryNum < CDF_OK) QuitError(…..);
status = CDFgetAttrzEntryNumElements (id, attrNum, entryNum, &numElements);
if (status !=  CDF_OK) UserStatusHandler (status);
```
6.4.25  CDFgetNumAttrgEntries

CDFstatus CDFgetNumAttrgEntries (CDFid id, long attrNum, long *entries);

CDFgetNumAttrgEntries returns the total number of entries (gEntries) written for the specified global attribute in a CDF.

The arguments to CDFgetNumAttrgEntries are defined as follows:

- **id** (CDFid) The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum** (long) The attribute number.
- **entries** (long) Number of gEntries for attrNum.

6.4.25.1. Example(s)

The following example retrieves the total number of gEntries for the global attribute MY_ATTR in a CDF.

```c
#include "cdf.h"

CDFstatus status; /* Returned status code. */
CDFid id; /* CDF identifier. */
long attrNum; /* Attribute number. */
long numEntries; /* Number of entries. */
int i;

attrNum = CDFgetAttrNum(id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
status = CDFgetNumAttrgEntries (id, attrNum, &numEntries);
if (status != CDF_OK) UserStatusHandler (status);
for (i=0; i < numEntries; i++) {
  /* process an entry */
}
```

184
6.4.26 CDFgetNumAttributes

CDFstatus CDFgetNumAttributes (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *numAttr); /* out -- Total number of attributes. */

CDFgetNumAttributes returns the total number of global and variable attributes in a CDF.

The arguments to CDFgetNumAttributes are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numAttr The total number of global and variable attributes.

6.4.26.1. Example(s)

The following example returns the total number of global and variable attributes in a CDF.

```
#include "cdf.h"

CDFstatus status; /* Returned status code. */
CDFid id; /* CDF identifier. */
long numAttr; /* Number of attributes. */

status = CDFgetNumAttributes (id, &numAttr);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.27 CDFgetNumAttrrEntries

CDFstatus CDFgetNumAttrrEntries (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute number. */
long *entries); /* out -- Total rEntries. */

CDFgetNumAttrrEntries returns the total number of entries (rEntries) written for the rVariables in the specified (variable) attribute of a CDF.

The arguments to CDFgetNumAttrrEntries are defined as follows:
The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum
The attribute number.

entries
Total rEntries.

### 6.4.27.1. Example(s)

The following example returns the total number of rEntries from the variable attribute “MY_ATTRIB” in a CDF.

```c
#include "cdf.h"

CDFstatus status; /* Returned status code. */
CDFid id; /* CDF identifier. */
long attrNum; /* Attribute number. */
long entries; /* Number of entries. */

attrNum = CDFgetAttrNum(id, "MY_ATTRIB");
if (attrNum < CDF_OK) QuitError(...);
status = CDFgetNumAttrEntries (id, attrNum, &entries);
if (status != CDF_OK) UserStatusHandler (status);
```

### 6.4.28 CDFgetNumAttrEntries

CDFstatus CDFgetNumAttrEntries ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute number. */
long *entries); /* out -- Total zEntries. */

CDFgetNumAttrEntries returns the total number of entries (zEntries) written for the zVariables in the specified variable attribute in a CDF.

The arguments to CDFgetNumAttrEntries are defined as follows:

id
The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum
The attribute number.

entries
Total zEntries.
6.4.28.1. Example(s)

The following example returns the total number of zEntries for the variable attribute MY_ATTR in a CDF.

```
#include "cdf.h"

CDFstatus status; /* Returned status code. */
CDFid id; /* CDF identifier. */
long attrNum; /* Attribute number. */
long entries; /* Number of entries. */

attrNum = CDFgetAttrNum(id, "MY_ATTR");
if (attrNum < CDF_OK) QuitError(…);
status = CDFgetNumAttrzEntries (id, attrNum, &entries);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.29 CDFgetNumgAttributes

CDFstatus CDFgetNumgAttributes ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long *numAttrs); /* out -- Total number of global attributes. */

CDFgetNumgAttributes returns the total number of global attributes in a CDF.

The arguments to CDFgetNumgAttributes are defined as follows:

- **id**: The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **numAttrs**: The number of global attributes.

6.4.29.1. Example(s)

The following example returns the total number of global attributes in a CDF.

```
#include "cdf.h"
```
6.4.30  CDFgetNumvAttributes

CDFstatus CDFgetNumvAttributes (/* out -- Completion status code. */
CDFid id,    /* in -- CDF identifier. */
long *numAttrs); /* out -- Total number of variable attributes. */

CDFgetNumvAttributes returns the total number of variable attributes in a CDF.

The arguments to CDFgetNumvAttributes are defined as follows:

id        The identifier of the current CDF. This identifier must have been initialized by a call to
           CDFcreate (or CDFcreateCDF) or CDFopenCDF.

numAttrs  The number of variable attributes.

6.4.30.1. Example(s)

The following example returns the total number of variable attributes of a CDF.

```c
#include "cdf.h"

CDFstatus status;    /* Returned status code. */
CDFid id;            /* CDF identifier. */
long numAttrs;       /* Number of variable attributes. */

status = CDFgetNumvAttributes (id, &numAttrs);
if (status != CDF_OK) UserStatusHandler (status);
```
CDFinquireAttr is used to inquire information about the specified attribute. This function expands the original Standard Interface function CDFattrInquire to provide an extra information about zEntry if the attribute has a variable scope.

The arguments to CDFinquireAttr are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum**: The attribute number to inquire. This number may be determined with a call to CDFgetAttrNum.
- **attrName**: The attribute's name that corresponds to attrNum. This character string must be large enough to hold CDF_ATTR_NAME_LEN256 + 1 characters (including the NUL terminator).
- **attrScope**: The scope of the attribute (GLOBAL_SCOPE or VARIABLE_SCOPE). Attribute scopes are defined in Section 4.12.
- **maxgEntry**: For vAttributes, this value of this field is -1 as it doesn’t apply to global attribute entry (gEntry). For gAttributes, this is the maximum entry (gentry) number used. This number may not correspond with the number of entries (if some entry numbers were not used). If no entries exist for the attribute, then the value of -1 is returned.
- **maxrEntry**: For gAttributes, this value of this field is -1 as it doesn’t apply to rVariable attribute entry (rEntry). For vAttributes, this is the maximum rVariable attribute entry (rEntry) number used. This number may not correspond with the number of entries (if some entry numbers were not used). If no entries exist for the attribute, then the value of -1 is returned.
- **maxzEntry**: For gAttributes, this value of this field is -1 as it doesn’t apply to zVariable attribute entry (zEntry). For vAttributes, this is the maximum zVariable attribute entry (zEntry) number used. This may not correspond with the number of entries (if some entry numbers were not used). If no entries exist for the attribute, then the value of -1 is returned.

### 6.4.31.1. Example(s)

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

```c
#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. */
int attrN; /* attribute number. */
char attrName[CDF_ATTR_NAME_LEN256+1]; /* attribute name -- +1 for NUL terminator. */
long attrScope; /* attribute scope. */
long maxgEntry, maxrEntry, maxzEntry; /* Maximum entry numbers. */

status = CDFinquireCDF (id, &numDims, dimSizes, &encoding, &majority, &maxRec,
 &numVars, &numAttrs);
if (status != CDF_OK) UserStatusHandler (status);
for (attrN = 0; attrN < (int)numAttrs; attrN++) {
    status = CDFinquireAttr (id, (long)attrN, attrName,
        &attrScope, &maxgEntry, &maxrEntry, &maxzEntry);
    if (status < CDF_OK) /* INFO status codes ignored. */
        UserStatusHandler (status);
    else
        printf ("%s\n", attrName);
}

6.4.32 CDFinquireAttrgEntry

CDFstatus CDFinquireAttrgEntry (/* out -- Completion status code. */
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). */

This function is identical to the original Standard Interface function CDFattrEntryInquire. CDFinquireAttrgEntry is used to inquire information about a global attribute entry.

The arguments to CDFinquireAttrgEntry are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to
    CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum The attribute number to inquire. This number may be determined with a call to
    CDFgetAttrNum.
entryNum The entry number to inquire.

dataType The data type of the specified entry. The data types are defined in Section 4.5.

NumElements The number of elements of the data type. For character 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.

6.4.32.1. Example(s)

The following example returns each entry for a global attribute named TITLE. 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.

```c
#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_LEN256+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 = CDFgetAttrNum (id, "TITLE");
if (attrN < CDF_OK) 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 = CDFinquireAttrEntry (id, attrN, entryN, &dataType, &numElems);
    if (status < CDF_OK) {
        if (status != NO_SUCH_ENTRY) UserStatusHandler (status);
    } else {
        /* process entries */
        .
    }
}
```
6.4.33  **CDFinquireAttrrEntry**

CDFstatus CDFinquireAttrrEntry (  
  /* out -- Completion status code. */  
  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). */

This function is identical to the original Standard Interface function CDFattrEntryInquire. CDFinquireAttrrEntry is used to inquire about an rVariable’s attribute entry.

The arguments to CDFinquireAttrrEntry are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum**: The attribute number to inquire. This number may be determined with a call to CDFgetAttrNum.
- **entryNum**: The entry number to inquire. This is the rVariable number (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 4.5.
- **NumElements**: The number of elements of the data type. For character 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.

6.4.33.1. Example(s)

The following example determines the data type of the “UNITS” attribute for the rVariable “Temperature”, then retrieves and displays the value of the UNITS attribute.

```c
#include "cdf.h"

CDFid id;  /* CDF identifier. */
CDFstatus status;  /* Returned status code. */
long attrN;  /* Attribute number. */
long entryN;  /* Entry number. */
char *buffer;
long dataType;  /* Data type. */
long numElems;  /* Number of elements (of the data type). */

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

entryN = CDFgetVarNum(id, "Temperature")
if (entryN < CDF_OK) UserStatusHandler (entryN);
```
status = CDFinquireAttrEntry (id, attrN, entryN, &dataType, &numElems);
if (status >= CDF_OK) {
    if (dataType == CDF_CHAR) {
        buffer = (char *) malloc (numElems + 1);
        if (buffer == NULL)...
        status = CDFgetAttrEntry (id, attrN, entryN, buffer);
        if (status != CDF_OK) UserStatusHandler (status);
        buffer[numElems] = '"'; /* NUL terminate. */
        printf ("Units of Temperature: %s\n", buffer);
        free (buffer);
    }
}

6.4.34 CDFinquireAttrzEntry

CDFstatus CDFinquireAttrzEntry ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- (Variable) Attribute number. */
long entryNum, /* in -- zEntry number. */
long *dataType, /* out -- Data type. */
long *numElements); /* out -- Number of elements (of the data type). */

CDFinquireAttrzEntry is used to inquire about a zVariable’s attribute entry.

The arguments to CDFinquireAttrzEntry are defined as follows:

id
The identifier of the CDF. This identifier must have been initialized by a call to
CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum
The (variable) attribute number for which to inquire an entry. This number may be
determined with a call to CDFgetAttrNum (see Section 6.4.18).

entryNum
The entry number to inquire. This is the zVariable number (the zVariable being
described in some way by the zEntry).

dataType
The data type of the specified entry. The data types are defined in Section 4.5.

NumElements
The number of elements of the data type. For character 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.

6.4.34.1. Example(s)

The following example determines the data type of the UNITS attribute for the zVariable Temperature, then retrieves
and displays the value of the UNITS attribute.
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long attrN; /* attribute number. */
long entryN; /* Entry number. */
char *buffer;
long dataType; /* Data type. */
long numElems; /* Number of elements (of the data type). */

attrN = CDFgetAttrNum (id, "UNITS");
if (attrN < CDF_OK) UserStatusHandler (attrN);
entryN = CDFgetVarNum(id, "Temperature")
if (entryN < CDF_OK) UserStatusHandler (entryN);
status = CDFinquireAttrzEntry (id, attrN, entryN, &dataType, &numElems);
if (status >= CDF_OK) {
  if (dataType == CDF_CHAR) {
    buffer = (char *) malloc (numElems + 1);
    if (buffer == NULL)...
    status = CDFgetAttrzEntry (id, attrN, entryN, buffer);
    if (status != CDF_OK) UserStatusHandler (status);
    buffer[numElems] = '\0'; /* NUL terminate. */
    printf ("Units of Temperature : %s\n", buffer);
    free (buffer);
  }
}

6.4.35 CDFputAttrgEntry

CDFstatus CDFputAttrgEntry( /* out -- Completion status code. */
  CDFid id, /* in -- CDF identifier. */
  long attrNum, /* in -- Attribute number. */
  long entryNum, /* in -- Attribute entry number. */
  long dataType, /* in -- Data type of this entry. */
  long numElements, /* in -- Number of elements in the entry (of the data type). */
  void *value); /* in -- Attribute entry value. */

CDFputAttrgEntry is used to write a global attribute entry. 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. A global attribute can have one or more attribute entries.

The arguments to CDFputAttrgEntry are defined as follows:
id

The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum

The attribute number. This number may be determined with a call to CDFgetAttrNum.

entryNum

The attribute entry number.

dataType

The data type of the specified entry. Specify one of the data types defined in Section 4.5.

numElements

The number of elements of the data type. For character 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.

6.4.35.1. Example(s)

The following example writes a global attribute entry to the global attribute called TITLE.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long entryNum; /* Attribute entry number. */
static char title[] = {"CDF title."}; /* Value of TITLE attribute, entry number 0. */

entryNum = 0;
status = CDFputAttrEntry (id, CDFgetAttrNum(id,"TITLE"), entryNum, CDF_CHAR, strlen(title), title);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.36 CDFputAttrEntry

CDFstatus CDFputAttrEntry(
    CDFid id, /* in -- CDF identifier. */
    long attrNum, /* in -- Attribute number. */
    long entryNum, /* in -- Attribute entry number. */
    long dataType, /* in -- Data type. */
    long numElems, /* in -- Number of elements in the entry. */
    void *value); /* in -- Attribute entry value. */
This function is identical to the original Standard Interface function CDFattrPut. CDFputAttrEntry is used to write rVariable’s attribute entry. 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 CDFputAttrEntry are defined as follows:

- **id**: The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

- **attrNum**: The attribute number. This number may be determined with a call to CDFgetAttrNum.

- **entryNum**: The attribute entry number that is the rVariable number to which this attribute entry belongs.

- **dataType**: The data type of the specified entry. Specify one of the data types defined in Section 4.5.

- **numElements**: The number of elements of the data type. For character 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.

### 6.4.36.1. Example(s)

The following example writes to the variable scope attribute VALIDs for the entry that corresponds to the rVariable TMP.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long entryNum; /* Entry number. */
long numElements; /* Number of elements (of data type). */

static short TMPvalids[] = {15,30}; /* Value(s) of VALIDs attribute, rEntry for rVariable TMP. */

numElements = 2;
status = CDFputAttrEntry (id, CDFgetAttrNum(id,"VALIDs"), CDFgetVarNum(id,"TMP"),
                          CDF_INT2, numElements, TMPvalids);
if (status != CDF_OK) UserStatusHandler (status);
```
6.4.37  CDFputAttrzEntry

CDFstatus CDFputAttrzEntry( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute number. */
long entryNum, /* in -- Attribute entry number. */
long dataType, /* in -- Data type of this entry. */
long numElements, /* in -- Number of elements in the entry (of the data type). */
void *value); /* in -- Attribute entry value. */

CDFputAttrzEntry is used to write zVariable’s attribute entry. 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 CDFputAttrzEntry are defined as follows:

id   The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum The (variable) attribute number. This number may be determined with a call to CDFgetAttrNum (see Section 6.4.18).

datatype The entry number that is the zVariable number to which this attribute entry belongs.

numElements The data type of the specified entry. Specify one of the data types defined in Section 4.5.

The number of elements of the data type. For character 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.

6.4.37.1. Example(s)

The following example writes a zVariable’s attribute entry. The entry has two elements (that is two values for non-CDF_CHAR type). The zEntry in the variable scope attribute VALIDs corresponds to the zVariable TMP.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long numElements; /* Number of elements (of data type). */

static short TMPvalids[] = {15,30}; /* Value(s) of VALIDs attribute, zEntry for zVariable TMP. */

numElements = 2;
```
status = CDFputAttrzEntry (id, CDFgetAttrNum(id,"VALIDs"),
CDFgetVarNum(id,"TMP"), CDF_INT2, numElements, TMPvalids);
if (status != CDF_OK) UserStatusHandler (status);

6.4.38 CDFrenameAttr

CDFstatus CDFrenameAttr(    /* out -- Completion status code. */
CDFid id,                  /* in -- CDF identifier. */
long attrNum,              /* in -- Attribute number. */
char *attrName);           /* in -- New attribute name. */

This function is identical to the original Standard Interface function CDFattrRename. CDFrenameAttr renames an existing attribute.

6.4.38.1. Example(s)

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

```
#include "cdf.h"

CDFid id;           /* CDF identifier. */
CDFstatus status;   /* Returned status code. */

status = CDFrenameAttr (id, CDFgetAttrNum(id,"LAT"), "LATITUDE");
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.39 CDFsetAttrgEntryDataSpec

CDFstatus CDFsetAttrgEntryDataSpec (    /* out -- Completion status code. */
CDFid id,                  /* in -- CDF identifier. */
long attrNum,              /* in -- Attribute number. */
long entryNum,             /* in -- gEntry number. */
long dataType)            /* in -- Data type. */

CDFsetAttrgEntryDataSpec respecifies the data type of a gEntry of a global attribute in a CDF. The new and old data type must be equivalent. Refer to the CDF User’s Guide for descriptions of equivalent data types.

The arguments to CDFsetAttrgEntryDataSpec are defined as follows:
id        The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum   The global attribute number.
entryNum  The gEntry number.
dataType  The new data type.

6.4.39.1. Example(s)

The following example modifies the third entry’s (entry number 2) data type of the global attribute MY_ATTR in a CDF. It will change its original data type from CDF_INT2 to CDF_UINT2.

```
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long entryNum; /* gEntry number. */
long dataType; /* The new data type */

entryNum = 2L;
dataType = CDF_UINT2;
numElems = 1L;
status = CDFsetAttrgEntryDataSpec (id, CDFgetAttrNum(id, "MY_ATTR"), entryNum, dataType);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.40  CDFsetAttrrEntryDataSpec

CDFstatus CDFsetAttrrEntryDataSpec (    /* out -- Completion status code. */
    CDFid id,      /* in -- CDF identifier. */
    long attrNum,  /* in -- Attribute number. */
    long entryNum, /* in -- rEntry number. */
    long dataType, /* in -- Data type. */
    long numElements); /* in -- Number of elements. */

CDFsetAttrrEntryDataSpec respecifies the data specification (data type and number of elements) of an rEntry of a variable attribute in a CDF. The new and old data type must be equivalent, and the number of elements must not be changed. Refer to the CDF User’s Guide for descriptions of equivalent data types.

The arguments to CDFsetAttrrEntryDataSpec are defined as follows:
id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum The variable attribute number.

typeNum The rEntry number.

dataType The new data type.

numElements The new number of elements.

6.4.40.1. Example(s)

The following example modifies the data specification for an rEntry, corresponding to rVariable “MY_VAR”, in the variable attribute “MY_ATTR” in a CDF. It will change its original data type from CDF_INT2 to CDF_UINT2.

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long dataType, numElements; /* Data type and number of elements. */

dataType = CDF_UINT2;
numElems = 1L;
status = CDFsetAttrrEntryDataSpec (id, CDFgetAttrNum(id, "MY_ATTR"), CDFgetVarNum(id, "MY_VAR"),
dataType, numElements);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.41 CDFsetAttrScope

CDFstatus CDFsetAttrScope ( /* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute number. */
long scope); /* in -- Attribute scope. */

CDFsetAttrScope respecifies the scope of an attribute in a CDF. Specify one of the scopes described in Section 4.12. Global-scoped attributes will contain only gEntries, while variable-scoped attributes can hold rEntries and zEntries.

The arguments to CDFsetAttrScope are defined as follows:

id The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.

attrNum The attribute number.
The new attribute scope. The value should be either VARIABLE_SCOPE or GLOBAL_SCOPE.

6.4.41.1. Example(s)

The following example changes the scope of the global attribute named MY_ATTR to a variable attribute (VARIABLE_SCOPE).

```c
#include "cdf.h"

CDFid id; /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long scope; /* New attribute scope. */

scope = VARIABLE_SCOPE;
status = CDFsetAttrScope (id, CDFgetAttrNum(id, "MY_ATTR"), scope);
if (status != CDF_OK) UserStatusHandler (status);
```

6.4.42 CDFsetAttrzEntryDataSpec

CDFstatus CDFsetAttrzEntryDataSpec (/* out -- Completion status code. */
CDFid id, /* in -- CDF identifier. */
long attrNum, /* in -- Attribute number. */
long entryNum, /* in -- zEntry number. */
long dataType) /* in -- Data type. */

CDFsetAttrzEntryDataSpec modifies the data type of a zEntry of a variable attribute in a CDF. The new and old data type must be equivalent. Refer to the CDF User’s Guide for the description of equivalent data types.

The arguments to CDFsetAttrzEntryDataSpec are defined as follows:

- **id** The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
- **attrNum** The variable attribute number.
- **entryNum** The zEntry number that is the zVariable number.
- **dataType** The new data type.
6.4.42.1. Example(s)

The following example respecifies the data type of the attribute entry of the attribute named MY_ATTR that is associated with the zVariable MY_VAR. It will change its original data type from CDF_INT2 to CDF_UINT2.

```c
#include "cdf.h"

CDFid id;    /* CDF identifier. */
CDFstatus status; /* Returned status code. */
long dataType;  /* Data type and number of elements. */

dataType = CDF_UINT2;
numElems = 1L;
status = CDFsetAttrzEntryDataSpec (id, CDFgetAttrNum(id, "MY_ATTR"),
                                   CDFgetVarNum(id, "MY_VAR"), dataType);
if (status != CDF_OK) UserStatusHandler (status);
```

```c
6.4.42.1. Example(s)
```
Chapter 7

7 Internal Interface - CDFlib

The Internal interface consists of only one routine, 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 consists of a function being 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 functions and corresponding items (on which to perform those functions) are described in Section 7.6.

The function prototype for CDFlib is as follows:

CDFstatus CDFlib (long function, ...);

This function prototype is found in the include file cdf.h.

7.1 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 CDFname[] = "test1";
/* File name of the CDF. */

long numDims = 2;
/* Number of dimensions. */

static long dimSizes[2] = {100,200};
/* Dimension sizes. */

long encoding = HOST_ENCODING;
/* Data encoding. */
```
long majority = ROW_MAJOR; /* Variable data majority. */
long format = SINGLE_FILE; /* Format of CDF. */

status = CDFcreate (CDFname, numDims, dimSizes, encoding, majority, &id);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (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 (assuming that is the default). 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 function 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.\(^{32}\) 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 function to perform. In this case it is a new function to perform - the NULL_ function. NULL_ indicates the end of the call to CDFlib. Specifying NULL_ at the end of the argument list is required because not all compilers/operating systems provide the ability for a called function to determine how many arguments were passed in by the calling function.

The next example shows how the same CDF could have been created using only one call to CDFlib. (The declarations would be the same.)

status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id,
    PUT__, CDF_ENCODING_, encoding,
    CDF_MAJORITY_, majority,
    CDF_FORMAT_, format,
    NULL_);
if (status != CDF_OK) UserStatusHandler (status);

The purpose of each argument is as follows:

CREATE_ The first function to be performed. In this case something will be created.

---

\(^{32}\) 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.
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.

CDFname The file name of the CDF.

numDims The number of dimensions in the CDF.

dimSizes The dimension sizes.

id The identifier to be used when referencing the created CDF in subsequent operations.

PUT_ This argument could have been one of two things. Another item to create or a new function to perform. In this case it is another function to perform - something will be put to 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 function 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 function 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 function to perform. Here it is another function to perform - the NULL_function 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).

7.2 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_>33 operation.

33 This notation is used to specify a function to be performed on an item. The syntax is <function__item__>.
There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.\(^\text{34}\)

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.

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.

\(^{34}\)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.
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 hyper 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.\footnote{The CDF library now maintains the current status code from one call to the next of CDFlib.}
7.3 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.

Chapter 8 explains how to interpret status codes. Appendix A lists the possible status codes and the type of each: error, warning, or informational.

7.4 Indentation/Style

Indentation should be used to make calls to CDFlib readable. The following example shows a call to CDFlib using proper indentation.

```
status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id,
                 PUT__, CDF_FORMAT_, format,
                 CDF_MAJORITY_, majority,
                 CREATE_, ATTR_, attrName, scope, &attrNum,
                 rVAR_, varName, dataType, numElements,
                 recVary, dimVarys, &varNum,
                 NULL_);
```

Note that the functions (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 functions.

The following example shows the same call to CDFlib without the proper indentation.

```
status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id, PUT__,
                 CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
                 ATTR_, attrName, scope, &attrNum, rVAR_, varName, dataType,
                 numElements, recVary, dimVarys, &varNum, NULL_);
```

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

7.5 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:

```c
status = CDFlib (fnc1, item1, arg1, arg2, ...argN,
                 item2, arg1, arg2, ...argN,
                 .
                 .
                 .
                 itemN, arg1, arg2, ...argN,
fnc2, item1, arg1, arg2, ...argN,
                 item2, arg1, arg2, ...argN,
                 .
                 .
                 .
                 itemN, arg1, arg2, ...argN,
                 .
                 .
                 .
                 fncN, item1, arg1, arg2, ...argN,
                 item2, arg1, arg2, ...argN,
                 .
                 .
                 .
                 itemN, arg1, arg2, ...argN,
NULL_);
```

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

### 7.6 Operations...

An operation consists of a function being performed on an item. The supported functions are as follows:

- `CLOSE_` Used to close an item.
- `CONFIRM_` Used to confirm the value of an item.
- `CREATE_` Used to create an item.
- `DELETE_` Used to delete an item.
- `GET_` Used to get (read) something from an item.
- `NULL_` Used to signal the end of the argument list of an internal interface call.
- `OPEN_` Used to open an item.
- `PUT_` Used to put (write) something to an item.
- `SELECT_` Used to select the value of an item.

For each function the supported items, required arguments, and required preselected objects-states are listed below. The required preselected objects-states are those objects-states that must be selected (typically with the `SELECT_` function) before a particular operation may be performed. Note that some of the required preselected objects-states have default values as described at Section 7.2.

**<CLOSE_,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.

<CLOSE_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.

<CLOSE_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.

<CONFIRM_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.

<CONFIRM_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:

in:  char *attrName

The attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<CONFIRM_CDF_>  
Confirms the current CDF. Required arguments are as follows:

out:  CDFid *id

The current CDF.

There are no required preselected objects/states.

<CONFIRM_CDF_ACCESS_>  
Confirms the accessibility of the current CDF. If a fatal error occurred while accessing the CDF the error code NO_MORE_ACCESS will be returned. If this is the case, the CDF should still be closed.

There are no required arguments.

The only required preselected object/state is the current CDF.

<CONFIRM_CDF_CACHESIZE_>
Confirms the number of cache buffers being used for the dotCDF file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: long *numBuffers

The number of cache buffers being used.

The only required preselected object/state is the current CDF.

<CONFIRM_,CDF_DECODING_>
Confirms the decoding for the current CDF. Required arguments are as follows:

out: long *decoding

The decoding. The decodings are described in Section 4.7.

The only required preselected object/state is the current CDF.

<CONFIRM_,CDF_NAME_>
Confirms the file name of the current CDF. Required arguments are as follows:

out: char CDFname[CDF_PATHNAME_LEN+1]

File name of the CDF.

The only required preselected object/state is the current CDF.

<CONFIRM_,CDF_NEGtoPOSfp0_MODE_>
Confirms the -0.0 to 0.0 mode for the current CDF. Required arguments are as follows:

out: long *mode

The -0.0 to 0.0 mode. The -0.0 to 0.0 modes are described in Section 4.15.

The only required preselected object/state is the current CDF.

<CONFIRM_,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 4.13.

The only required preselected object/state is the current CDF.

<CONFIRM_,CDF_STATUS_>
Confirms the current status code. Note that this is not the most recently returned status code but rather the most recently selected status code (see the <SELECT_,CDF_STATUS_> operation).

Required arguments are as follows:

out: CDFstatus *status

The status code.
The only required preselected object/state is the current status code.

<CONFIRM_<zMODE_>
Confirms the zMode for the current CDF. Required arguments are as follows:

    out: long *mode

    The zMode. The zModes are described in Section 4.14.

The only required preselected object/state is the current CDF.

<CONFIRM_<COMPRESS_CACHESIZE_>
Confirms the number of cache buffers being used for the compression scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

    out: long *numBuffers

    The number of cache buffers being used.

The only required preselected object/state is the current CDF.

<CONFIRM_<CURgENTRY_EXISTENCE_>
Confirms the existence of the gEntry at the current gEntry number for the current attribute (in the current CDF). If the gEntry does not exist, an error code will be returned.

There are no required arguments.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<CONFIRM_<CURrENTRY_EXISTENCE_>
Confirms the existence of the rEntry at the current rEntry number for the current attribute (in the current CDF). If the rEntry does not exist, an error code will be returned.

There are no required arguments.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<CONFIRM_<CURzENTRY_EXISTENCE_>
Confirms the existence of the zEntry at the current zEntry number for the current attribute (in the current CDF). If the zEntry does not exist, an error code will be returned.

There are no required arguments.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<CONFIRM_<gENTRY_>
Confirms the current gEntry number for all attributes in the current CDF. Required arguments are as follows:

    out: long *entryNum
The gEntry number.

The only required preselected object/state is the current CDF.

<CONFIRM_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. Required arguments are as follows:

  in: long entryNum

The gEntry number.

The required preselected objects/states are the current CDF and its current attribute.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<CONFIRM_rENTRY_*>
Confirms the current rEntry number for all attributes in the current CDF. Required arguments are as follows:

  out: long *entryNum

The rEntry number.

The only required preselected object/state is the current CDF.

<CONFIRM_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. Required arguments are as follows:

  in: long entryNum

The rEntry number.

The required preselected objects/states are the current CDF and its current attribute.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<CONFIRM_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.

<CONFIRM_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. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

  out: long *numBuffers
The number of cache buffers being used.

The required preselected objects/states are the current CDF and its current rVariable.

<CONFIRM_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:

in: char *varName

The rVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<CONFIRM_rVAR_PADVALUE_>
Confirms the existence of an explicitly specified pad value for the current rVariable (in the current CDF). If an explicit pad value has not been specified, the informational status code NO_PADVALUE_SPECIFIED will be returned.

There are no required arguments.

The required preselected objects/states are the current CDF and its current rVariable.

<CONFIRM_rVAR_reservepercent_>
Confirms the reserve percentage being used for the current rVariable (of the current CDF). This operation is only applicable to compressed rVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

out: long *percent

The reserve percentage.

The required preselected objects/states are the current CDF and its current rVariable.

<CONFIRM_rVAR_sequpos_>
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[CDF_MAX_DIMS]

Dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but must be present).

The required preselected objects/states are the current CDF and its current rVariable.

<CONFIRM_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[CDF_MAX_DIMS]
Dimension counts. Each element of counts receives the corresponding dimension count.

The only required preselected object/state is the current CDF.

<CONFIRM_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[CDF_MAX_DIMS]
```

Dimension indices. Each element of indices receives the corresponding dimension index.

The only required preselected object/state is the current CDF.

<CONFIRM_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: long intervals[CDF_MAX_DIMS]
```

Dimension intervals. Each element of intervals receives the corresponding dimension interval.

The only required preselected object/state is the current CDF.

<CONFIRM_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.

<CONFIRM_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.

<CONFIRM_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.

<CONFIRM_STAGE_CACHESIZE_>
Confirms the number of cache buffers being used for the staging scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:
out: long *numBuffers

The number of cache buffers being used.

The only required preselected object/state is the current CDF.

<CONFIRM_zENTRY_>
Confirms the current zEntry number for all attributes in the current CDF. Required arguments are as follows:

out: long *entryNum

The zEntry number.

The only required preselected object/state is the current CDF.

<CONFIRM_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. Required arguments are as follows:

in: long entryNum

The zEntry number.

The required preselected objects/states are the current CDF and its current attribute.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<CONFIRM_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.

<CONFIRM_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. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: long *numBuffers

The number of cache buffers being used.

The required preselected objects/states are the current CDF and its current zVariable.

<CONFIRM_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[CDF_MAX_DIMS]

Dimension counts. Each element of counts receives the corresponding dimension count.
The required preselected objects/states are the current CDF and its current zVariable.

<CONFIRM_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[CDF_MAX_DIMS]

Dimension indices. Each element of indices receives the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

<CONFIRM_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[CDF_MAX_DIMS]

Dimension intervals. Each element of intervals receives the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

<CONFIRM_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:

in:  char *varName

The zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<CONFIRM_zVAR_PADVALUE_>
Confirms the existence of an explicitly specified pad value for the current zVariable (in the current CDF). If an explicit pad value has not been specified, the informational status code NO_PADVALUE_SPECIFIED will be returned.

There are no required arguments.

The required preselected objects/states are the current CDF and its current zVariable.

<CONFIRM_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.

<CONFIRM_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.

<CONFIRM_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.

<CONFIRM_zVAR_RESERVEPERCENT_>
Confirms the reserve percentage being used for the current zVariable (of the current CDF). This operation is only applicable to compressed zVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

out: long *percent

The reserve percentage.

The required preselected objects/states are the current CDF and its current zVariable.

<CONFIRM_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[CDF_MAX_DIMS]

Dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but must be present).

The required preselected objects/states are the current CDF and its current zVariable.

<CREATE_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_LEN256 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 4.12.
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.

<CREATE_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 *CDFname

File name 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 file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

UNIX: File names 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. 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 must be present). 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 then be changed with the corresponding <PUT_CDF_FORMAT_>, <PUT_CDF_ENCODING_>, and <PUT_CDF_MAJORORITY_> operations if necessary.

A CDF must be closed with the <CLOSE_CDF_> operation to ensure that the CDF will be correctly written to disk.

There are no required preselected objects/states.

<CREATE_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_LEN256 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 4.5.

in: long numElements

Number of elements of the data type at each value. For character 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 data types this must be one (1) - multiple elements are not allowed for non-character data types.

in: long recVary

Record variance. Specify one of the variances described in Section 4.9.

in: long dimVarys[]

Dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9. For 0-dimensional rVariables this argument is ignored (but must be present).

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.

<CREATE_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_LEN256 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 4.5.

in: long numElements

Number of elements of the data type at each value. For character 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 data types this must be one (1) - multiple elements are not allowed for non-character 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. 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 must be present).

in: long recVary

Record variance. Specify one of the variances described in Section 4.9.

in: long dimVarys[]

Dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9. For a 0-dimensional zVariable this argument is ignored (but must be present).

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.

<DELETE_.ATTR_>
Deletes the current attribute (in the current CDF). Note that the attribute's entries are also deleted. The attributes, which numerically follow the attribute being deleted, are immediately renumbered. When the attribute is deleted, there is no longer a current attribute.

There are no required arguments.

The required preselected objects/states are the current CDF and its current attribute.

<DELETE_.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.

<DELETE_.gENTRY_>
Deletes the gEntry at the current gEntry number of the current attribute (in the current CDF). Note that this does not affect the current gEntry number.

There are no required arguments.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<DELETE_.rENTRY_>
Deletes the rEntry at the current rEntry number of the current attribute (in the current CDF). Note that this does not affect the current rEntry number.
There are no required arguments.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

**<DELETE_rVAR_>**
Deletes the current rVariable (in the current CDF). Note that the rVariable's corresponding rEntries are also deleted (from each vAttribute). The rVariables, which numerically follow the rVariable being deleted, are immediately renumbered. The rEntries, which numerically follow the rEntries being deleted, are also immediately renumbered. When the rVariable is deleted, there is no longer a current rVariable. **NOTE:** This operation is only allowed on single-file CDFs.

There are no required arguments.

The required preselected objects/states are the current CDF and its current rVariable.

**<DELETE_rVAR_RECORDS_>**
Deletes the specified range of records from the current rVariable (in the current CDF). If the rVariable has sparse records a gap of missing records will be created. If the rVariable does not have sparse records, the records following the range of deleted records are immediately renumbered beginning with the number of the first deleted record. **NOTE:** This operation is only allowed on single-file CDFs.

Required arguments are as follows:

```plaintext
in:  long  firstRecord
    The record number of the first record to be deleted.

in:  long  lastRecord
    The record number of the last record to be deleted.
```

The required preselected objects/states are the current CDF and its current rVariable.

**<DELETE_zENTRY_>**
Deletes the zEntry at the current zEntry number of the current attribute (in the current CDF). Note that this does not affect the current zEntry number.

There are no required arguments.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

**<DELETE_zVAR_>**
Deletes the current zVariable (in the current CDF). Note that the zVariable's corresponding zEntries are also deleted (from each vAttribute). The zVariables, which numerically follow the zVariable being deleted, are immediately renumbered. The rEntries, which numerically follow the rEntries being deleted, are also immediately renumbered. When the zVariable is deleted, there is no longer a current zVariable. **NOTE:** This operation is only allowed on single-file CDFs.

There are no required arguments.

The required preselected objects/states are the current CDF and its current rVariable.
<DELETE_zVAR_RECORDS_>  
Deletes the specified range of records from the current zVariable (in the current CDF). If the zVariable has 
sparse records a gap of missing records will be created. If the zVariable does not have sparse records, the 
records following the range of deleted records are immediately renumbered beginning with the number of the 
first deleted record. **NOTE:** This operation is only allowed on single-file CDFs. Required arguments are as 
follows:

**in:** long firstRecord  
The record number of the first record to be deleted.

**in:** long lastRecord  
The record number of the last record to be deleted.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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. Required arguments are as follows:

**out:** long *maxEntry  
The maximum gEntry number for the attribute. If no gEntries exist, then a value of –1 will be passed 
back.

The required preselected objects/states are the current CDF and its current attribute.  
**NOTE:** Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<GET_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. Required arguments are as follows:

**out:** long *maxEntry  
The maximum rEntry number for the attribute. If no rEntries exist, then a value of –1 will be passed 
back.

The required preselected objects/states are the current CDF and its current attribute.  
**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_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. Required arguments are as follows:

**out:** long *maxEntry  
The maximum zEntry number for the attribute. If no zEntries exist, then a value of –1 will be passed 
back.

The required preselected objects/states are the current CDF and its current attribute.  
**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.
<GET_ATTR_NAME_>
Inquires the name of the current attribute (in the current CDF). Required arguments are as follows:

out: char attrName[CDF_ATTR_NAME_LEN256+1]

Attribute name.

The required preselected objects/states are the current CDF and its current attribute.

<GET_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_LEN256 characters (excluding the NUL terminator).

out: long *attrNum

The attribute number.

The only required preselected object/state is the current CDF.

<GET_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. 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.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<GET_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. 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.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_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. 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.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

**<GET_ATTR_SCOPE >**
Inquires the scope of the current attribute (in the current CDF). Required arguments are as follows:

```cpp
out:  long  *scope
```

Attribute scope. The scopes are described in Section 4.12.

The required preselected objects/states are the current CDF and its current attribute.

**<GET_CDF_CHECKSUM__>**
Inquires the checksum mode of the current CDF. Required arguments are as follows:

```cpp
out:  long  *checksum
```

The checksum mode of the current CDF (NO_CHECKSUM or MD5_CHECKSUM). The checksum mode is described in Section 4.19.

The required preselected objects/states is the current CDF.

**<GET_CDF_COMPRESSION__>**
Inquires the compression type/parameters of the current CDF. This refers to the compression of the CDF - not of any compressed variables. Required arguments are as follows:

```cpp
out:  long  *cType
```

The compression type. The types of compressions are described in Section 4.10.

```cpp
out:  long  cParms[CDF_MAX_PARMS]
```

The compression parameters. The compression parameters are described in Section 4.10.

```cpp
out:  long  *cPct
```

If compressed, the percentage of the uncompressed size of the CDF needed to store the compressed CDF.

The only required preselected object/state is the current CDF.

**<GET_CDF_COPYRIGHT__>**
Reads the Copyright notice for the CDF library that created the current CDF. Required arguments are as follows:

```cpp
out:  char  Copyright[CDF_COPYRIGHT_LEN+1]
```

CDF Copyright text.

The only required preselected object/state is the current CDF.

**<GET_CDF_ENCODING__>**
Inquires the data encoding of the current CDF. Required arguments are as follows:

```cpp
out:  long  *encoding
```
Data encoding. The encodings are described in Section 4.6.

The only required preselected object/state is the current CDF.

<GET_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 4.4.

The only required preselected object/state is the current CDF.

<GET_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.

<GET_CDF_INFO_>  
Inquires the compression type/parameters of a CDF without having to open the CDF. This refers to the compression of the CDF - not of any compressed variables. Required arguments are as follows:

  in: char *CDFname

  File name of the CDF to be inquired. (Do not append an extension.) This can be at most CDF_PATHNAME_LEN characters (excluding the NUL terminator). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

  UNIX: File names are case-sensitive.

  out: long *cType

  The CDF compression type. The types of compressions are described in Section 4.10.

  out: long cParms[CDF_MAX_PARMS]

  The compression parameters. The compression parameters are described in Section 4.10.

  out: OFF_T³⁶ *cSize

  If compressed, size in bytes of the dotCDF file. If not compressed, set to zero (0).

  out: OFF_T⁵ *uSize

  If compressed, size in bytes of the dotCDF file when decompressed. If not compressed, size in bytes of the dotCDF file.

³⁶ It is type long for V2.6 and V2.7.
There are no required preselected objects/states.

<GET_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 4.8.

The only required preselected object/state is the current CDF.

<GET_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.

<GET_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.

<GET_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.

<GET_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.

<GET_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.

\texttt{<GET\_CDF\_RELEASE_>}

Inquires the release number of the CDF library that created the current CDF. Required arguments are as follows:

\begin{verbatim}
out: long *release

    Release number.
\end{verbatim}

The only required preselected object/state is the current CDF.

\texttt{<GET\_CDF\_VERSION_>}

Inquires the version number of the CDF library that created the current CDF. Required arguments are as follows:

\begin{verbatim}
out: long *version

    Version number.
\end{verbatim}

The only required preselected object/state is the current CDF.

\texttt{<GET\_DATATYPE\_SIZE_>}

Inquires the size (in bytes) of an element of the specified data type. Required arguments are as follows:

\begin{verbatim}
in: long dataType

    Data type.

out: long *numBytes

    Number of bytes per element.
\end{verbatim}

There are no required preselected objects/states.

\texttt{<GET\_gENTRY\_DATA_>}

Reads the gEntry data value from the current attribute at the current gEntry number (in the current CDF). Required arguments are as follows:

\begin{verbatim}
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.
\end{verbatim}

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

\textbf{NOTE:} Only use this operation on gAttributes. An error will occur if used on a vAttribute.

\texttt{<GET\_gENTRY\_DATATYPE_>}

Inquires the data type of the gEntry at the current gEntry number for the current attribute (in the current CDF). Required arguments are as follows:

\begin{verbatim}
out: long *dataType

    Data type. The data types are described in Section 4.5.
\end{verbatim}

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.
NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<GET_, gENTRY_NUMELEM_>
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). Required arguments are as follows:

out: long *numElements

Number of elements of the data type. For character 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.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<GET_, LIB_COPYRIGHT_>
Reads the Copyright notice of the CDF library being used. Required arguments are as follows:

out: char Copyright[CDF_COPYRIGHT_LEN+1]

CDF library Copyright text.

There are no required preselected objects/states.

<GET_, 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.

<GET_, 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.

<GET_, 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.

<GET_, 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.

<GET_rENTRY_DATA_>
Reads the rEntry data value from the current attribute at the current rEntry number (in the current CDF).
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.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_rENTRY_DATATYPE_>
Inquires the data type of the rEntry at the current rEntry number for the current attribute (in the current CDF).
Required arguments are as follows:

out: long *dataType

Data type. The data types are described in Section 4.5.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_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). Required arguments are as follows:

out: long *numElements

Number of elements of the data type. For character 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.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_rVAR_ALLOCATEDFROM_>
Inquires the next allocated record at or after a given record for the current rVariable (in the current CDF).
Required arguments are as follows:

in: long startRecord

The record number at which to begin searching for the next allocated record. If this record exists, it will be considered the next allocated record.

out: long *nextRecord
The number of the next allocated record.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_ALLOCATEDTO_>  
Inquires the last allocated record (before the next unallocated record) at or after a given record for the current rVariable (in the current CDF). Required arguments are as follows:

in:  long  startRecord  
The record number at which to begin searching for the last allocated record.

out:  long  *nextRecord  
The number of the last allocated record.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_BLOCKINGFACTOR_>37  
Inquires the blocking factor for the current rVariable (in the current CDF). Blocking factors are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

out:  long  *blockingFactor  
The blocking factor. A value of zero (0) indicates that the default blocking factor is being used.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_COMPRESSION_>  
Inquires the compression type/parameters of the current rVariable (in the current CDF). Required arguments are as follows:

out:  long  *cType  
The compression type. The types of compressions are described in Section 4.10.

out:  long  cParms[CDF_MAX_PARMS]  
The compression parameters. The compression parameters are described in Section 4.10.

out:  long  *cPct  
If compressed, the percentage of the uncompressed size of the rVariable's data values needed to store the compressed values.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,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

---

37 The item rVAR_BLOCKINGFACTOR was previously named rVAR_EXTENDRECS.
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.

<GET_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 4.5.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_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[CDF_MAX_DIMS]

Dimension variances. Each element of dimVarys receives the corresponding dimension variance. The variances are described in Section 4.9.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_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.

<GET_rVAR_MAXallocREC_>  
Inquires the maximum record number allocated for the current rVariable (in the current CDF). 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.

<GET_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.

<GET_,rVAR_NAME_>
Inquires the name of the current rVariable (in the current CDF). Required arguments are as follows:

out: char varName[CDF_VAR_NAME_LEN256+1

Name of the rVariable.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_numINDEXENTRIES_>
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. The Concepts chapter in the CDF User's Guide 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.

<GET_,rVAR_numINDEXLEVELS_>
Inquires the number of index levels for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: long *numLevels

Number of index levels.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_numINDEXRECORDS_>
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. The Concepts chapter in the CDF User's Guide 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.

<GET_,rVAR_NUMallocRECS_>
Inquires the number of records allocated for the current rVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes the allocation of variable records in a single-file CDF. Required arguments are as follows:

out: long *numRecords

Number of allocated records.
The required preselected objects/states are the current CDF and its current rVariable.

<GET_,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_LEN256 characters (excluding the NUL terminator).

out: long *varNum

The rVariable number.

The only required preselected object/state is the current CDF.

<GET_,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 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 data types.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_NUMRECS_>

Inquires the number of records written for the current rVariable (in the current CDF). This may not correspond to the maximum record written (see <GET_,rVAR_MAXREC_>) if the rVariable has sparse records. Required arguments are as follows:

out: long *numRecords

Number of records written.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_PADVALUE_>

Inquires the pad value of the current rVariable (in the current CDF). If a pad value has not been explicitly specified for the rVariable (see <PUT_,rVAR_PADVALUE_>), the informational status code NO_PADVALUE_SPECIFIED will be returned and the default pad value for the rVariable's data type will be placed in the pad value buffer provided. 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.

<GET_,rVAR_RECVARY_>

235
Inquires the record variance of the current rVariable (in the current CDF). Required arguments are as follows:

```c
out: long *recVary
```

Record variance. The variances are described in Section 4.9.

The required preselected objects/states are the current CDF and its current rVariable.

**<GET_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:

```c
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.

**<GET_rVAR_SPARSEARRAYS_>**

Inquires the sparse arrays type/parameters of the current rVariable (in the current CDF). Required arguments are as follows:

```c
out: long *sArraysType
```

The sparse arrays type. The types of sparse arrays are described in Section 4.11.2.

```c
out: long sArraysParms[CDF_MAX_PARMS]
```

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.2.

```c
out: long *sArraysPct
```

If sparse arrays, the percentage of the non-sparse size of the rVariable's data values needed to store the sparse values.

The required preselected objects/states are the current CDF and its current rVariable.

**<GET_rVAR_SPARSERECORDS_>**

Inquires the sparse records type of the current rVariable (in the current CDF). Required arguments are as follows:

```c
out: long *sRecordsType
```

The sparse records type. The types of sparse records are described in Section 4.11.1.

The required preselected objects/states are the current CDF and its current rVariable.

**<GET_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[CDF_MAX_DIMS]

Dimension sizes. Each element of dimSizes receives the corresponding dimension size.

The only required preselected object/state is the current CDF.

<GET_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 the <GET_rVAR_MAXREC_> operation. Required arguments are as follows:

out: long *maxRec

Maximum record number.

The only required preselected object/state is the current CDF.

<GET_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.

<GET_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.

out: 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 the Concepts chapter in the CDF User's Guide 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. 38

38 A Standard Interface CDFgetrVarsRecordDatabyNumbers provides the same functionality.
<GET_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[CDF_STATUSTEXT_LEN+1]

Text explaining the status code.

The only required preselected object/state is the current status code.

<GET_zENTRY_DATA_>
Reads the zEntry data value from the current attribute at the current zEntry number (in the current CDF). 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.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_zENTRY_DATATYPE_>
Inquires the data type of the zEntry at the current zEntry number for the current attribute (in the current CDF). Required arguments are as follows:

out: long *dataType

Data type. The data types are described in Section 4.5.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_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). Required arguments are as follows:

out: long *numElements

Number of elements of the data type. For character 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.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_zVAR_ALLOCATEDFROM_>
Inquires the next allocated record at or after a given record for the current zVariable (in the current CDF). Required arguments are as follows:

in: long startRecord
The record number at which to begin searching for the next allocated record. If this record exists, it will be considered the next allocated record.

out: long *nextRecord

The number of the next allocated record.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_ALLOCTEDTO_>
Inquires the last allocated record (before the next unallocated record) at or after a given record for the current zVariable (in the current CDF). Required arguments are as follows:

in: long startRecord

The record number at which to begin searching for the last allocated record.

out: long *nextRecord

The number of the last allocated record.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_BLOCKINGFACTO_>_³⁹
Inquires the blocking factor for the current zVariable (in the current CDF). Blocking factors are described in the Concepts chapter in the CDF User’s Guide. Required arguments are as follows:

out: long *blockingFactor

The blocking factor. A value of zero (0) indicates that the default blocking factor is being used.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_COMPRESSION_>
Inquires the compression type/parameters of the current zVariable (in the current CDF). Required arguments are as follows:

out: long *cType

The compression type. The types of compressions are described in Section 4.10.

out: long cParms[CDF_MAX_PARMS]

The compression parameters. The compression parameters are described in Section 4.10.

out: long *cPct

If compressed, the percentage of the uncompressed size of the zVariable's data values needed to store the compressed values.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_DATA_>

³⁹The item zVAR_BLOCKINGFACTOR was previously named zVAR_EXTENDRECS.
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.

<GET_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 4.5.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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[CDF_MAX_DIMS]

Dimension sizes. Each element of dimSizes receives the corresponding dimension size.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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[CDF_MAX_DIMS]

Dimension variances. Each element of dimVarys receives the corresponding dimension variance. The variances are described in Section 4.9.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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.
<GET_zVAR_MAXallocREC_>
Inquires the maximum record number allocated for the current zVariable (in the current CDF). 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.

<GET_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.

<GET_zVAR_NAME_>
Inquires the name of the current zVariable (in the current CDF). Required arguments are as follows:

out: char varName[CDF_VAR_NAME_LEN256+1]

Name of the zVariable.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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. The Concepts chapter in the CDF User's Guide 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.

<GET_zVAR_nINDEXLEVELS_>
Inquires the number of index levels for the current zVariable (in the current CDF). This only has significance for zVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: long *numLevels

Number of index levels.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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. The Concepts chapter in the CDF User's Guide 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.

<GET_zVAR_NUMallocRECS_>
Inquires the number of records allocated for the current zVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes the allocation of variable records in a single-file CDF. Required arguments are as follows:

out: long *numRecords

Number of allocated records.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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_LEN256 characters (excluding the NUL terminator).

out: long *varNum

The zVariable number.

The only required preselected object/state is the current CDF.

<GET_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.

<GET_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 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 data types.

The required preselected objects/states are the current CDF and its current zVariable.
<GET_<zVAR_NUMRECS_>>
Inquires the number of records written for the current zVariable (in the current CDF). This may not correspond to the maximum record written (see <GET_<zVAR_MAXREC_>>) if the zVariable has sparse records. Required arguments are as follows:

out: long *numRecords

Number of records written.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_<zVAR_PADVALUE_>>
Inquires the pad value of the current zVariable (in the current CDF). If a pad value has not been explicitly specified for the zVariable (see <PUT_<zVAR_PADVALUE_>), the informational status code NO_PADVALUE_SPECIFIED will be returned and the default pad value for the zVariable's data type will be placed in the pad value buffer provided. 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.

<GET_<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 4.9.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_<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.

<GET_<zVAR_SPARSEARRAYS_>>
Inquires the sparse arrays type/parameters of the current zVariable (in the current CDF). Required arguments are as follows:

out: long *sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 4.11.2.
out: long sArraysParms[CDF_MAX_PARMS]

The sparse arrays parameters.

out: long *sArraysPct

If sparse arrays, the percentage of the non-sparse size of the zVariable's data values needed to store the sparse values.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_SPARSERECORDS_>
Inquires the sparse records type of the current zVariable (in the current CDF). Required arguments are as follows:

out: long *sRecordsType

The sparse records type. The types of sparse records are described in Section 4.11.1.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_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 the <GET_zVAR_MAXREC_> operation. Required arguments are as follows:

out: long *maxRec

Maximum record number.

The only required preselected object/state is the current CDF.

<GET_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.

out: 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 the Concepts chapter in the CDF User's Guide 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, \(<\text{SELECT}_z\text{VARs RECNUMBER}_>\), 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 \(<\text{SELECT}_z\text{VAR RECNUMBER}_>\)). \(^{40}\)

\(<\text{NULL}_>\)
Marks the end of the argument list that is passed to An internal interface call. No other arguments are allowed after it.

\(<\text{OPEN}_\text{CDF}_>\)
Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

\[\text{in: char } *\text{CDFname}\]

File name 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 file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems).

UNIX: File names are case-sensitive.

\[\text{out: CDFid } *\text{id}\]

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

There are no required preselected objects/states.

\(<\text{PUT}_z\text{ATTR NAME}_>\)
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:

\[\text{in: char } *\text{attrName}\]

New attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters (excluding the NUL terminator).

The required preselected objects/states are the current CDF and its current attribute.

\(<\text{PUT}_z\text{ATTR SCOPE}_>\)
Respecifies the scope for the current attribute (in the current CDF). Required arguments are as follows:

\[\text{in: long } \text{scope}\]

New attribute scope. Specify one of the scopes described in Section 4.12.

The required preselected objects/states are the current CDF and its current attribute.

\(<\text{PUT}_z\text{CDF CHECKSUM}_>\)
Respecifies the checksum mode of the current CDF. Required arguments are as follows:

\(^{40}\) A Standard Interface CDFgetzVarsRecordDatabyNumbers provides the same functionality.
in: long checksum

The checksum mode to be used (NO_CHECKSUM or MD5_CHECKSUM). The checksum mode is described in Section 4.19.

The required preselected objects/states is the current CDF.

<PUT_CDF_COMPRESSION_>
Specifies the compression type/parameters for the current CDF. This refers to the compression of the CDF - not of any variables. Required arguments are as follows:

in: long cType

The compression type. The types of compressions are described in Section 4.10.

in: long cParms[]

The compression parameters. The compression parameters are described in Section 4.10.

The only required preselected object/state is the current CDF.

<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.

The only required preselected object/state is the current CDF.

<PUT_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 4.4.

The only required preselected object/state is the current CDF.

<PUT_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 4.8.

The only required preselected object/state is the current CDF.

<PUT_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. Required arguments are as follows:
in: long dataType

Data type of the gEntry. Specify one of the data types described in Section 4.5.

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 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.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<PUT_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 the Concepts chapter in the CDF User’s Guide. Required arguments are as follows:

in: long dataType

New data type of the gEntry. Specify one of the data types described in Section 4.5.

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.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<PUT_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. Required arguments are as follows:

in: long dataType

Data type of the rEntry. Specify one of the data types described in Section 4.5.

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 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.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<!--PUT_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 the Concepts chapter in the CDF User’s Guide. Required arguments are as follows:

- **in:** long dataType
  
  New data type of the rEntry. Specify one of the data types described in Section 4.5.

- **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.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<!--PUT_rVAR_ALLOCATEBLOCK-->

Specifies a range of records to allocate for the current rVariable (in the current CDF). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

- **in:** long firstRecord
  
  The first record number to allocate.

- **in:** long lastRecord
  
  The last record number to allocate.

The required preselected objects/states are the current CDF and its current rVariable.

<!--PUT_rVAR_ALLOCATERECS-->

Specifies the number of records to allocate for the current rVariable (in the current CDF). The records are allocated beginning at record number 0 (zero). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

- **in:** long nRecords
  
  Number of records to allocate.

The required preselected objects/states are the current CDF and its current rVariable.

<!--PUT_rVAR_BLOCKINGFACTOR-->\(^{41}\)

\(^{41}\) The item rVAR_BLOCKINGFACTOR was previously named rVAR_EXTENDRECS.
Specifies the blocking factor for the current rVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes a variable's blocking factor. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs. Required arguments are as follows:

```plaintext
in: long blockingFactor
```

The blocking factor. A value of zero (0) indicates that the default blocking factor should be used.

The required preselected objects/states are the current CDF and its current rVariable.

**<PUT_,rVAR_COMPRESSION>_**

Specifies the compression type/parameters for the current rVariable (in current CDF). Required arguments are as follows:

```plaintext
in: long cType
```

The compression type. The types of compressions are described in Section 4.10.

```plaintext
in: long cParms[]
```

The compression parameters. The compression parameters are described in Section 4.10.

The required preselected objects/states are the current CDF and its current rVariable.

**<PUT_,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:

```plaintext
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.

**<PUT_,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 the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

```plaintext
in: long dataType
```

New data type. Specify one of the data types described in Section 4.5.

```plaintext
in: long numElements
```

Number of elements of the data type at each value. For character 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 data types this must be one (1) - arrays of values are not allowed for non-character data types.

The required preselected objects/states are the current CDF and its current rVariable.

**<PUT_,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. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_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.

<PUT_rVAR_INITIALRECS_>

Specifies the number of records to initially write to the current rVariable (in the current CDF). The records are written beginning at record number 0 (zero). 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 the Concepts chapter in the CDF User’s Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: long nRecords

Number of records to write.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_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_LEN256 characters (excluding the NUL terminator).

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_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). The Concepts chapter in the CDF User's Guide 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.

<PUT_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 4.9.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_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.

<PUT_rVAR_SPARSEARRAYS_>
Specifies the sparse arrays type/parameters for the current rVariable (in the current CDF). Required arguments are as follows:

in: long sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 4.11.2.

in: long sArraysParms[]

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.2.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_rVAR_SPARSERECORDS_>
Specifies the sparse records type for the current rVariable (in the current CDF). Required arguments are as follows:

in: long sRecordsType

The sparse records type. The types of sparse records are described in Section 4.11.1.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_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 struct may not be contiguous). See the Concepts chapter in the CDF User's Guide 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.  

<PUT_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. Required arguments are as follows:

in: long dataType

Data type of the zEntry. Specify one of the data types described in Section 4.5.

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 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.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<PUT_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 the Concepts chapter in the CDF User’s Guide. Required arguments are as follows:

A Standard Interface CDFputrVarsRecordDatabyNumbers provides the same functionality.
in: long dataType

New data type of the zEntry. Specify one of the data types described in Section 4.5.

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.

**NOTE:** Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<PUT_,zVAR_ALLOCATEBLOCK_>
Specifies a range of records to allocate for the current zVariable (in the current CDF). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: long firstRecord

The first record number to allocate.

in: long lastRecord

The last record number to allocate.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_ALLOCATERECS_>
Specifies the number of records to allocate for the current zVariable (in the current CDF). The records are allocated beginning at record number 0 (zero). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: long nRecords

Number of records to allocate.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_BLOCKINGFACTOR_>\(^{43}\)
Specifies the blocking factor for the current zVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes a variable's blocking factor. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs. Required arguments are as follows:

in: long blockingFactor

The blocking factor. A value of zero (0) indicates that the default blocking factor should be used.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_COMPRESSION_>
Specifies the compression type/parameters for the current zVariable (in current CDF). Required arguments are as follows:

\(^{43}\) The item zVAR_BLOCKINGFACTOR was previously named zVAR_EXTENDRECS.
in: long cType

The compression type. The types of compressions are described in Section 4.10.

in: long cParms[]

The compression parameters. The compression parameters are described in Section 4.10.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,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.

<PUT_,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 the Concepts chapter in the CDF User's Guide) 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 4.5.

in: long numElements

Number of elements of the data type at each value. For character 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 data types this must be one (1) - arrays of values are not allowed for non-character data types.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,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. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_INITIALRECS_>
 Specifies the number of records to initially write to the current zVariable (in the current CDF). The records are written beginning at record number 0 (zero). 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 the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

   in: long nRecords

   Number of records to write.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_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.

<PUT_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_LEN256 characters (excluding the NUL terminator).

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_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). The Concepts chapter in the CDF User's Guide 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.

<PUT_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 4.9.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_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.

<PUT_zVAR_SPARSEARRAYS_

Specifies the sparse arrays type/parameters for the current zVariable (in the current CDF). Required arguments are as follows:

in: long sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 4.11.2.

in: long sArraysParms[]

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.2.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_zVAR_SPARSERECORDS_

Specifies the sparse records type for the current zVariable (in the current CDF). Required arguments are as follows:

in: long sRecordsType

The sparse records type. The types of sparse records are described in Section 4.11.1.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_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.

```c
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 the Concepts chapter in the CDF User's Guide 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_>`.  

```c
ATT >
```

Explicitly selects the current attribute (in the current CDF) by number. Required arguments are as follows:

```c
long attrNum
```

Attribute number.

The only required preselected object/state is the current CDF.

```c
ATTR_NAME_>
```

Explicitly selects the current attribute (in the current CDF) by name. **NOTE:** Selecting the current attribute by number (see `<SELECT_.ATTR_>`) is more efficient. Required arguments are as follows:

```c
char *attrName
```

Attribute name. This may be at most `CDF_ATTR_NAME_LEN256` characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

```c
CDF_>
```

Explicitly selects the current CDF. Required arguments are as follows:

```c
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.

```c
CDF_CACHE_SIZE_>
```

Selects the number of cache buffers to be used for the dotCDF file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

```c
long numBuffers
```

A Standard Interface CDFputzVarsRecordDatabyNumbers provides the same functionality.
The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

<SELECT_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 4.7.

The only required preselected object/state is the current CDF.

<SELECT_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 4.15.

The only required preselected object/state is the current CDF.

<SELECT_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 4.13.

The only required preselected object/state is the current CDF.

<SELECT_CDF_SCRATCHDIR_>
Selects a directory to be used for scratch files (by the CDF library) for the current CDF. The Concepts chapter in the CDF User’s Guide describes how the CDF library uses scratch files. This scratch directory will override the directory specified by the CDF$TMP logical name (on OpenVMS systems) or CDF TMP environment variable (on UNIX and MS-DOS systems). Required arguments are as follows:

    in:  char  *scratchDir

    The directory to be used for scratch files. The length of this directory specification is limited only by the operating system being used.

The only required preselected object/state is the current CDF.

<SELECT_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.

<SELECT_CDF_zMODE_>
Selects a zMode (for the current CDF). Required arguments are as follows:
The zMode. Specify one of the zModes described in Section 4.14.

The only required preselected object/state is the current CDF.

<SELECT_COMPRESS_CACHESIZE_>

Selects the number of cache buffers to be used for the compression scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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.

<SELECT_gENTRY_>

Selects the current gEntry number for all gAttributes in the current CDF. Required arguments are as follows:

in: long entryNum

gEntry number.

The only required preselected object/state is the current CDF.

<SELECT_rENTRY_>

Selects the current rEntry number for all vAttributes in the current CDF. Required arguments are as follows:

in: long entryNum

rEntry number.

The only required preselected object/state is the current CDF.

<SELECT_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: Selecting the current rEntry by number (see <SELECT_rENTRY_>) is more efficient. Required arguments are as follows:

in: char *varName

rVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<SELECT_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.

<SELECT_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. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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.

<SELECT_rVAR_NAME>
Explicitly selects the current rVariable (in the current CDF) by name. **NOTE:** Selecting the current rVariable by number (see <SELECT_rVAR_>) is more efficient. Required arguments are as follows:

in: char *varName

rVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<SELECT_rVAR_reservePercentage>
Selects the reserve percentage to be used for the current rVariable (in the current CDF). This operation is only applicable to compressed rVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

in: long percent

The reserve percentage.

The required preselected objects/states are the current CDF and its current rVariable.

<SELECT_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. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but must be present).

The required preselected objects/states are the current CDF and its current rVariable.

<SELECT_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. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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.

<SELECT_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. Each element of counts specifies the corresponding dimension count.

The only required preselected object/state is the current CDF.

<SELECT_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. Each element of indices specifies the corresponding dimension index.

The only required preselected object/state is the current CDF.

<SELECT_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. Each element of intervals specifies the corresponding dimension interval.

The only required preselected object/state is the current CDF.

<SELECT_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.

<SELECT_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.

<SELECT_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.

<SELECT_STAGE CACHESIZE >
Selects the number of cache buffers to be used for the staging scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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.

<SELECT_zENTRY >
Selects the current zEntry number for all vAttributes in the current CDF. Required arguments are as follows:

in: long entryNum

zEntry number.

The only required preselected object/state is the current CDF.

<SELECT_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:** Selecting the current zEntry by number (see <SELECT_zENTRY >) is more efficient. Required arguments are as follows:

in: char *varName

zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<SELECT_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.

<SELECT_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. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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.

<SELECT_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. Each element of counts specifies the corresponding dimension count.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_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. Each element of indices specifies the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_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. Each element of intervals specifies the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_zVAR_NAME_>
Explicitly selects the current zVariable (in the current CDF) by name. **NOTE:** Selecting the current zVariable by number (see <SELECT_zVAR_>) is more efficient. Required arguments are as follows:

in: char *varName

zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

The only required preselected object/state is the current CDF.

<SELECT_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.

<SELECT_zVAR_RECINTERVAL_>

263
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.

`<SELECT_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.

`<SELECT_zVAR_RESERVEPERCENT_>`

Selects the reserve percentage to be used for the current zVariable (in the current CDF). This operation is only applicable to compressed zVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

`in: long percent`

The reserve percentage.

The required preselected objects/states are the current CDF and its current zVariable.

`<SELECT_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. Each element of indices specifies the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but must be present).

The required preselected objects/states are the current CDF and its current zVariable.

`<SELECT_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. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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.
SELECT_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 the Concepts chapter in the CDF User’s Guide). Required arguments are as follows:

in: long recNum

Record number.

The only required preselected object/state is the current CDF.

7.7 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_).

7.7.1 rVariable Creation

In this example an rVariable will be created with a pad value being specified; initial records will be written; and the rVariable's blocking factor 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.

```c
#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_BLOCKINGFACTOR__, (long) 50,
NULL_);
if (status != CDF_OK) UserStatusHandler (status);
```

7.7.2 zVariable Creation (Character Data Type)
In this example a zVariable with a character data type will be created with a pad value being specified. It is assumed that the current CDF has already been selected.

```c
#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. */
static 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);
```

### 7.7.3 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.

```c
#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", "Station", CDF_CHAR, numElems, numDims,
                dimSizes, NOVARY, dimVarys, &varNum,
                PUT_, rVAR_VALUE_, values,
                NULL_);
```
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 */
    .
}

7.7.4  Attribute Renaming

In this example the attribute named Tmp will be renamed to 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);

7.7.5  Sequential Access

In this example the values for a zVariable will be averaged. The values will be read using the sequential access method
(see the Concepts chapter in the CDF User's Guide). 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.
7.7.6 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, 
                "Latitud

#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;


7.7.6 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

#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;


7.7.6 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, 
                "Latitud

#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;


7.7.6 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.
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);

7.7.7 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 the Concepts chapter in the CDF User's Guide). 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);
}

269
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.

### 7.8 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:

```c
#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. 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. 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.

### 7.9 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:

```c
#define CDFcreate(CDFname,numDims,dimSizes,encoding,majority,id) \  CDFlib(CREATE_, CDF_, CDFname, numDims, dimSizes, id, \  PUT__, CDF_ENCODING_, encoding, \  CDF_MAJORITY_, majority, \  NULL_)
```

These macros are defined in cdf.h. Where your application calls CDFcreate, the C compiler (preprocessor) expands the macro into the corresponding call to CDFlib.

---

45 Fortran programmers can get away with doing something like this because everything is passed by reference.
The flexibility of CDFlib allows you to define your own custom CDF functions using C macros. For instance, a function that returns the format of a CDF could be defined as follows:

```c
#define  CDFinquireFormat(id,format) \  
                  CDFlib (SELECT_, CDF_, id, \  
                  GET_, CDF_FORMAT_, format, \  
                  NULL_) 
```

Your application would call the function as follows:

```c
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 8

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 A explains each status code. When the status code returned from a CDF function is tested, the following rules apply.

- \( \text{status} > \text{CDF\_OK} \) Indicates successful completion but some additional information is provided. These are informational codes.
- \( \text{status} = \text{CDF\_OK} \) Indicates successful completion.
- \( \text{CDF\_WARN} < \text{status} < \text{CDF\_OK} \) Indicates that the function completed but probably not as expected. These are warning codes.
- \( \text{status} < \text{CDF\_WARN} \) 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.

```c
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.

```c
#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 encodes in a text string an explanation of a given status code.
Chapter 9

9 EPOCH Utility Routines

Several functions exist that compute, decompose, parse, and encode CDF_EPOCH and CDF_EPOCH16 values. These functions may be called by applications using the CDF_EPOCH and CDF_EPOCH16 data types and are included in the CDF library. Function prototypes for these functions may be found in the include file cdf.h. The Concepts chapter in the CDF User's Guide describes EPOCH values. The date/time components for CDF_EPOCH and CDF_EPOCH16 are UTC-based, without leap seconds.

The CDF_EPOCH and CDF_EPOCH16 data types are used to store time values referenced from a particular epoch. For CDF that epoch values for CDF_EPOCH and CDF_EPOCH16 are milliseconds from 01-Jan-0000 00:00:00.000 and pico-seconds from 01-Jan-0000 00:00:00.000.000.000.000, respectively.

9.1 computeEPOCH

computeEPOCH calculates a CDF_EPOCH value given the individual components. If an illegal component is detected, the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double computeEPOCH(   /* out -- CDF_EPOCH value returned. */
    long year,         /* in -- Year (AD). */
    long month,        /* in -- Month */
    long day,          /* in -- Day */
    long hour,         /* in -- Hour */
    long minute,       /* in -- Minute */
    long second,       /* in -- Second */
    long msec);        /* in -- Millisecond */
```

NOTE: Previously, fields for month, day, hour, minute, second and msec should have a valid ranges, mainly 1-12 for month, 1-31 for day, 0-23 for hour, 0-59 for minute and second, and 0-999 for msec. However, there are two variations on how computeEPOCH can be used. The month argument is allowed to be 0 (zero), in which case, the day argument is assumed to be the day of the year (DOY) having a range of 1 through 366. Also, if the hour, minute, and second arguments are all 0s (zero), then the msec argument is assumed to be the millisecond of the day, having a range of 0 through 86400000. The modified computeEPOCH, since the CDF V3.3.1, allows month, day, hour minute, second and msec to be any values, even negative ones, without range checking as long as the cumulative date is after 0AD. Any cumulative date before 0AD will cause this function to return ILLEGAL_EPOCH_VALUE (–1.0) By not checking the range of dta fields, the epoch will be computed from any given values for month, day, hour, etc. For example, the epoch can be computed by passing a Unix-time (seconds from 1970-1-1) in a set of arguments of “1970, 1, 1, 0, 0,
unix-time, 0". While the second field is allowed to have a value of 60 (or greater), the CDF epoch still does not support of leap second. An input of 60 for the second field will automatically be interpreted as 0 (zero) second in the following minute. If the month field is 0, the day field is still considered as DOY. If the day field is 0, the date will fall back to the last day of the previous month, e.g., a date of 2010-2-0 becoming 2010-1-31. The following table shows how the year, month and day components of the epoch will be interpreted by the following EPOCHbreakdown function when the month and/or day field is passed in with 0 or negative value to computeEPOCH function.

<table>
<thead>
<tr>
<th>Year</th>
<th>Month</th>
<th>Day</th>
<th>→</th>
<th>Year</th>
<th>Month</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>0</td>
<td>0</td>
<td>→</td>
<td>2009</td>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>2010</td>
<td>-1</td>
<td>0</td>
<td>→</td>
<td>2009</td>
<td>11</td>
<td>30</td>
</tr>
<tr>
<td>2010</td>
<td>0</td>
<td>1</td>
<td>→</td>
<td>2010</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2010</td>
<td>1</td>
<td>0</td>
<td>→</td>
<td>2009</td>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>2010</td>
<td>0</td>
<td>-1</td>
<td>→</td>
<td>2009</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>2010</td>
<td>-1</td>
<td>-1</td>
<td>→</td>
<td>2009</td>
<td>11</td>
<td>29</td>
</tr>
</tbody>
</table>

9.2 EPOCHbreakdown

EPOCHbreakdown decomposes a CDF_EPOCH value into the individual components.

```c
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). */
```

9.3 encodeEPOCH

encodeEPOCH encodes a CDF_EPOCH value into the standard date/time character string. The format of the string is `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, 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).

```c
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.
9.4 **encodeEPOCH1**

`encodeEPOCH1` encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is `yyyyymmdd.tttttt`, where `yyyy` is the year, `mm` is the month (1-12), `dd` is the day of the month (1-31), and `tttttt` is the fraction of the day (e.g., 500000 is 12 o'clock noon).

```c
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.

9.5 **encodeEPOCH2**

`encodeEPOCH2` encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is `yyyyymmdhhmmss` where `yyyy` is the year, `mmd` 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).

```c
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.

9.6 **encodeEPOCH3**

`encodeEPOCH3` encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is `yyyy-mo-ddThh:mm:ss.cccZ` where `yyyy` is the year, `mo` 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), `ss` is the second (0-59), and `ccc` is the millisecond (0-999).

```c
void encodeEPOCH3(
    double epoch;    /* in -- The CDF_EPOCH value. */
    char epString[EPOCH3_STRING_LEN+1];    /* out -- The alternate date/time character string. */
)
```

`EPOCH3_STRING_LEN` is defined in cdf.h.

9.7 **encodeEPOCH4**
encodeEPOCH4 encodes a CDF_EPOCH value into an alternate, ISO 8601 date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.ccc where yyyy is the year, mo 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), ss is the second (0-59), and ccc is the millisecond (0-999).

```c
void encodeEPOCH4(
    double epoch;
    char epString[EPOCH4_STRING_LEN+1];
    /* in -- The CDF_EPOCH value. */
    /* out -- The ISO 8601 date/time character string. */
)

EPOCH4_STRING_LEN is defined in cdf.h.
```

### 9.8 encodeEPOCHx

encodeEPOCHx encodes a CDF_EPOCH value into a custom date/time character string. The format of the encoded string is specified by a format string.

```c
void encodeEPOCHx(
    double epoch;
    char format[EPOCHx_FORMAT_MAX];
    char encoded[EPOCHx_STRING_MAX];
    /* in -- The CDF_EPOCH value. */
    /* in -- The format string. */
    /* out -- The custom date/time character string. */
)
```

The format string consists of EPOCH components, which are encoded, and text that is simply copied to the encoded custom string. Components are enclosed in angle brackets and consist of a component token and an optional width. The syntax of a component is: `<token[.width]>`. If the optional width contains a leading zero, then the component will be encoded with leading zeroes (rather than leading blanks).

The supported component tokens and their default widths are as follows.

<table>
<thead>
<tr>
<th>Token</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom</td>
<td>Day of month (1-31)</td>
<td><code>&lt;dom.0&gt;</code></td>
</tr>
<tr>
<td>doy</td>
<td>Day of year (001-366)</td>
<td><code>&lt;doy.03&gt;</code></td>
</tr>
<tr>
<td>month</td>
<td>Month (’Jan’,’Feb’,...,’Dec’)</td>
<td><code>&lt;month&gt;</code></td>
</tr>
<tr>
<td>mm</td>
<td>Month (1,2,...,12)</td>
<td><code>&lt;mm.0&gt;</code></td>
</tr>
<tr>
<td>year</td>
<td>Year (4-digit)</td>
<td><code>&lt;year.04&gt;</code></td>
</tr>
<tr>
<td>yr</td>
<td>Year (2-digit)</td>
<td><code>&lt;yr.02&gt;</code></td>
</tr>
<tr>
<td>hour</td>
<td>Hour (00-23)</td>
<td><code>&lt;hour.02&gt;</code></td>
</tr>
<tr>
<td>min</td>
<td>Minute (00-59)</td>
<td><code>&lt;min.02&gt;</code></td>
</tr>
<tr>
<td>sec</td>
<td>Second (00-59)</td>
<td><code>&lt;sec.02&gt;</code></td>
</tr>
<tr>
<td>fos</td>
<td>Fraction of second.</td>
<td><code>&lt;fos.3&gt;</code></td>
</tr>
<tr>
<td>fod</td>
<td>Fraction of day.</td>
<td><code>&lt;fod.8&gt;</code></td>
</tr>
</tbody>
</table>

Note that a width of zero indicates that as many digits as necessary should be used to encoded the component. The `<month>` component is always encoded with three characters. The `<fos>` and `<fod>` components are always encoded with leading zeroes.

If a left angle bracket is desired in the encoded string, then simply specify two left angle brackets (`<<`) in the format string (character stuffing).

For example, the format string used to encode the standard EPOCH date/time character string (see Section 9.3) would be.

```c
<dom.02>:<month>:<year>:<hour>:<min>:<sec>:<fos>
```
EPOCHx_FORMAT_LEN and EPOCHx_STRING_MAX are defined in cdf.h.

9.9 parseEPOCH

parseEPOCH parses a standard date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH function described in Section 9.3. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH(
    char epString[EPOCH_STRING_LEN+1]);
/* in -- The standard date/time character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

9.10 parseEPOCH1

parseEPOCH1 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH1 function described in Section 9.4. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH1(
    char epString[EPOCH1_STRING_LEN+1]);
/* in -- The alternate date/time character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

9.11 parseEPOCH2

parseEPOCH2 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH2 function described in Section 9.5. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH2(
    char epString[EPOCH2_STRING_LEN+1]);
/* in -- The alternate date/time character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

9.12 parseEPOCH3

parseEPOCH3 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH3 function described in Section 9.6. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.
9.13 parseEPOCH4

parseEPOCH4 parses an alternate, ISO 8601 date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH4 function described in Section 9.7. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH4(
    char epString[EPOCH4_STRING_LEN+1]);
```

EPOCH4_STRING_LEN is defined in cdf.h.

9.14 computeEPOCH16

computeEPOCH16 calculates a CDF_EPOCH16 value given the individual components. If an illegal component is detected, the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double computeEPOCH16(
    long year, /* in -- Year (AD, e.g., 1994). */
    long month, /* in -- Month. */
    long day, /* in -- Day. */
    long hour, /* in -- Hour. */
    long minute, /* in -- Minute. */
    long second, /* in -- Second. */
    long msec, /* in -- Millisecond. */
    long microsec, /* in -- Microsecond. */
    long nanosec, /* in -- Nanosecond. */
    long picosec, /* in -- Picosecond. */
    double epoch[2]); /* out -- CDF_EPOCH16 value returned */
```

Similar to computeEPOCH, this function no longer performs range checks for each individual component as long as the cumulative date is after 0AD.

9.15 EPOCH16breakdown

EPOCH16breakdown decomposes a CDF_EPOCH16 value into the individual components.

```c
void EPOCH16breakdown(
    double epoch[2], /* in -- The CDF_EPOCH16 value. */
```
9.16 encodeEPOCH16

encodeEPOCH16 encodes a CDF_EPOCH16 value into the standard date/time character string. The format of the string is `dd-mmm-yyyy hh:mm:ss.mmm:uuu:nnn:ppp` 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, `hh` is the hour (0-23), `mm` is the minute (0-59), `ss` is the second (0-59), `mmm` is the millisecond (0-999), `uuu` is the microsecond (0-999), `nnn` is the nanosecond (0-999), and `ppp` is the picosecond (0-999).

```c
void encodeEPOCH16(
    double epoch[2]; /* in -- The CDF_EPOCH16 value. */
    char epString[EPOCH16_STRING_LEN+1]); /* out -- The date/time character string. */
```

EPOCH16_STRING_LEN is defined in cdf.h.

9.17 encodeEPOCH16_1

encodeEPOCH16_1 encodes a CDF_EPOCH16 value into an alternate date/time character string. The format of the string is `yyyyymmdd.ttttttttttttt`, where `yyyy` is the year, `mm` is the month (1-12), `dd` is the day of the month (1-31), and `tttttttttttt` is the fraction of the day (e.g., 500000000000000 is 12 o'clock noon).

```c
void encodeEPOCH16_1(
    double epoch[2]; /* in -- The CDF_EPOCH16 value. */
    char epString[EPOCH16_1 STRING_LEN +1]); /* out -- The date/time character string. */
```

EPOCH16_1 STRING_LEN is defined in cdf.h.

9.18 encodeEPOCH16_2

encodeEPOCH16_2 encodes a CDF_EPOCH16 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).

```c
void encodeEPOCH16_2(
```
double epoch[2]; /* in -- The CDF_EPOCH16 value. */
char epString[EPOCH16_2_STRING_LEN+1]; /* out -- The date/time character string. */

EPOCH16_2_STRING_LEN is defined in cdf.h.

**9.19 encodeEPOCH16_3**

encodeEPOCH16_3 encodes a CDF_EPOCH16 value into an alternate date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.mmmuuunnnpppZ where yyyy is the year, mo 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), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```c
void encodeEPOCH16_3(
    double epoch[2]; /* in -- The CDF_EPOCH16 value. */
    char epString[EPOCH16_3_STRING_LEN+1]; /* out -- The alternate date/time character string. */
)
```

EPOCH16_3_STRING_LEN is defined in cdf.h.

**9.20 encodeEPOCH16_4**

encodeEPOCH16_4 encodes a CDF_EPOCH16 value into an alternate, ISO 8601 date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.mmmuuunnnppp where yyyy is the year, mo 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), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```c
void encodeEPOCH16_4(
    double epoch[2]; /* in -- The CDF_EPOCH16 value. */
    char epString[EPOCH16_4_STRING_LEN+1]; /* out -- The ISO 8601 date/time character string. */
)
```

EPOCH16_4_STRING_LEN is defined in cdf.h.

**9.21 encodeEPOCH16_x**

encodeEPOCH16_x encodes a CDF_EPOCH16 value into a custom date/time character string. The format of the encoded string is specified by a format string.

```c
void encodeEPOCH16_x(
    double epoch[2]; /* in -- The CDF_EPOCH16 value. */
    char format[EPOCHx_FORMAT_MAX]; /* in -- The format string. */
    char encoded[EPOCHx_STRING_MAX]; /* out -- The date/time character string. */
)
```

The format string consists of EPOCH components, which are encoded, and text that is simply copied to the encoded custom string. Components are enclosed in angle brackets and consist of a component token and an optional width.
The syntax of a component is: `<token[.width]>`. If the optional width contains a leading zero, then the component will be encoded with leading zeroes (rather than leading blanks).

The supported component tokens and their default widths are as follows.

<table>
<thead>
<tr>
<th>Token</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom</td>
<td>Day of month (1-31)</td>
<td><code>&lt;dom.0&gt;</code></td>
</tr>
<tr>
<td>doy</td>
<td>Day of year (001-366)</td>
<td><code>&lt;doy.03&gt;</code></td>
</tr>
<tr>
<td>month</td>
<td>Month ('Jan', 'Feb', 'Dec')</td>
<td><code>&lt;month&gt;</code></td>
</tr>
<tr>
<td>mm</td>
<td>Month (1, 2, ..., 12)</td>
<td><code>&lt;mm.0&gt;</code></td>
</tr>
<tr>
<td>year</td>
<td>Year (4-digit)</td>
<td><code>&lt;year.04&gt;</code></td>
</tr>
<tr>
<td>yr</td>
<td>Year (2-digit)</td>
<td><code>&lt;yr.02&gt;</code></td>
</tr>
<tr>
<td>hour</td>
<td>Hour (00-23)</td>
<td><code>&lt;hour.02&gt;</code></td>
</tr>
<tr>
<td>min</td>
<td>Minute (00-59)</td>
<td><code>&lt;min.02&gt;</code></td>
</tr>
<tr>
<td>sec</td>
<td>Second (00-59)</td>
<td><code>&lt;sec.02&gt;</code></td>
</tr>
<tr>
<td>msc</td>
<td>Millisecond (000-999)</td>
<td><code>&lt;msc.3&gt;</code></td>
</tr>
<tr>
<td>usc</td>
<td>Microsecond (000-999)</td>
<td><code>&lt;usc.3&gt;</code></td>
</tr>
<tr>
<td>nsc</td>
<td>Nanosecond (000-999)</td>
<td><code>&lt;nsc.3&gt;</code></td>
</tr>
<tr>
<td>psc</td>
<td>Picosecond (000-999)</td>
<td><code>&lt;psc.3&gt;</code></td>
</tr>
<tr>
<td>fos</td>
<td>Fraction of second.</td>
<td><code>&lt;fos.12&gt;</code></td>
</tr>
<tr>
<td>fod</td>
<td>Fraction of day.</td>
<td><code>&lt;fod.8&gt;</code></td>
</tr>
</tbody>
</table>

Note that a width of zero indicates that as many digits as necessary should be used to encode the component. The `<month>` component is always encoded with three characters. The `<fos>` and `<fod>` components are always encoded with leading zeroes.

If a left angle bracket is desired in the encoded string, then simply specify two left angle brackets (`<<`) in the format string (character stuffing).

For example, the format string used to encode the standard EPOCH date/time character string would be...

```plaintext
<dom.02><month><year><hour>:<min>:<sec><msc><usc><nsc><psc><fos>
```

EPOCHx_FORMAT_LEN and EPOCHx_STRING_MAX are defined in cdf.h.

### 9.22 parseEPOCH16

parseEPOCH16 parses a standard date/time character string and returns a CDF_EPOCH16 value. The format of the string is that produced by the encodeEPOCH16 function. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH16(
    char epString[EPOCH16_STRING_LEN+1],
    double epoch[2]);

/* out -- The status code returned. */
/* in -- The date/time character string. */
/* out -- The CDF_EPOCH16 value returned */
```

EPOCH16_STRING_LEN is defined in cdf.h.
9.23 parseEPOCH16_1

parseEPOCH16_1 parses an alternate date/time character string and returns a CDF_EPOCH16 value. The format of the string is that produced by the encodeEPOCH16_1 function. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH16_1(
    char epString[EPOCH16_1_STRING_LEN+1],
    double epoch[2]);
/* out -- The status code returned. */
/* in -- The date/time character string. */
/* out -- The CDF_EPOCH16 value returned */
```

EPOCH16_1_STRING_LEN is defined in cdf.h.

9.24 parseEPOCH16_2

parseEPOCH16_2 parses an alternate date/time character string and returns a CDF_EPOCH16 value. The format of the string is that produced by the encodeEPOCH16_2 function. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH16_2(
    char epString[EPOCH16_2_STRING_LEN +1],
    double epoch[2]);
/* out -- The status code returned. */
/* in -- The date/time character string. */
/* out -- The CDF_EPOCH16 value returned */
```

EPOCH16_2_STRING_LEN is defined in cdf.h.

9.25 parseEPOCH16_3

parseEPOCH16_3 parses an alternate date/time character string and returns a CDF_EPOCH16 value. The format of the string is that produced by the encodeEPOCH16_3 function. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH16_3(
    char epString[EPOCH16_3_STRING_LEN +1],
    double epoch[2]);
/* out -- The status code returned. */
/* in -- The date/time character string. */
/* out -- The CDF_EPOCH16 value returned */
```

EPOCH16_3_STRING_LEN is defined in cdf.h.

9.26 parseEPOCH16_4

parseEPOCH16_4 parses an alternate, ISO 8601 date/time character string and returns a CDF_EPOCH16 value. The format of the string is that produced by the encodeEPOCH16_4 function. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```c
double parseEPOCH16_4(
    /* out -- The status code returned. */
```
char epString[EPOCH16_4_STRING_LEN +1],
double epoch[2]);

/* in -- The ISO 8601 date/time string. */
/* out -- The CDF_EPOCH16 value returned */

EPOCH16_4_STRING_LEN is defined in cdf.h.
10 TT2000 Utility Routines

Several functions exist that compute, decompose, parse, and encode CDF_TIME_TT2000 values. These functions may be called by applications using the CDF_TIME_TT2000 data type and are included in the CDF library. Function prototypes for these functions may be found in the include file cdf.h. The Concepts chapter in the CDF User's Guide describes TT2000 values. The date/time components for CDF_TIME_TT2000 are UTC-based, with leap seconds.

The CDF_TIME_TT2000 data types are used to store time values referenced from J2000 (2000-01-01T12:00:00.000000000), the Terrestrial Time (TT). For CDF, values in CDF_TIME_TT2000 are nanoseconds from J2000 with leap seconds included. TT2000 data can cover years between 1707 and 2292.

10.1 CDF_TT2000_from.UTC_parts

CDF_TT2000_from.UTC_parts calculates a CDF_TIME_TT2000 value, given the individual UTC-based time components. If an illegal component is detected, the value returned will be ILLEGAL_TT2000_VALUE.

The variable argument form:

    long long CDF_TT2000_from.UTC_parts(    /* out -- CDF_TIME_TT2000 value returned. */
        double year,                     /* in -- Year (AD). */
        double month,                   /* in -- Month */
        double day,                     /* in -- Day */
        ....
        double TT2000END);

The full form:

    long long CDF_TT2000_from.UTC_parts(    /* out -- CDF_TIME_TT2000 value returned. */
        double year,                     /* in -- Year (AD). */
        double month,                   /* in -- Month */
        double day,                     /* in -- Day */
        double hour,                    /* in -- Hour. */
        double minute,                  /* in -- Minute */
        double second,                  /* in -- Second */
        double msec,                    /* in -- Millisecond */
        double usec,                    /* in -- Microsecond */
        double nsec);                   /* in -- Nanosecond */

This function is also aliased as computeTT2000 for short. This function accepts variable number of arguments after the first three components of year, month and day. It allows a full argument list of nine (9) fields: year, month, day, hour, minute, second, millisecond, microsecond and nanosecond. If less than full arguments is passed in, a predefined TT2000END has to be appended to signify the end of argument list. Without it, a unexpected value might be returned.
**Note:** Note: Even all components are defined as double, to simplify the computation, this function only allows the very last argument to have a non-zero fractional part. The followings are some samples.

For three date/time arguments (sub-day),

```c
float tt2000 = CDF_TT2000_from_UTC_parts(2010.0, 10.0, 12.5, TT2000END);
```

For four date/time arguments (sub-hour),

```c
float tt2000 = CDF_TT2000_from_UTC_parts(2010.0, 10.0, 12.0, 12.5, TT2000END);
```

For five date/time arguments (sub-minute),

```c
float tt2000 = CDF_TT2000_from_UTC_parts(2010.0, 10.0, 12.0, 12.0, 30.5, TT2000END);
```

For six date/time arguments (sub-second),

```c
float tt2000 = CDF_TT2000_from_UTC_parts(2010.0, 10.0, 12.0, 12.0, 30.0, 30.5, TT2000END);
```

For the complete argument list:

```c
float tt2000 = CDF_TT2000_from_UTC_parts(2010.0, 10.0, 12.0, 1.0, 2.0, 3.0, 111.0, 222.0, 333.5);
```

This call is not allowed,

```c
float tt2000 = CDF_TT2000_from_UTC_parts(2010.0, 10.0, 12.5, 12.5, TT2000END);
```

Any invalid component is detected, an predefined `ILLEGAL_TT2000_VALUE` (-99999999999LL) is returned.

---

### 10.2 `CDF_TIME_to_UTC_parts`

`CDF_TT2000_to_UTC_parts` decomposes a `CDF_TIME_TT2000` value into the individual UTC-based time components.

The variable argument form:

```c
void CDF_TT2000_to_UTC_parts(
    long long tt2000,            /* in -- The CDF_TIME_TT2000 value. */
    double *year,                /* out -- Year (AD). */
    double *month,               /* out -- Month */
    double *day,                 /* out -- Day */
    ....
    double TT2000NULL);
```

The full form:

```c
void CDF_TT2000_to_UTC_parts(
    long long tt2000,            /* in -- The CDF_TIME_TT2000 value. */
    double *year,                /* out -- Year (AD). */
    double *month,               /* out -- Month */
    double *day,                 /* out -- Day */
    ....
    double TT2000NULL);
```
This function is also aliased as **TT2000breakdown** for short. This function accepts variable number of arguments after the first four components of TT2000 value, year, month and day. It allows a full argument list of ten (10) fields: tt2000, year, month, day, hour, minute, second, millisecond, microsecond and nanosecond. If less than the full arguments are passed in for the decomposed date/time fields, a predefined **TT2000NULL** has to be appended to signify the end of argument list. Without it, an unexpected field value might be returned. Even all components are defined as double, only the very last argument may have really fractional value, e.g.,

For decomposing into three date/time arguments (sub-day),

CDF_TT2000_to_UTC_parts(tt2000, &year, &month, &day, TT2000NULL);

For decomposing into four date/time arguments (sub-hour),

CDF_TT2000_to_UTC_parts(tt2000, &year, &month, &day, &hour, TT2000NULL);

For decomposing into five date/time arguments (sub-minute),

CDF_TT2000_to_UTC_parts(tt2000, &year, &month, &day, &hour, &minute, TT2000NULL);

For decomposing into six date/time arguments (sub-second),

CDF_TT2000_to_UTC_parts(tt2000, &year, &month, &day, &hour, &minute, &second, TT2000NULL);

For decomposing into the complete argument list:

CDF_TT2000_to_UTC_parts(tt2000, &year, &month, &day, &hour, &minute, &second, &msec, &usec, &nsec);

### 10.3 CDF_TT2000_to_UTC_string

CDF_TT2000_to_UTC_string encodes a CDF_TIME_TT2000 value into the standard UTC-based date/time character string. The default format of the string is of **ISO 8601** format: yyyy-mm-ddT hh:ms:ss.mmmuuunnn where yyyy is the year (1707-2292), mm is the month (01-12), dd is the day of the month (1-31), hh is the hour (0-23), nn is the minute (0-59), ss is the second (0-59 or 0-60 if leap second), mmm is the millisecond (0-999), uuu is the microsecond (0-999) and nnn is the nanosecond (0-999).

The variable argument form:

```c
void CDF_TT2000_to_UTC_string(  
    long long tt2000;  /* in -- The CDF_TIME_TT2000 value. */  
    char *string);  /* out -- encode UTC string */
```

The full form:
void CDF_TT2000_to_UTC_string(
    long long tt2000; /* in -- The CDF_TIME_TT2000 value. */
    char *string, /* out -- encode UTC string */
    int form); /* in -- The string format. */

This function is also aliased as `encodeTT2000` for short. This function accepts variable number of arguments after the first two components of TT2000 value, and UTC string. It allows an optional argument field of an integer for format. If the format is not passed in, a format of value 3 is assumed and the default encoded UTC string is returned. The format has a valid value from 0 to 3.

For a format of value 0, the encoded UTC string is `DD-Mon-YYYY hh:mm:ss.mmmuuunnn`, where DD is the day of the month (1-31), Mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec), YYYY is the year, hh is the hour (0-23), mm is the minute (0-59 or 0-60 if leap second), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999). The encoded string has a length of `TT2000_0_STRING_LEN (30)`.

For a format of value 1, the encoded UTC string is `YYYYMMDD.ttttttttt`, where YYYY is the year, MM is the month (1-12) DD is the day of the month (1-31), and ttttttttt is sub-day.(0-999999999). The encoded string has a length of `TT2000_1_STRING_LEN (19)`.

For a format of value 2, the encoded UTC 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 or 0-60 if leap second). The encoded string has a length of `TT2000_2_STRING_LEN (14)`.

For a format of value 3, the encoded UTC string is `YYYY-MM-DDThh:mm:ss.mmmuuunnn`, 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 or 0-60 if leap second), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999). The encoded string has a length of `TT2000_3_STRING_LEN (29)`.

### 10.4 CDF_TT2000_from.UTC_string

`CDF_TT2000_from.UTC_string` parses a standard UTC-based date/time character string and returns a CDF_TIME_TT2000 value. The format of the string is one of the strings produced by the `CDF_TT2000_to.UTC_string` function described in Section 10.3. If the epoch is outside the range for TT2000, the value returned will be `ILLEGAL_TT2000_VALUE`.

```c
long long CDF_TT2000_from.UTC_string(
    char *epString); /* out -- CDF_TIME_TT2000 value returned. */
/* in -- The standard date/time character string. */
```

This function is also aliased as `parseTT2000` for short.

### 10.5 CDF_TT2000_from.UTC_EPOCH
CDF_TT2000_from.UTC_EPOCH converts a value of CDF_EPOCH type to CDF_TIME_TT2000 type. If the epoch is outside the range for TT2000, the value returned will be ILLEGAL_TT2000_VALUE. If the epoch is a predefined, filled dummy value, DUMMY_TT2000_VALUE is returned.

```
long long CDF_TT2000_from.UTC_EPOCH( /* out -- CDF_TIME_TT2000 value returned. */
    double epoch);                   /* in -- CDF_EPOCH value. */
```

This function converts a CDF_EPOCH data value to CDF_TIME_TT2000 value. Both microsecond and nanosecond fields for TT2000 are zero-filled.

### 10.6 CDF_TT2000_to.UTC_EPOCH

CDF_TT2000_to.UTC_EPOCH converts a value in CDF_TIME_TT2000 type to CDF_EPOCH type.

```
double CDF_TT2000_to.UTC_EPOCH( /* out -- The CDF_EPOCH value */
    long long tt2000);               /* in -- The CDF_TIME_TT2000 value. */
```

The microsecond and nanosecond fields in TT2000 are ignored. As the CDF_EPOCH type does not have leap seconds, the date/time falls on a leap second from TT2000 type will be converted to the zero (0) second of the next day.

### 10.7 CDF_TT2000_from.UTC_EPOCH16

CDF_TT2000_from.UTC_EPOCH16 converts a data value in CDF_EPOCH16 type to CDF_TT2000 type. If the epoch is outside the range for TT2000, the value returned will be ILLEGAL_TT2000_VALUE. If the epoch is a predefined, filled dummy value, DUMMY_TT2000_VALUE is returned.

```
long long CDF_TT2000_from.UTC_EPOCH16( /* out -- CDF_TIME_TT2000 value returned. */
    double *epoch16);               /* in -- The CDF_EPOCH16 value. */
```

The picoseconds from CDF_EPOCH16 is ignored.

### 10.8 CDF_TT2000_to.UTC_EPOCH16

CDF_TT2000_to.UTC_EPOCH16 converts a data value in CDF_TIME_TT2000 type to CDF_EPOCH16 type.

```
void CDF_TT2000_to.UTC_EPOCH16( /* in -- The CDF_TIME_TT2000 value. */
    long long tt2000;               /* out -- CDF_EPOCH16 value */
    double *epoch16);              /* out -- CDF_EPOCH16 value */
```

The picoseconds to CDF_EPOCH16 are zero(0)-filled. As the CDF_EPOCH16 type does not have leap seconds, the date/time falls on a leap second in TT2000 type will be converted to the zero (0) second of the next day.
Appendix A

A.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.

A.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.

- **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_LEN256
  characters. The attribute was created but with a truncated name.
  [Warning]

- **BAD_ALLOCATE_RECS**: An illegal number of records to allocate for a variable was
  specified. For RV variables the number must be one or greater.
  For NRV variables the number must be exactly one. [Error]

- **BAD_ARGUMENT**: An illegal/undefined argument was passed. Check that all
  arguments are properly declared and initialized. [Error]
<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_ATTR_NAME</td>
<td>Illegal attribute name specified. Attribute names must contain at least one character, and each character must be printable. [Error]</td>
</tr>
<tr>
<td>BAD_ATTR_NUM</td>
<td>Illegal attribute number specified. Attribute numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_BLOCKING_FACTOR</td>
<td>An illegal blocking factor was specified. Blocking factors must be at least zero (0). [Error]</td>
</tr>
<tr>
<td>BAD_CACHESIZE</td>
<td>An illegal number of cache buffers was specified. The value must be at least zero (0). [Error]</td>
</tr>
<tr>
<td>BAD_CDF_EXTENSION</td>
<td>An illegal file extension was specified for a CDF. In general, do not specify an extension except possibly for a single-file CDF that has been renamed with a different file extension or no file extension. [Error]</td>
</tr>
<tr>
<td>BAD_CDF_ID</td>
<td>CDF identifier is unknown or invalid. The CDF identifier specified is not for a currently open CDF. [Error]</td>
</tr>
<tr>
<td>BAD_CDF_NAME</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_CDFSTATUS</td>
<td>Unknown CDF status code received. The status code specified is not used by the CDF library. [Error]</td>
</tr>
<tr>
<td>BAD_CHECKSUM</td>
<td>An illegal checksum mode received. It is invalid or currently not supported. [Error]</td>
</tr>
<tr>
<td>BAD_COMPRESSION_PARM</td>
<td>An illegal compression parameter was specified. [Error]</td>
</tr>
<tr>
<td>BAD_DATA_TYPE</td>
<td>An unknown data type was specified or encountered. The CDF data types are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_DECODING</td>
<td>An unknown decoding was specified. The CDF decodings are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_DIM_COUNT</td>
<td>Illegal dimension count specified. A dimension count must be at least one (1) and not greater than the size of the dimension. [Error]</td>
</tr>
<tr>
<td>BAD_DIM_INDEX</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_DIM_INTERVAL</td>
<td>Illegal dimension interval specified. Dimension intervals must be at least one (1). [Error]</td>
</tr>
</tbody>
</table>

46 The status code BAD_BLOCKING_FACTOR was previously named BAD_EXTEND_RECS.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_DIM_SIZE</td>
<td>Illegal dimension size specified. A dimension size must be at least one (1). [Error]</td>
</tr>
<tr>
<td>BAD_ENCODING</td>
<td>Unknown data encoding specified. The CDF encodings are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_ENTRY_NUM</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_FNC_OR_ITEM</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_FORMAT</td>
<td>Unknown format specified. The CDF formats are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_INITIAL_RECS</td>
<td>An illegal number of records to initially write has been specified. The number of initial records must be at least one (1). [Error]</td>
</tr>
<tr>
<td>BAD_MAJORITY</td>
<td>Unknown variable majority specified. The CDF variable majorities are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_MALLOC</td>
<td>Unable to allocate dynamic memory - system limit reached. Contact CDF User Support if this error occurs. [Error]</td>
</tr>
<tr>
<td>BAD_NEGtoPOSfp0_MODE</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_NUM_DIMS</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_NUM_ELEMS</td>
<td>The number of elements of the data type is illegal. The number of elements must be at least one (1). For variables with a non-character data type, the number of elements must always be one (1). [Error]</td>
</tr>
<tr>
<td>BAD_NUM_VARS</td>
<td>Illegal number of variables in a record access operation. [Error]</td>
</tr>
<tr>
<td>BAD_READONLY_MODE</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_REC_COUNT</td>
<td>Illegal record count specified. A record count must be at least one (1). [Error]</td>
</tr>
<tr>
<td>BAD_REC_INTERVAL</td>
<td>Illegal record interval specified. A record interval must be at least one (1). [Error]</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BAD_REC_NUM</td>
<td>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]</td>
</tr>
<tr>
<td>BAD_SCOPE</td>
<td>Unknown attribute scope specified. The attribute scopes are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_SCRATCH_DIR</td>
<td>An illegal scratch directory was specified. The scratch directory must be writeable and accessible (if a relative path was specified) from the directory in which the application has been executed. [Error]</td>
</tr>
<tr>
<td>BAD_SPARSEARRAYS_PARM</td>
<td>An illegal sparse arrays parameter was specified. [Error]</td>
</tr>
<tr>
<td>BAD_VAR_NAME</td>
<td>Illegal variable name specified. Variable names must contain at least one character and each character must be printable. [Error]</td>
</tr>
<tr>
<td>BAD_VAR_NUM</td>
<td>Illegal variable number specified. Variable numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]</td>
</tr>
<tr>
<td>BAD_zMODE</td>
<td>Illegal zMode specified. The CDF zModes are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]</td>
</tr>
<tr>
<td>CANNOT_ALLOCATE_RECORDS</td>
<td>Records cannot be allocated for the given type of variable (e.g., a compressed variable). [Error]</td>
</tr>
<tr>
<td>CANNOT_CHANGE</td>
<td>Because of dependencies on the value, it cannot be changed. Some possible causes of this error follow:</td>
</tr>
</tbody>
</table>

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 or if a compressed single-file CDF.

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 or after records have been allocated for that variable.

5. Changing a variable's record variance after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.

6. Changing a variable's dimension variances after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.
7. Writing “initial” records to a variable after a value (excluding the pad value) has already been written to that variable.

8. Changing a variable's blocking factor when a compressed variable and a value (excluding the pad value) has been written or when a variable with sparse records and a value has been accessed.

9. Changing an attribute entry's data specification where the new specification is not equivalent to the old specification.

**CANNOT_COMPRESS**

The CDF or variable cannot be compressed. For CDFs, this occurs if the CDF has the multi-file format. For variables, this occurs if the variable is in a multi-file CDF, values have been written to the variable, or if sparse arrays have already been specified for the variable. [Error]

**CANNOT_SPARSEARRAYS**

Sparse arrays cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, records have been allocated for the variable, or if compression has already been specified for the variable. [Error]

**CANNOT_SPARSERECORDS**

Sparse records cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, or records have been allocated for the variable. [Error]

**CDF_CLOSE_ERROR**

Error detected while trying to close CDF. Check that sufficient disk space exists for the dotCDF file and that it has not been corrupted. [Error]

**CDF_CREATE_ERROR**

Cannot create the CDF specified - error from file system. Make sure that sufficient privilege exists to create the dotCDF file in the disk/directory location specified and that an open file quota has not already been reached. [Error]

**CDF_DELETE_ERROR**

Cannot delete the CDF specified - error from file system. Insufficient privileges exist the delete the CDF file(s). [Error]

**CDF_EXISTS**

The CDF named already exists - cannot create it. The CDF library will not overwrite an existing CDF. [Error]

**CDF_INTERNAL_ERROR**

An unexpected condition has occurred in the CDF library. Report this error to CDFsupport. [Error]

**CDF_NAME_TRUNC**

CDF file name 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 the CDF specified - error from file system. Check that the dotCDF 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]
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF_READ_ERROR</td>
<td>Failed to read the CDF file - error from file system. Check that the dotCDF file is not corrupted.</td>
</tr>
<tr>
<td>CDF_WRITE_ERROR</td>
<td>Failed to write the CDF file - error from file system. Check that the dotCDF file is not corrupted.</td>
</tr>
<tr>
<td>CHECKSUM_ERROR</td>
<td>The data integrity verification through the checksum failed.</td>
</tr>
<tr>
<td>CHECKSUM_NOT_ALLOWED</td>
<td>The checksum is not allowed for old versioned files.</td>
</tr>
<tr>
<td>COMPRESSION_ERROR</td>
<td>An error occurred while compressing a CDF or block of variable records. This is an internal error in the CDF library. Contact CDF User Support.</td>
</tr>
<tr>
<td>CORRUPTED_V2_CDF</td>
<td>This Version 2 CDF is corrupted. An error has been detected in the CDF's control information. If the CDF file(s) are known to be valid, please contact CDF User Support.</td>
</tr>
<tr>
<td>DECOMPRESSION_ERROR</td>
<td>An error occurred while decompressing a CDF or block of variable records. The most likely cause is a corrupted dotCDF file.</td>
</tr>
<tr>
<td>DID_NOT_COMPRESS</td>
<td>For a compressed variable, a block of records did not compress to smaller than their uncompressed size. They have been stored uncompressed. This can result if the blocking factor is set too low or if the characteristics of the data are such that the compression algorithm chosen is unsuitable.</td>
</tr>
<tr>
<td>EMPTY_COMPRESSED_CDF</td>
<td>The compressed CDF being opened is empty. This will result if a program, which was creating/modifying, the CDF abnormally terminated.</td>
</tr>
<tr>
<td>END_OF_VAR</td>
<td>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).</td>
</tr>
<tr>
<td>FORCED_PARAMETER</td>
<td>A specified parameter was forced to an acceptable value (rather than an error being returned).</td>
</tr>
<tr>
<td>IBM_PC_OVERFLOW</td>
<td>An operation involving a buffer greater than 64k bytes in size has been specified for PCs running 16-bit DOS/Windows 3.*.</td>
</tr>
<tr>
<td>ILLEGAL_EPOCH_VALUE</td>
<td>Illegal component is detected in computing an epoch value or an illegal epoch value is provided in decomposing an epoch value.</td>
</tr>
<tr>
<td>ILLEGAL_FOR_SCOPE</td>
<td>The operation is illegal for the attribute's scope. For example, only gEntries may be written for gAttributes - not rEntries or zEntries.</td>
</tr>
<tr>
<td>ILLEGAL_IN_zMODE</td>
<td>The attempted operation is illegal while in zMode. Most operations involving rVariables or rEntries will be illegal.</td>
</tr>
</tbody>
</table>
ILLEGAL_ON_V1_CDF  The specified operation (i.e., opening) is not allowed on 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]

NA_FOR_VARIABLE  The attempted operation is not applicable to the given variable. [Warning]

NEGATIVE_FP_ZERO  One or more of the values read/written are -0.0 (An illegal value on VAXes and DEC Alphas running OpenVMS). [Warning]

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, an attempt was made to save the changes made prior to the severe error. In any event, the CDF should still be closed. [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 file name specified is correct. [Error]

NO_SUCH.Entry  No such entry for specified attribute. [Error]

NO_SUCH_RECORD  The specified record does not exist for the given variable. [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]
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_VARS_IN_CDF</td>
<td>This CDF contains no rVariables. The operation performed is not applicable to a CDF with no rVariables. [Informational]</td>
</tr>
<tr>
<td>NO_WRITE_ACCESS</td>
<td>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]</td>
</tr>
<tr>
<td>NOT_A_CDF</td>
<td>Named CDF is corrupted or not actually a CDF. Contact CDF User Support if you are sure that the specified file is a CDF that should be readable by the CDF distribution being used. [Error]</td>
</tr>
<tr>
<td>NOT_A_CDF_OR_NOT_SUPPORTED</td>
<td>This can occur if an older CDF distribution is being used to read a CDF created by a more recent CDF distribution. Contact CDF User Support if you are sure that the specified file is a CDF that should be readable by the CDF distribution being used. CDF is backward compatible but not forward compatible. [Error]</td>
</tr>
<tr>
<td>PRECEDEDING_RECORDS_ALLOCATED</td>
<td>Because of the type of variable, records preceding the range of records being allocated were automatically allocated as well. [Informational]</td>
</tr>
<tr>
<td>READ_ONLY_DISTRIBUTION</td>
<td>Your CDF distribution has been built to allow only read access to CDFs. Check with your system manager if you require write access. [Error]</td>
</tr>
<tr>
<td>READ_ONLY_MODE</td>
<td>The CDF is in read-only mode - modifications are not allowed. [Error]</td>
</tr>
<tr>
<td>SCRATCH_CREATE_ERROR</td>
<td>Cannot create a scratch file - error from file system. If a scratch directory has been specified, ensure that it is writeable. [Error]</td>
</tr>
<tr>
<td>SCRATCH_DELETE_ERROR</td>
<td>Cannot delete a scratch file - error from file system. [Error]</td>
</tr>
<tr>
<td>SCRATCH_READ_ERROR</td>
<td>Cannot read from a scratch file - error from file system. [Error]</td>
</tr>
<tr>
<td>SCRATCH_WRITE_ERROR</td>
<td>Cannot write to a scratch file - error from file system. [Error]</td>
</tr>
<tr>
<td>SINGLE_FILE_FORMAT</td>
<td>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]</td>
</tr>
<tr>
<td>SOME_ALREADY_ALLOCATED</td>
<td>Some of the records being allocated were already allocated. [Informational]</td>
</tr>
<tr>
<td>TOO_MANY_PARMS</td>
<td>A type of sparse arrays or compression was encountered having too many parameters. This could be causes by a corrupted CDF or if the CDF was created/modified by a CDF distribution more recent than the one being used. [Error]</td>
</tr>
<tr>
<td>TOO_MANY_VARS</td>
<td>A multi-file CDF on a 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]</td>
</tr>
<tr>
<td>UNKNOWN_COMPRESSION</td>
<td>An unknown type of compression was specified or encountered. [Error]</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UNKNOWN_SPARSENESS</td>
<td>An unknown type of sparseness was specified or encountered. [Error]</td>
</tr>
<tr>
<td>UNSUPPORTED_OPERATION</td>
<td>The attempted operation is not supported at this time. [Error]</td>
</tr>
<tr>
<td>VAR_ALREADY_CLOSED</td>
<td>The specified variable is already closed. [Informational]</td>
</tr>
<tr>
<td>VAR_CLOSE_ERROR</td>
<td>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]</td>
</tr>
<tr>
<td>VAR_CREATE_ERROR</td>
<td>An error occurred while creating a variable file in a multi-file CDF. Check that a file quota has not been reached. [Error]</td>
</tr>
<tr>
<td>VAR_DELETE_ERROR</td>
<td>An error occurred while deleting a variable file in a multi-file CDF. Check that sufficient privilege exist to delete the CDF files. [Error]</td>
</tr>
<tr>
<td>VAR_EXISTS</td>
<td>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]</td>
</tr>
<tr>
<td>VAR_NAME_TRUNC</td>
<td>Variable name truncated to CDF_VAR_NAME_LEN256 characters. The variable was created but with a truncated name. [Warning]</td>
</tr>
<tr>
<td>VAR_OPEN_ERROR</td>
<td>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]</td>
</tr>
<tr>
<td>VAR_READ_ERROR</td>
<td>Failed to read variable as requested - error from file system. Check that the associated file is not corrupted. [Error]</td>
</tr>
<tr>
<td>VAR_WRITE_ERROR</td>
<td>Failed to write variable as requested - error from file system. Check that the associated file is not corrupted. [Error]</td>
</tr>
<tr>
<td>VIRTUAL_RECORD_DATA</td>
<td>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 the Concepts chapter in the CDF User's Guide. [Informational]</td>
</tr>
</tbody>
</table>
Appendix B

B.1 Original Standard Interface

CDFstatus CDFattrCreate (id, attrName, attrScope, attrNum)
CDFid  id; /* in */
char   *attrName; /* in */
long   attrScope; /* in */
long   *attrNum; /* out */

CDFstatus CDFattrEntryInquire (id, attrNum, entryNum, dataType, numElements)
CDFid  id; /* in */
long   attrNum; /* in */
long   entryNum; /* in */
long   *dataType; /* out */
long   *numElements; /* out */

CDFstatus CDFattrGet (id, attrNum, entryNum, value)
CDFid  id; /* in */
long   attrNum; /* in */
long   entryNum; /* in */
void   *value; /* out */

CDFstatus CDFattrInquire (id, attrNum, attrName, attrScope, maxEntry)
CDFid  id; /* in */
long   attrNum; /* in */
char   *attrName; /* out */
long   *attrScope; /* out */
long   *maxEntry; /* out */

long CDFattrNum (id, attrName)
CDFid  id; /* in */
char   *attrName; /* in */

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 CDFattrRename (id, attrNum, attrName)
CDFid  id; /* in */
long   attrNum; /* in */
char *attrName;          /* in */

CDFstatus CDFclose (id)
CDFid id;               /* in */

CDFstatus CDFcreate (CDFname, numDims, dimSizes, encoding, majority, id)
char *CDFname;          /* in */
long numDims;           /* in */
long dimSizes[];        /* in */
long encoding;          /* in */
long majority;          /* in */
CDFid *id;              /* out */

CDFstatus CDFdelete (id)
CDFid id;               /* in */

CDFstatus CDFdoc (id, version, release, text)
CDFid id;               /* in */
long *version;          /* out */
long *release;          /* out */
char text[CDF_DOCUMENT_LEN+1];            /* out */

CDFstatus CDFerror (status, message)
CDFstatus status;       /* in */
char message[CDF_STATUSTEXT_LEN+1];       /* out */

CDFstatus CDFgetrVarsRecordData (id, numVars, varNames, varRecNum, buffer)
CDFid id;               /* in */
long numVars;           /* in */
char *varNames[];       /* in */
void *varRecNum;        /* in */
void *buffer[];         /* out */

CDFstatus CDFgetzVarsRecordData (id, numVars, varNames, varRecNum, buffer)
CDFid id;               /* in */
long numVars;           /* in */
char *varNames[];       /* in */
void *varRecNum;        /* in */
void *buffer[];         /* 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 CDFopen (CDFname, id)
char *CDFname;          /* in */
CDFid *id;              /* out */

CDFstatus CDFputrVarsRecordData (id, numVars, varNames, varRecNum, buffer)
CDFid id; /* in */
long numVars; /* in */
char *varNames[]; /* in */
long varRecNum; /* in */
void *buffer; /* in */

CDFstatus CDFputzVarsRecordData (id, numVars, varNames, varRecNum, buffer)
CDFid id; /* in */
long numVars; /* in */
char *varNames[]; /* in */
long varRecNum; /* in */
void *buffer[]; /* in */

CDFstatus CDFvarClose (id, varNum)
CDFid id; /* in */
long varNum; /* 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 */

CDFstatus CDFvarGet (id, varNum, recNum, indices, value)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
long indices[]; /* in */
void *value; /* out */

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 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 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 */

long CDFvarNum (id, varName)
CDFid id; /* in */
char *varName; /* in */

CDFstatus CDFvarPut (id, varNum, recNum, indices, value)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
long indices[]; /* in */
void *value; /* in */

CDFstatus CDFvarRename (id, varNum, varName)
CDFid id; /* in */
long varNum; /* in */
char *varName; /* in */
B.2   Extended Standard Interface

CDFstatus CDFcloseCDF (id)
CDFid  id;                             /* in */

CDFstatus CDFclosezVar (id, varNum)
CDFid  id;                             /* in */
long   varNum;                         /* in */

CDFstatus CDFconfirmAttrExistence (id, attrName)
CDFid  id;                             /* in */
char   *attrName;                      /* in */

CDFstatus CDFconfirmgEntryExistence (id, attrNum, entryNum)
CDFid  id;                             /* in */
long   attrNum;                        /* in */
long   entryNum;                       /* in */

CDFstatus CDFconfirmrEntryExistence (id, attrNum, entryNum)
CDFid  id;                             /* in */
long   attrNum;                        /* in */
long   entryNum;                       /* in */

CDFstatus CDFconfirmzEntryExistence (id, attrNum, entryNum)
CDFid  id;                             /* in */
long   attrNum;                        /* in */
long   entryNum;                       /* in */

CDFstatus CDFconfirmzVarExistence (id, varNum)
CDFid  id;                             /* in */
long   varNum;                         /* in */

CDFstatus CDFconfirmzVarPadValueExistence (id, varNum)
CDFid  id;                             /* in */
long   varNum;                         /* in */

CDFstatus CDFcreateAttr (id, attrName, scope, attrNum)
CDFid  id;                             /* in */
char   *attrName;                       /* in */
long   scope;                           /* in */
long   *attrNum;                        /* out */

CDFstatus CDFcreateCDF (CDFname, dimSizes, id)
char   *CDFname;                        /* in */
CDFid  *id;                             /* out */

CDFstatus CDFcreatezVar (id, varName, dataType, numElements, numDims, 
dimSizes, recVary, dimVarys, varNum)
CDFid  id;                             /* in */
char   *varName;                        /* in */
long   dataType;                        /* in */
long   numElements;                    /* in */
long numDims; /* in */
long dimSizes[]; /* in */
long recVary; /* in */
long dimVarys[]; /* in */
long *varNum; /* out */

CDFstatus CDFdeleteCDF (id)
CDFid id; /* in */

CDFstatus CDFdeleteAttr (id, attrNum)
CDFid id; /* in */
long attrNum; /* in */

CDFstatus CDFdeleteAttrgEntry (id, attrNum, entryNum)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */

CDFstatus CDFdeleteAttrrEntry (id, attrNum, entryNum)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */

CDFstatus CDFdeleteAttrzEntry (id, attrNum, entryNum)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */

CDFstatus CDFdeletezVar (id, varNum)
CDFid id; /* in */
long varNum; /* in */

CDFstatus CDFdeletezVarRecords (id, varNum, startRec, endRec)
CDFid id; /* in */
long varNum; /* in */
long startRec; /* in */
long endRec; /* in */

CDFstatus CDFgetAttrgEntryDataType (id, attrNum, entryNum, dataType)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *dataType; /* out */

CDFstatus CDFgetAttrgEntryNumElements (id, attrNum, entryNum, numElems)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *numElems; /* out */

CDFstatus CDFgetAttrgEntry (id, attrNum, entryNum, value)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
void *value; /* out */
CDFstatus CDFgetAttrEntry (id, attrNum, entryNum, value)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
void *value; /* out */

CDFstatus CDFgetAttrMaxgEntry (id, attrNum, entryNum)
CDFid id; /* in */
long attrNum; /* in */
long *entryNum; /* out */

CDFstatus CDFgetAttrMaxrEntry (id, attrNum, entryNum)
CDFid id; /* in */
long attrNum; /* in */
long *entryNum; /* out */

CDFstatus CDFgetAttrMaxzEntry (id, attrNum, entryNum)
CDFid id; /* in */
long attrNum; /* in */
long *entryNum; /* out */

CDFstatus CDFgetAttrName (id, attrNum, attrName)
CDFid id; /* in */
long attrNum; /* in */
char *attrName; /* out */

long CDFgetAttrNum (id, attrName) /* out */
CDFid id; /* in */
char *attrName; /* in */

CDFstatus CDFgetAttrrEntryDataType (id, attrNum, entryNum, dataType)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *dataType; /* out */

CDFstatus CDFgetAttrrEntryNumElements (id, attrNum, entryNum, numElems)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *numElems; /* out */

CDFstatus CDFgetAttrScope (id, attrNum, scope)
CDFid id; /* in */
long attrNum; /* in */
long *scope; /* out */

CDFstatus CDFgetAttrzEntry (id, attrNum, entryNum, value)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
void *value; /* out */

CDFstatus CDFgetAttrzEntryDataType (id, attrNum, entryNum, dataType)
CDFid id; /* in */
long attrNum; /* in */
long entryNum;  /* in */
long *dataType;  /* out */

CDFstatus CDFgetAttrzEntryNumElements (id, attrNum, entryNum, numElems)
CDFid id;  /* in */
long attrNum;  /* in */
long entryNum;  /* in */
long *numElems;  /* out */

CDFstatus CDFgetCacheSize (id, numBuffers)
CDFid id;  /* in */
long *numBuffers;  /* out */

CDFstatus CDFgetChecksum (id, checksum)
CDFid id;  /* in */
long *checksum;  /* out */

CDFstatus CDFgetCompression (id, compressionType, compressionParms, compressionPercent)
CDFid id;  /* in */
long *compressionType;  /* out */
long compressionParms[];  /* out */
long *compressionPercent;  /* out */

CDFstatus CDFgetCompressionCacheSize (id, numBuffers)
CDFid id;  /* in */
long *numBuffers;  /* out */

CDFstatus CDFgetCompressionInfo (cdfName, compressionType, compressionParms, compressionSize, uncompressionSize)
char *cdfName;  /* in */
long *compressionType;  /* out */
long compressionParms[];  /* out */
OFF_T *compressionSize;  /* out */
OFF_T *uncompressionSize;  /* out */

CDFstatus CDFgetCopyright (id, Copyright)
CDFid id;  /* in */
char *Copyright;  /* out */

CDFstatus CDFgetDataTypeSize (dataType, numBytes)
long dataType;  /* in */
long *numBytes;  /* out */

CDFstatus CDFgetDecoding (id, decoding)
CDFid id;  /* in */
long *decoding;  /* out */

CDFstatus CDFgetEncoding (id, encoding)
CDFid id;  /* in */
long *encoding;  /* out */

int CDFgetFileBackward ()

CDFstatus CDFgetFormat (id, format)
CDFid id;  /* in */
long *format; /* out */

CDFstatus CDFgetLibraryCopyright (Copyright)
char *Copyright; /* out */

CDFstatus CDFgetLibraryVersion (version, release, increment, subIncrement)
long *version; /* out */
long *release; /* out */
long *increment; /* out */
char *subIncrement; /* out */

CDFstatus CDFgetMajority (id, majority)
CDFid id; /* in */
long *majority; /* out */

CDFstatus CDFgetMaxWrittenRecNums (id, maxRecrVars, maxReczVars)
CDFid id; /* in */
long *maxRecrVars; /* out */
long *maxReczVars; /* out */

CDFstatus CDFgetName (id, name)
CDFid id; /* in */
char *name; /* out */

CDFstatus CDFgetNegtoPosfp0Mode (id, negtoPosfp0)
CDFid id; /* in */
long *negtoPosfp0; /* out */

CDFstatus CDFgetNumAttrgEntries (id, attrNum, entries)
CDFid id; /* in */
long attrNum; /* in */
long *entries; /* out */

CDFstatus CDFgetNumAttributes (id, numAttrs)
CDFid id; /* in */
long *numAttrs; /* out */

CDFstatus CDFgetNumAttrrEntries (id, attrNum, entries)
CDFid id; /* in */
long attrNum; /* in */
long *entries; /* out */

CDFstatus CDFgetNumAttrzEntries (id, attrNum, entries)
CDFid id; /* in */
long attrNum; /* in */
long *entries; /* out */

CDFstatus CDFgetNumAttributes (id, numAttrs)
CDFid id; /* in */
long *numAttrs; /* out */

CDFstatus CDFgetNumAttributes (id, numAttrs)
CDFid id; /* in */
long *numAttrs; /* out */

CDFstatus CDFgetNumAttributes (id, numAttrs)
CDFid id; /* in */
long *numAttrs; /* out */

CDFstatus CDFgetNumAttributes (id, numAttrs)
CDFid id; /* in */
long *numAttrs; /* out */

CDFstatus CDFgetNumrVars (id, numVars)
CDFid *id; /* in */
long *numrVars; /* out */

CDFstatus CDFgetNumzVars (id, numzVars)
CDFid *id; /* in */
long *numzVars; /* out */

CDFstatus CDFgetReadOnlyMode (id, mode)
CDFid *id; /* in */
long *mode; /* out */

CDFstatus CDFgetStageCacheSize (id, numBuffers)
CDFid *id; /* in */
long *numBuffers; /* out */

CDFstatus CDFgetStatusText (status, text)
CDFstatus *status; /* in */
char *text; /* out */

CDFstatus CDFgetVarAllRecordsByVarName (id, varName, buffer)
CDFid *id; /* in */
char *varName; /* in */
void *buffer; /* out */

long CDFgetVarNum (id, varName)
CDFid *id; /* in */
char *varName; /* in */

int CDFgetValidate ()

CDFstatus CDFgetVarAllRecordsByVarName (id, varName, buffer)
CDFid *id; /* in */
char *varName; /* in */
void *buffer; /* out */

CDFstatus CDFgetVarRangeRecordsByVarName (id, varName, startRec, stopRec, buffer)
CDFid *id; /* in */
char *varName; /* in */
long startRec; /* in */
long stopRec; /* in */
void *buffer; /* out */

CDFstatus CDFgetVersion (id, version, release, increment)
CDFid *id; /* in */
long *version; /* out */
long *release; /* out */
long *increment; /* out */

CDFstatus CDFgetzMode (id, zMode)
CDFid *id; /* in */
long *zMode; /* out */

CDFstatus CDFgetzVarAllocRecords (id, varNum, allocRecs)
CDFid *id; /* in */
long varNum; /* in */
long *allocRecs; /* out */
CDFstatus CDFgetzVarAllRecordsByVarID (id, varNum, buffer)
CDFid id;               /* in */
long varNum;            /* in */
void *buffer;           /* out */

CDFstatus CDFgetzVarBlockingFactor (id, varNum, bf)
CDFid id;               /* in */
long varNum;            /* in */
long *bf;               /* out */

CDFstatus CDFgetzVarCacheSize (id, varNum, numBuffers)
CDFid id;               /* in */
long varNum;            /* in */
long *numBuffers;       /* out */

CDFstatus CDFgetzVarCompression (id, varNum, cType, cParms, cPercent)
CDFid id;               /* in */
long varNum;            /* in */
long *cType;            /* out */
long cParms[ ];         /* out */
long *cPercent;         /* out */

CDFstatus CDFgetzVarData (id, varNum, recNum, indices, value)
CDFid id;               /* in */
long varNum;            /* in */
long recNum;            /* in */
long indices[ ];        /* in */
void *value;            /* out */

CDFstatus CDFgetzVarDataType (id, varNum, dataType)
CDFid id;               /* in */
long varNum;            /* in */
long *dataType;         /* out */

CDFstatus CDFgetzVarDimSizes (id, varNum, dimSizes)
CDFid id;               /* in */
long varNum;            /* in */
long dimSizes[ ];       /* out */

CDFstatus CDFgetzVarDimVariances (id, varNum, dimVarys)
CDFid id;               /* in */
long varNum;            /* in */
long dimVarys[ ];       /* out */

CDFstatus CDFgetzVarMaxAllocRecNum (id, varNum, maxRec)
CDFid id;               /* in */
long varNum;            /* in */
long *maxRec;           /* out */

CDFstatus CDFgetzVarMaxWrittenRecNum (id, varNum, maxRec)
CDFid id;               /* in */
long varNum;            /* in */
long *maxRec;           /* out */

CDFstatus CDFgetzVarName (id, varNum, varName)
CDFid id; /* in */
long varNum; /* in */
char *varName; /* out */

CDFstatus CDFgetzVarNumDims (id, varNum, numDims)
CDFid id; /* in */
long varNum; /* in */
long *numDims; /* out */

CDFstatus CDFgetzVarNumElements (id, varNum, numElems)
CDFid id; /* in */
long varNum; /* in */
long *numElems; /* out */

CDFstatus CDFgetzVarNumRecsWritten (id, varNum, numRecs)
CDFid id; /* in */
long varNum; /* in */
long *numRecs; /* out */

CDFstatus CDFgetzVarPadValue (id, varNum, padValue)
CDFid id; /* in */
long varNum; /* in */
void *padValue; /* out */

CDFstatus CDFgetzVarRangeRecordsByVarID (id, varNum, startRec, stopRec, buffer)
CDFid id; /* in */
long varNum; /* in */
long startRec; /* in */
long stopRec; /* in */
void *buffer; /* out */

CDFstatus CDFgetzVarRecordData (id, varNum, recNum, buffer)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
void *buffer; /* out */

CDFstatus CDFgetzVarRecVariance (id, varNum, recVary)
CDFid id; /* in */
long varNum; /* in */
long *recVary; /* out */

CDFstatus CDFgetzVarReservePercent (id, varNum, percent)
CDFid id; /* in */
long varNum; /* in */
long *percent; /* out */

CDFstatus CDFgetzVarSeqData (id, varNum, value)
CDFid id; /* in */
long varNum; /* in */
void *value; /* out */

CDFstatus CDFgetzVarSeqPos (id, varNum, recNum, indices)
CDFid id; /* in */
long varNum; /* in */
long *recNum; /* out */
long indices[]; /* out */

CDFstatus CDFgetzVarsMaxWrittenRecNum (id, recNum)
CDFid id; /* in */
long *recNum; /* out */

CDFstatus CDFgetzVarSparseRecords (id, varNum, sRecords)
CDFid id; /* in */
long varNum; /* in */
long *sRecords; /* out */

CDFstatus CDFgetzVarsRecordDatabyNumbers (id, numVars, varNums, varRecNum, buffer)
CDFid id; /* in */
long numVars; /* in */
long varNums[]; /* in */
long varRecNum; /* in */
void *buffer; /* out */

CDFstatus CDFhyperGetzVarData (id, varNum, recNum, rCount, recInterval, indices, counts, intervals, buffer)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
long recCount; /* in */
long recInterval; /* in */
long indices[]; /* in */
long counts[]; /* in */
long intervals[]; /* in */
void *buffer; /* out */

CDFstatus CDFhyperPutzVarData (id, varNum, recNum, reCount, recInterval, indices, counts, intervals, buffer)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
long recCount; /* in */
long recInterval; /* in */
long indices[]; /* in */
long counts[]; /* in */
long intervals[]; /* in */
void *buffer; /* in */

CDFstatus CDFinquireAttr (id, attrNum, attrName, attrScope, maxgEntry, maxrEntry, maxzEntry)
CDFid id; /* in */
long attrNum; /* in */
char *attrName; /* out */
long *attrScope; /* out */
long *maxgEntry; /* out */
long *maxrEntry; /* out */
long *maxzEntry; /* out */

CDFstatus CDFinquireAttrgEntry (id, attrNum, entryNum, dataType, numElems)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *dataType; /* out */
long *numElems; /* out */

CDFstatus CDFinquireAttrEntry (id, attrNum, entryNum, dataType, numElems)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *dataType; /* out */
long *numElems; /* out */

CDFstatus CDFinquireAttrzEntry (id, attrNum, entryNum, dataType, numElems)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long *dataType; /* out */
long *numElems; /* out */

CDFstatus CDFinquireCDF (id, numDims, dimSizes, encoding, majority, maxrRec,
numrVars, maxzRec, numzVars, numAttrs)
CDFid id; /* in */
long *numDims; /* out */
long *encoding; /* out */
long *majority; /* out */
long *maxrRec; /* out */
long *numrVars; /* out */
long *maxzRec; /* out */
long *numzVars; /* out */
long *numAttrs; /* out */

CDFstatus CDFinquirezVar (id, varNum, varName, dataType, numElems,
numDims, dimSizes, recVary, dimVarys)
CDFid id; /* in */
long varNum; /* in */
char *varName; /* out */
long *dataType; /* out */
long *numElems; /* out */
long *numDims; /* out */
long *dimSizes[]; /* out */
long *recVary; /* out */
long *dimVarys[]; /* out */

CDFstatus CDFinsertVarAllRecordsByVarName (id, varName, startRec, numRecs, buffer)
CDFid id; /* in */
char *varName; /* in */
long startRec; /* in */
long numRecs; /* in */
void *buffer; /* in */

CDFstatus CDFinsertVarAllRecordsByVarID (id, varNum, startRec, numRecs, buffer)
CDFid id; /* in */
long varNum; /* in */
long startRec; /* in */
long numRecs; /* in */
void *buffer; /* in */
CDFstatus CDFinsertzVarAllRecordsByVarID (id, varNum, startRec, numRecs, buffer)  
CDFid id;  /* in */  
long varNum;  /* in */  
long startRec;  /* in */  
long numRecs;  /* in */  
void *buffer;  /* in */

CDFstatus CDFputAttrgEntry (id, attrNum, entryNum, dataType, numElems, value)  
CDFid id;  /* in */  
long attrNum;  /* in */  
long entryNum;  /* in */  
long dataType;  /* in */  
long numElems;  /* in */  
void *value;  /* in */

CDFstatus CDFopenCDF (CDFname, id)  
char *CDFname;  /* in */  
CDFid *id;  /* out */

CDFstatus CDFputAttrrEntry (id, attrNum, entryNum, dataType, numElems, value)  
CDFid id;  /* in */  
long attrNum;  /* in */  
long entryNum;  /* in */  
long dataType;  /* in */  
long numElems;  /* in */  
void *value;  /* in */

CDFstatus CDFputAttrzEntry (id, attrNum, entryNum, dataType, numElems, value)  
CDFid id;  /* in */  
long attrNum;  /* in */  
long entryNum;  /* in */  
long dataType;  /* in */  
long numElems;  /* in */  
void *value;  /* in */

CDFstatus CDFputVarAllRecordsByVarName (id, varName, buffer)  
CDFid id;  /* in */  
char *varName;  /* in */  
void *buffer;  /* in */

CDFstatus CDFputVarRangeRecordsByVarName (id, varName, startRec, stopRec, buffer)  
CDFid id;  /* in */  
char *varName;  /* in */  
long startRec;  /* in */  
long stopRec;  /* in */  
void *buffer;  /* in */

CDFstatus CDFputzVarAllRecordsByVarID (id, varNum, buffer)  
CDFid id;  /* in */  
long varNum;  /* in */  
void *buffer;  /* in */

CDFstatus CDFputzVarData (id, varNum, recNum, indices, value)  
CDFid id;  /* in */  
long varNum;  /* in */
long recNum; /* in */
long indices[]; /* in */
void *value; /* in */

CDFstatus CDFputzVarRangeRecordsByVarID (id, varNum, startRec, stopRec, buffer)
CDFid id; /* in */
long varNum; /* in */
long startRec; /* in */
long stopRec; /* in */
void *buffer; /* in */

CDFstatus CDFputzVarRecordData (id, varNum, recNum, values)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
void *values; /* in */

CDFstatus CDFputzVarSeqData (id, varNum, value)
CDFid id; /* in */
long varNum; /* in */
void *value; /* in */

CDFstatus CDFputzVarsRecordDatabyNumbers (id, numVars, varNums, varRecNum, buffer)
CDFid id; /* in */
long numVars; /* in */
long varNums[]; /* in */
long varRecNum; /* in */
void *buffer; /* in */

CDFstatus CDFrenameAttr (id, attrNum, attrName)
CDFid id; /* in */
long attrNum; /* in */
char *attrName; /* in */

CDFstatus CDFrenamezVar (id, varNum, varName)
CDFid id; /* in */
long varNum; /* in */
char *varName; /* in */

CDFstatus CDFsetAttrgEntryDataSpec (id, attrNum, entryNum, dataType)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long dataType; /* in */

CDFstatus CDFsetAttrrEntryDataSpec (id, attrNum, entryNum, dataType)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long dataType; /* in */

CDFstatus CDFsetAttrScope (id, attrNum, scope)
CDFid id; /* in */
long attrNum; /* in */
long scope; /* in */
CDFstatus CDFsetAttrEntryDataSpec (id, attrNum, entryNum, dataType)
CDFid id; /* in */
long attrNum; /* in */
long entryNum; /* in */
long dataType; /* in */

CDFstatus CDFsetCacheSize (id, numBuffers)
CDFid id; /* in */
long numBuffers; /* in */

CDFstatus CDFsetChecksum (id, checksum)
CDFid id; /* in */
long checksum; /* in */

CDFstatus CDFsetCompression (id, compressionType, compressionParms)
CDFid id; /* in */
long compressionType; /* in */
long compressionParms[]; /* in */

CDFstatus CDFsetCompressionCacheSize (id, numBuffers)
CDFid id; /* in */
long numBuffers; /* in */

CDFstatus CDFsetDecoding (id, decoding)
CDFid id; /* in */
long decoding; /* in */

CDFstatus CDFsetEncoding (id, encoding)
CDFid id; /* in */
long encoding; /* in */

void CDFsetFileBackward (mode)
long mode; /* in */

CDFstatus CDFsetFormat (id, format)
CDFid id; /* in */
long format; /* in */

CDFstatus CDFsetMajority (id, majority)
CDFid id; /* in */
long majority; /* in */

CDFstatus CDFsetNegtoPosfp0Mode (id, negtoPosfp0)
CDFid id; /* in */
long negtoPosfp0; /* in */

CDFstatus CDFsetReadOnlyMode (id, readOnly)
CDFid id; /* in */
long readOnly; /* in */

CDFstatus CDFsetStageCacheSize (id, numBuffers)
CDFid id; /* in */
long numBuffers; /* in */

void CDFsetValidate (mode)
long mode; /* in */

CDFstatus CDFsetzMode (id, zMode)
CDFid id; /* in */
long zMode; /* in */

CDFstatus CDFsetzVarAllocBlockRecords (id, varNum, firstRec, lastRec)
CDFid id; /* in */
long varNum; /* in */
long firstRec; /* in */
long lastRec; /* in */

CDFstatus CDFsetzVarAllocRecords (id, varNum, numRecs)
CDFid id; /* in */
long varNum; /* in */
long numRecs; /* in */

CDFstatus CDFsetzVarBlockingFactor (id, varNum, bf)
CDFid id; /* in */
long varNum; /* in */
long bf; /* in */

CDFstatus CDFsetzVarCacheSize (id, varNum, numBuffers)
CDFid id; /* in */
long varNum; /* in */
long numBuffers; /* in */

CDFstatus CDFsetzVarCompression (id, varNum, compressionType, 
compressionParms)
CDFid id; /* in */
long varNum; /* in */
long compressionType; /* in */
long compressionParms[]; /* in */

CDFstatus CDFsetzVarDataSpec (id, varNum, dataType)
CDFid id; /* in */
long varNum; /* in */
long dataType; /* in */

CDFstatus CDFsetzVarDimVariances (id, varNum, dimVarys)
CDFid id; /* in */
long varNum; /* in */
long dimVarys[]; /* in */

CDFstatus CDFsetzVarInitialRecs (id, varNum, initialRecs)
CDFid id; /* in */
long varNum; /* in */
long initialRecs; /* in */

CDFstatus CDFsetzVarPadValue (id, varNum, padValue)
CDFid id; /* in */
long varNum; /* in */
void *padValue; /* in */

CDFstatus CDFsetzVarRecVariance (id, varNum, recVary)
CDFid id; /* in */
long varNum; /* in */
long recVary; /* in */

CDFstatus CDFsetzVarReservePercent (id, varNum, reservePercent)
CDFid id; /* in */
long varNum; /* in */
long reservePercent; /* in */

CDFstatus CDFsetzVarsCacheSize (id, numBuffers)
CDFid id; /* in */
long numBuffers; /* in */

CDFstatus CDFsetzVarSeqPos (id, varNum, recNum, indices)
CDFid id; /* in */
long varNum; /* in */
long recNum; /* in */
long indices[]; /* in */

CDFstatus CDFsetzVarSparseRecords (id, varNum, sRecords)
CDFid id; /* in */
long varNum; /* in */
long sRecords; /* in */
B.3 Internal Interface

CDFstatus CDFlib (op, ...)  
long op;       /* in */

CLOSE_   
        CDF_        
rVAR_  
zVAR_        

CONFIRM_  
        ATTR_      long *attrNum  /* out */
ATTR_EXISTENCE_   char *attrName  /* in */
CDF_       CDFid *id  /* out */
CDF_ACCESS_  
CDF_CACHESIZE_  long *numBuffers  /* out */
CDF_DECODING_  long *decoding  /* out */
CDF_NAME_  char CDFname[CDF_PATHNAME_LEN+1]  /* out */
CDF_NEGtoPOSfp0_MODE_  long *mode  /* out */
CDF_READONLY_MODE_  long *mode  /* out */
CDF_STATUS_  CDFstatus *status  /* out */
CDF_zMODE_  long *mode  /* out */
COMPRESS_CACHESIZE_  long *numBuffers  /* out */
gENTRY_  
gENTRY_EXISTENCE_  long entryNum  /* out */
rENTRY_  
rENTRY_EXISTENCE_  long entryNum  /* in */
rVAR_  
rVAR_CACHESIZE_  long *numBuffers  /* out */
rVAR_EXISTENCE_  char *varName  /* in */
rVAR_PADVALUE_  
rVAR_RESERVEDPERCENT_  long *percent  /* out */
rVAR_SEQPOS_  long *recNum  /* out */
rVARs_DIMCOUNTS_  long counts[CDF_MAX_DIMS]  /* out */
rVARs_DIMINDICES_  long indices[CDF_MAX_DIMS]  /* out */
rVARs_DIMINTERVALS_  long intervals[CDF_MAX_DIMS]  /* out */
rVARs_RECCOUNT_  long *recCount  /* out */
rVARs_RECINTERVAL_  long *recInterval  /* out */
rVARs_RECNUMBER_  long *recNum  /* out */
STAGE_CACHESIZE_  long *numBuffers  /* out */
zENTRY_  
zENTRY_EXISTENCE_  long entryNum  /* out */
zVAR_  
zVAR_CACHESIZE_  long *numBuffers  /* out */
zVAR_DIMCOUNTS_  long counts[CDF_MAX_DIMS]  /* out */
zVAR_DIMINDICES_  long indices[CDF_MAX_DIMS]  /* out */
zVAR_DIMINTERVALS_  long intervals[CDF_MAX_DIMS]  /* out */
zVAR_EXISTENCE_  char *varName  /* in */
zVAR_PADVALUE_ long *recCount /* out */
zVAR_RECACCOUNT_ long *recCount /* out */
zVAR_RECINTERVAL_ long *recInterval /* out */
zVAR_RECNUMBER_ long *recNum /* out */
zVAR_RESERVEPERCENT_ long *percent /* out */
zVAR_SEQPOS_ long *recNum /* out */

long indices[CDF_MAX_DIMS] /* out */

CREATE_
ATTR_ char *attrName /* in */
long scope /* in */
long *attrNum /* out */

CDF_ char *CDFname /* 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 */

DELETE_
ATTR_ CDF_ gENTRY_ rENTRY_ rVAR_ rVAR_RECORDS_ long firstRecord /* in */
long lastRecord /* in */

zENTRY_ zVAR_ zVAR_RECORDS_ long firstRecord /* in */
long lastRecord /* in */

GET_
ATTR_MAXgENTRY_ long *maxEntry /* out */
ATTR_MAXrENTRY_ long *maxEntry /* out */
ATTR_MAXzENTRY_ long *maxEntry /* out */
ATTR_NAME_ char attrName[CDF_ATTR_NAME_LEN256+1] /* out */

ATTR_NUMBER_ char *attrName /* in */
long *attrNum /* out */

ATTR_NUMgENTRIES_ long *numEntries /* out */

323
ATTR_NUMrENTRIES_  long *numEntries  /* out */
ATTR_NUMmENTRIES_  long *numEntries  /* out */
ATTR_SCOPE_  long *scope  /* out */
CDF_CHECKSUM_  long *checksum  /* out */
CDF_COMPRESSION_  long *cType  /* out */
long cParms[CDF_MAX_PARMS]  /* out */
long *cPct  /* out */
CDF_COPYRIGHT_  char Copyright[CDF_COPYRIGHT_LEN+1]  /* out */
CDF_ENCODING_  long *encoding  /* out */
CDF_FORMAT_  long *format  /* out */
CDF_INCREMENT_  long *increment  /* out */
CDF_INFO_  char *name  /* in */
long *cType  /* out */
long cParms[CDF_MAX_PARMS]  /* out */
OFF_T *cSize  /* out */
OFF_T *uSize  /* out */
CDF_MAJORITY_  long *majority  /* out */
CDF_NUMATTgRS_  long *numAtts  /* out */
CDF_NUMgATTRS_  long *numAtts  /* out */
CDF_NUMrVARS_  long *numVars  /* out */
CDF_NUMmVARS_  long *numVars  /* out */
CDF_RELEASE_  long *release  /* out */
CDF_VERSION_  long *version  /* out */
DATATYPE_SIZE_  long dataType  /* in */
long *numBytes  /* out */
gENTRY_DATA_  void *value  /* out */
gENTRY_DATATYPE_  long *dataType  /* out */
gENTRY_NUMELEMENTS_  long *numElements  /* out */
LIB_COPYRIGHT_  char Copyright[CDF_COPYRIGHT_LEN+1]  /* out */
LIB_INCREMENT_  long *increment  /* out */
LIB_RELEASE_  long *release  /* out */
LIB_subINCREMENT_  char *subincrement  /* out */
LIB_VERSION_  long *version  /* out */
ENTRY_DATA_  void *value  /* out */
ENTRY_DATATYPE_  long *dataType  /* out */
ENTRY_NUMELEMENTS_  long *numElements  /* out */
VAR_ALLOCATEDFROM_  long startRecord  /* in */
long *nextRecord  /* out */
VAR_ALLOCATEDTO_  long startRecord  /* in */
long *lastRecord  /* out */
VAR_BLOCKINGFACTOR_  long *blockingFactor  /* out */
VAR_COMPRESSION_  long *cType  /* out */
long cParms[CDF_MAX_PARMS]  /* out */
long *cPct  /* out */
VAR_DATA_  void *value  /* out */
VAR_DATATYPE_  long *dataType  /* out */
VAR_DIMVARYS_  long dimVarys[CDF_MAX_DIMS]  /* out */
VAR_HYPERDATA_  void *buffer  /* out */
VAR_MAXALLOCREC_  long *maxRec  /* out */
VAR_MAXREC_  long *maxRec  /* out */
VAR_NAME_  char varName[CDF_VAR_NAME_LEN256+1]  /* out */
VAR_NUMINDEXENTRIES_  long *numEntries  /* out */
VAR_NUMINDEXLEVELS_  long *numLevels  /* out */
rVAR_nINDEXRECORDS_ long *numRecords /* out */
rVAR_NUMallocRECS_ long *numRecords /* out */
rVAR_NUMBER_ char *varName /* in */
long *varNum /* out */
rVAR_NUMELEMS_ long *numElements /* out */
rVAR_NUMRECS_ long *numRecords /* out */
rVAR_PADVALUE_ void *value /* out */
rVAR_RECVARY_ long *recVary /* out */
rVAR_SEQDATA_ void *value /* out */
rVAR_SPARSEARRAYS_ long *sArraysType /* out */
long sArraysParms[CDF_MAX_PARMS] /* out */
long *sArraysPct /* out */
rVAR_SPARSERECORDS_ long *sRecordsType /* out */
rVARs_DIMSIZES_ long *maxRec /* out */
rVARs_MAXREC_ long *maxRec /* out */
rVARs_NUMDIMS_ long *numDim /* out */
rVARs_RECDATA_ long numVNum /* in */
long varNums[] /* in */
void *buffer /* out */
STATUS_TEXT_ char text[CDF_STATUSTEXT_LEN+1] /* out */
zENTRY_DATA_ void *value /* out */
zENTRY_DATATYPE_ long *dataType /* out */
zENTRY_NUMELEMS_ long *numElements /* out */
zVAR_ALLOCATEDFROM_ long startRecord /* in */
long *nextRecord /* out */
zVAR_ALLOCATEDTO_ long startRecord /* in */
long *lastRecord /* out */
zVAR_BLOCKINGFACTOR_ long *blockingFactor /* out */
zVAR_COMPRESSION_ long *cType /* out */
long cParms[CDF_MAX_PARMS] /* out */
long *cPct /* out */
zVAR_DATA_ void *value /* out */
zVAR_DATATYPE_ long *dataType /* out */
zVAR_DIMSIZES_ long *maxRec /* out */
zVAR_DIMVARYS_ long *maxRec /* out */
zVAR_HYPERDATA_ void *buffer /* out */
zVAR_MAXallocRECS_ long *maxRec /* out */
zVAR_MAXREC_ long *maxRec /* out */
zVAR_NAME_ char *varName[CDF_VAR_NAME_LEN256+1] /* out */
zVAR_nINDEXENTRIES_ long *numEntries /* out */
zVAR_nINDEXLEVELS_ long *numLevels /* out */
zVAR_nINDEXRECORDS_ long *numRecords /* out */
zVAR_NUMallocRECS_ long *numRecords /* out */
zVAR_NUMBER_ char *varName /* in */
long *varNum /* out */
zVAR_NUMDIMS_ long *numDim /* out */
zVAR_NUMELEMS_ long *numElements /* out */
zVAR_NUMRECS_ long *numRecords /* out */
zVAR_PADVALUE_ void *value /* out */
zVAR_RECVARY_ long *recVary /* out */
zVAR_SEQDATA_ void *value /* out */
zVAR_SPARSEARRAYS_ long *sArraysType /* out */
long sArraysParms[CDF_MAX_PARMS] /* out */
long *sArraysPct /* out */
zVAR_SPARSERECORDS_ long *sRecordsType /* out */
zVARs_MAXREC_ long *maxRec /* out */
zVARs_RECDATA_  long numVars  /* in */
    long varNums[]  /* in */
    void *buffer  /* out */

NULL_

OPEN_  CDF_  char *CDFname  /* in */
    CDFid *id  /* out */

PUT__  ATTR_NAME_  char *attrName  /* in */
    ATTR_SCOPE_  long scope  /* in */
    CDF_CHECKSUM_  long checksum  /* in */
    CDF_COMPRESSION_  long cType  /* in */
        long cParms[]  /* in */
    CDF_ENCODING_  long encoding  /* in */
    CDF_FORMAT_  long format  /* in */
    CDF_MAJORITY_  long majority  /* in */
    gENTRY_DATA_  long dataType  /* in */
        long numElements  /* in */
        void *value  /* in */
    gENTRY_DATASPEC_  long dataType  /* in */
        long numElements  /* in */
        void *value  /* in */
    rENTRY_DATA_  long dataType  /* in */
        long numElements  /* in */
        void *value  /* in */
    rENTRY_DATASPEC_  long dataType  /* in */
        long numElements  /* in */
    rVAR_ALLOCATEBLOCK_  long firstRecord  /* in */
        long lastRecord  /* in */
    rVAR_ALLOCATERECS_  long numRecords  /* in */
    rVAR_BLOCKINGFACTOR_  long blockingFactor  /* in */
    rVAR_COMPRESSION_  long cType  /* in */
        long cParms[]  /* in */
    rVAR_DATA_  void *value  /* in */
    rVAR_DATASPEC_  long dataType  /* in */
        long numElements  /* in */
    rVAR_DIMVARYS_  long dimVarys[]  /* in */
    rVAR_HYPERDATA_  void *buffer  /* in */
    rVAR_INITIALRECS_  long nRecords  /* in */
    rVAR_NAME_  char *varName  /* in */
    rVAR_PADVALUE_  void *value  /* in */
    rVAR_RECVARY_  long recVary  /* in */
    rVAR_SEQDATA_  void *value  /* in */
    rVAR_SPARSEARRAYS_  long sArraysType  /* in */
        long sArraysParms[]  /* in */
    rVAR_SPARSERECORDS_  long sRecordsType  /* in */
    rVARs_RECDATA_  long varNums[]  /* in */
        void *buffer  /* in */
    zENTRY_DATA_  long dataType  /* in */
        long numElements  /* in */
        void *value  /* in */
    zENTRY_DATASPEC_  long dataType  /* in */
        long numElements  /* in */
        void *value  /* in */
    zVAR_ALLOCATEBLOCK_  long firstRecord  /* in */
        long lastRecord  /* in */
zVAR_ALLOCATERECS_ long numRecords /* in */
zVAR_BLOCKINGFACTOR_ long blockingFactor /* in */
zVAR_COMPRESSION_ long cType /* in */
long eParms[] /* in */
zVAR_DATA_ void *value /* in */
zVAR_DATASPEC_ long dataType /* in */
long numElements /* in */
zVAR_DIMVARYS_ long dimVarys[] /* in */
zVAR_INITIALRECS_ long nRecords /* in */
zVAR_HYPERDATA_ void *buffer /* in */
zVAR_NAME_ char *varName /* in */
zVAR_PADVALUE_ void *value /* in */
zVAR_RECVARY_ long recVary /* in */
zVAR_SEQDATA_ void *value /* in */
zVAR_SPARSEARRAYS_ long sArraysType /* in */
long sArraysParms[] /* in */
zVAR_SPARSERECORDS_ long sRecordsType /* in */
zVARs_RECDATA_ long numVars /* in */
long varNums[] /* in */
void *buffer /* in */
SELECT_
ATTR_ long attrNum /* in */
ATTR_NAME_ char *attrName /* in */
CDF_ CDFid id /* in */
CDF_CACHESIZE_ long numBuffers /* in */
CDF_DECODING_ long decoding /* in */
CDF_NEGtoPOSfp0_MODE_ long mode /* in */
CDF_READONLY_MODE_ long mode /* in */
CDF_SCRATCHDIR_ char *dirPath /* in */
CDF_STATUS_ CDFstatus status /* in */
CDF_zMODE_ long mode /* in */
COMPRESS_CACHESIZE_ long numBuffers /* in */
gENTRY_ long entryNum /* in */
rENTRY_ long entryNum /* in */
rENTRY_NAME_ char *varName /* in */
rVAR_ long varNum /* in */
rVAR_CACHESIZE_ long numBuffers /* in */
rVAR_NAME_ char *varName /* in */
rVAR_RESERVEPERCENT_ long percent /* in */
rVAR_SEQPOS_ long recNum /* in */
long indices[] /* in */
rVARs_CACHESIZE_ long numBuffers /* in */
rVARs_DIMCOUNTS_ long counts[] /* in */
rVARs_DIMINDICES_ long indices[] /* in */
rVARs_DIMINTERVALS_ long intervals[] /* in */
rVARs_RECCOUNT_ long recCount /* in */
rVARs_RECINTERVAL_ long recInterval /* in */
rVARs_RECNUMBER_ long recNum /* in */
STAGE_CACHESIZE_ long numBuffers /* in */
zENTRY_ long entryNum /* in */
zENTRY_NAME_ char *varName /* in */
zVAR_ long varNum /* in */
zVAR_CACHESIZE_ long numBuffers /* in */
zVAR_DIMCOUNTS_ long counts[] /* in */
zVAR_DIMINDICES_ long indices[] /* in */
zVAR_DIMINTERVALS_ long intervals[] /* in */
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zVAR_NAME_</td>
<td>char *varName</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVAR_RECCOUNT_</td>
<td>long recCount</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVAR_RECINTERVAL_</td>
<td>long recInterval</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVAR_RECNUMBER_</td>
<td>long recNum</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVAR_RESERVEPERCENT_</td>
<td>long percent</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVAR_SEQPOS_</td>
<td>long recNum</td>
<td>/* in */</td>
</tr>
<tr>
<td></td>
<td>long indices[]</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVARs_CACHESIZE_</td>
<td>long numBuffers</td>
<td>/* in */</td>
</tr>
<tr>
<td>zVARs_RECNUMBER_</td>
<td>long recNum</td>
<td>/* in */</td>
</tr>
</tbody>
</table>
B.4 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 */

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 */

void encodeEPOCH3 (epoch, epString)
double epoch; /* in */
char epString[EPOCH3_STRING_LEN+1]; /* out */

void encodeEPOCH4 (epoch, epString)
double epoch; /* in */
char epString[EPOCH4_STRING_LEN+1]; /* out */

void encodeEPOCHx (epoch, format, epString)
double epoch; /* in */
char format[EPOCHx_FORMAT_MAX+1]; /* in */
char epString[EPOCHx_STRING_MAX+1]; /* out */

double parseEPOCH (epString)
char epString[EPOCH_STRING_LEN+1]; /* in */

double parseEPOCH1 (epString)
char epString[EPOCH1_STRING_LEN+1]; /* in */

double parseEPOCH2 (epString)
char epString[EPOCH2_STRING_LEN+1]; /* in */
double parseEPOCH3 (epString)
char epString[EPOCH3_STRING_LEN+1]; /* in */
double parseEPOCH4 (epString)
char epString[EPOCH4_STRING_LEN+1]; /* in */

double computeEPOCH16 (year, month, day, hour, minute, second, msec, microsec, nanosec, picosec)
long year; /* in */
long month; /* in */
long day; /* in */
long hour; /* in */
long minute; /* in */
long second; /* in */
long msec; /* in */
long microsec; /* in */
long nanosec; /* in */
long picosec; /* in */
double epoch[2]; /* out */

void EPOCH16breakdown (epoch, year, month, day, hour, minute, second, msec, microsec, nanosec, picosec)
double epoch[2]; /* in */
long *year; /* out */
long *month; /* out */
long *day; /* out */
long *hour; /* out */
long *minute; /* out */
long *second; /* out */
long *msec; /* out */
long *microsec; /* out */
long *nanosec; /* out */
long *picosec; /* out */

void encodeEPOCH16 (epoch, epString)
double epoch[2]; /* in */
char epString[EPOCH16_STRING_LEN+1]; /* out */

void encodeEPOCH16_1 (epoch, epString)
double epoch[2]; /* in */
char epString[EPOCH16_1_STRING_LEN+1]; /* out */

void encodeEPOCH16_2 (epoch, epString)
double epoch[2]; /* in */
char epString[EPOCH16_2_STRING_LEN+1]; /* out */

void encodeEPOCH16_3 (epoch, epString)
double epoch[2]; /* in */
char epString[EPOCH16_3_STRING_LEN+1]; /* out */

void encodeEPOCH16_4 (epoch, epString)
double epoch[2]; /* in */
char epString[EPOCH16_4_STRING_LEN+1]; /* out */

void encodeEPOCH16_x (epoch, format, epString)
double epoch[2]; /* in */
char format[EPOCHx_FORMAT_MAX+1]; /* in */
char epString[EPOCHx_STRING_MAX+1]; /* out */

double parseEPOCH16 (epString, epoch)
char epString[EPOCH16_STRING_LEN+1]; /* in */
double epoch[2]; /* out */

double parseEPOCH16_1 (epString)
char epString[EPOCH16_1_STRING_LEN+1]; /* in */
double epoch[2]; /* out */

double parseEPOCH16_2 (epString)
char epString[EPOCH16_2_STRING_LEN+1]; /* in */
double epoch[2]; /* out */

double parseEPOCH16_3 (epString)
char epString[EPOCH16_3_STRING_LEN+1]; /* in */
double epoch[2]; /* out */

double parseEPOCH16_4 (epString)
char epString[EPOCH16_4_STRING_LEN+1]; /* in */
double epoch[2]; /* out */

B.5 TT2000 Utility Routines

CDF_TT2000_from.UTC_parts OR computeTT2000
long long computeTT2000 (year, month, day, …) (*Variable argument form)
double year; /* in */
double month; /* in */
double day; /* in */
…
TT2000NULL; /* in */

long long computeTT2000 (year, month, day, hour, minute, second, msec, usec, nsec) (*Full form)
double year; /* in */
double month; /* in */
double day; /* in */
double hour; /* in */
double minute; /* in */
double second; /* in */
double msec; /* in */
double usec; /* in */
double nsec; /* in */

CDF_TT2000_to.UTC_parts OR TT2000breakdown
void TT2000breakdown (tt2000, year, month, day, …)47
long long tt2000; /* in */
double *year; /* out */
double *month; /* out */
double *day; /* out */
…
TT2000NULL; /* in */

47 Variable argument list form after the day field. But, need to have TT2000NULL to indicate the end of the list.
void TT2000breakdown (tt2000, year, month, day, hour, minute, second, msec, usec, nsec)48
long long tt2000; /* in */
double *year; /* out */
double *month; /* out */
double *day; /* out */
double *hour; /* out */
double *minute; /* out */
double *second; /* out */
double *msec; /* out */
double *usec; /* out */
double *nsec; /* out */

CDF_TT2000_to_UTC_string OR encodeTT2000
void encodeTT2000 (tt2000, epString) (*Variable argument form)
long long tt2000; /* in */
char *epString; /* out */

void encodeTT2000 (tt2000, epString, form) (*Full form)
l long long tt2000; /* in */
char *epString; /* out */
int form; /* in */

CDF_TT2000_from_UTC_string OR parseTT2000
long long parseTT2000 (epString)
char *epString; /* in */

long CDF_TT2000_from_UTC_EPOCH (epoch)
double epoch; /* in */
long CDF_TT2000_from_UTC_EPOCH16 (epoch16)
double *epoch16; /* in */
double CDF_TT2000_to_UTC_EPOCH (tt2000)
long long tt2000; /* in */
void CDF_TT2000_to_UTC_EPOCH16 (tt2000, epoch16)
long long tt2000; /* in */
double *epoch16; /* out */

48 Full list form
Index

ALPHAOSF1_DECODING, 14
ALPHAOSF1_ENCODING, 13
ALPHAVMSd_DECODING, 14
ALPHAVMSd_ENCODING, 13
ALPHAVMSg_DECODING, 14
ALPHAVMSg_ENCODING, 13
ALPHAVMSi_DECODING, 14
ALPHAVMSi_ENCODING, 13
attribute
  inquiring, 29
  number
    inquiring, 31
  renaming, 33
Attributes
  entries
    global entry
      deleting, 166
      reading, 168
  attributes
    checking existence, 161
    creating, 25, 164, 219
    current, 206
    confirming, 211
    selecting
      by name, 257
      by number, 257
    deleting, 165, 222
    entries
      rVariable entry
        deleting, 167
      entries
        current, 206
        confirming, 213, 214, 217
        selecting
          by name, 259, 262
          by number, 259, 262
      data specification
        changing, 248, 252
        data type
          inquiring, 229, 231, 238
          number of elements
            inquiring, 230, 231, 238
          deleting, 222, 223
          existence, determining, 214, 217
        global entries
          number of
            inquiring, 184
        global entry
          checking existence, 161
          data specification
            resetting, 198
          data type
            inquiring, 170, 177
            inquiring, 190
          last entry number
            inquiring, 173
            number of elements
            inquiring, 171, 178
            writing, 194
            inquiring, 26
            maximum
              inquiring, 224
              number of
                inquiring, 225
                reading, 28, 229, 231, 238
            rEntries
              number of
                inquiring, 185
            rVariable entry
              checking existence, 162
              data specification
                resetting, 199
                inquiring, 192
                last entry number
                  inquiring, 174
                reading, 172
                writing, 195
                writing, 32, 246, 247, 252
            zEntries
              number of
                inquiring, 186
            zVariable entry
              checking existence, 163
              data specification
                resetting, 201
              data type
                inquiring, 182
                deleting, 168
                inquiring, 193
                last entry number
                  inquiring, 175
                  number of elements
                    inquiring, 183
                    reading, 180
                    writing, 197
                existence, determining, 211
                inquiring, 189
              name
                inquiring, 176
                naming, 19, 26, 164
                inquiring, 30, 225
                renaming, 245
                number
                  inquiring, 177
                  number of
                    inquiring, 42, 185, 228
                  numbering
                    inquiring, 225
                  renaming, 198
                  scope
                    inquiring, 179
                    resetting, 200
                  scopes
                    changing, 245

333
constants, 17
GLOBAL SCOPE, 17
VARIABLE_SCOPE, 17
inquiring, 30, 189, 226

CDF
backward file, 19
backward file flag
getting, 20
setting, 19
cache size
compression
resetting, 86
Checksum, 20
Checksum mode
setting, 21, 22
closing, 34
Copyright
inquiring, 72
creating, 35
deleting, 36, 67
Long Integer, 23
opening, 44, 83
set
majority, 90
Validation, 22
CDF getNegtoPosfp0Mode, 77
CDF library
copy right notice
max length, 19
reading, 230
Extended Standard Interface, 61
Internal interface, 203
modes
-0.0 to 0.0
confirming, 212
constants
NEGtoPOSfp0off, 18
NEGtoPOSfp0on, 18
selecting, 258
decoding
confirming, 212
constants
ALPHAOASF1_DECODING, 14
ALPHAVMSd_DECODING, 14
ALPHAVMSg_DECODING, 14
ALPHAVMSi_DECODING, 14
DECASTATION_DECODING, 14
HOST_DECODING, 14
HP_DECODING, 15
IBMRS_DECODING, 14
MAC_DECODING, 15
NETWORK_DECODING, 14
NeXT_DECODING, 15
PC_DECODING, 15
SGI_DECODING, 14
SUN_DECODING, 14
VAX_DECODING, 14
selecting, 258
read-only
confirming, 212
constants
READONLYoff, 17
READONLYon, 17
selecting, 17, 258
zMode
confirming, 213
constants
zMODEoff, 18
zMODEon1, 18
zMODEon2, 18
selecting, 18, 258
Original Standard Interface, 25
shared CDF library, 7
version
inquiring, 230
CDF setNegtoPosfp0Mode, 91
CDF_ATTR_NAME_LEN, 19
CDF_BYTE, 12
CDF_CHAR, 12
CDF_COPYRIGHT_LEN, 19
CDF_DOUBLE, 12
CDF_EPOCH, 12
CDF_EPOCH16, 12
CDF_error or CDFerror, 293
CDF_FLOAT, 12
CDF_INC, 2
CDF_INT1, 12
CDF_INT2, 12
CDF_INT4, 12
CDF_INT8, 12
CDF_LIB, 5
CDF_MAX_DIMS, 18
CDF_MAX_PARMS, 18
CDF_OK, 11
CDF_PATHNAME_LEN, 18
CDF_REAL4, 12
CDF_REAL8, 12
CDF_STATUSTEXT_LEN, 19
CDF_TIME_TT2000, 13
CDF_TT2000_from.UTC_EPOCH, 290
CDF_TT2000_from.UTC_EPOCH16, 291
CDF_TT2000_from.UTC_parts, 287
CDF_TT2000_from.UTC_string, 290
CDF_TT2000_to.UTC_EPOCH, 291
CDF_TT2000_to.UTC_EPOCH16, 291
CDF_TT2000_to.UTC_parts, 289
CDF_TT2000_to.UTC_string, 289
CDF_UCHAR, 12
CDF_UINT1, 12
CDF_UINT2, 12
CDF_UINT4, 12
CDF_VAR_NAME_LEN, 19
CDF_WARN, 12
cdf.h, 1, 11
cdfsinc, 1
cfslib, 5
cdfattrCreate, 25
cdfattrEntryInquire, 26
cdfattrGet, 28
cdfattrInquire, 29
cdfattrNum, 31
cdfattrPut, 32
cdfattrRename, 33
CDFclose, 34
CDFrenameAttr, 198
CDFrenamezVar, 148
CDFs
  compression
    inquiring, 69, 71
CDFs
  -0.0 to 0.0 mode
    inquiring, 77
    resetting, 91
accessing, 211
browsing, 17
cache buffers
  confirming, 211, 213, 214, 216, 217
  selecting, 257, 259, 260, 262, 264
cache size
  compression
    inquiring, 70
    inquiring, 67
    resetting, 84
  stage
    inquiring, 78
    resetting, 92
checksum
  inquiring, 68, 226
  resetting, 84
  specifying, 245
closing, 65, 210
compression
  inquiring, 226, 232, 239
  resetting, 85
  specifying, 246
compression types/parameters, 16
copy right notice
  max length, 19
  reading, 37, 226
corrupted, 35, 66
creating, 65, 220
current, 205
  confirming, 211
  selecting, 257
decoding
  inquiring, 72, 73
  resetting, 87
deleting, 222
encoding
  changing, 246
  constants, 13
    ALPHAOSF1_ENCODING, 13
    ALPHAVMSd_ENCODING, 13
    ALPHAVMSg_ENCODING, 13
    ALPHAVMSi_ENCODING, 13
    DECSTATION_ENCODING, 13
    HOST_ENCODING, 13
    HP_ENCODING, 14
    IBMRS_ENCODING, 13
    MAC_ENCODING, 14
    NETWORK_ENCODING, 13
    NeXT_ENCODING, 14
    PC_ENCODING, 14
    SGI_ENCODING, 13
    SUN_ENCODING, 13
    VAX_ENCODING, 13
default, 13
  inquiring, 42, 226
  resetting, 88
file backward
  inquiring, 74
File Backward
  resetting, 88
format
  changing, 246
  constants
    MULTI_FILE, 12
    SINGLE_FILE, 12
default, 12
  inquiring, 74
  inquiring, 227
  resetting, 89
global attributes
  number of
    inquiring, 187
  inquiring, 81
majority
  inquiring, 75
name
  inquiring, 76
naming, 18, 35, 44, 66, 83
nulling, 245
opening, 245
overwriting, 35, 66
read-only mode
  inquiring, 77
  resetting, 91
record number
  maximum written for zVariables and rVariables, 101
rVariables
  number of
    inquiring, 101
scratch directory
  specifying, 258
validation
  inquiring, 79
  resetting, 93
variable attributes
  number of
    inquiring, 188
version
  inquiring, 37, 80, 227, 229
zMode
  inquiring, 80
  resetting, 93
zVariables
  number of
    inquiring, 102
CDFsetAttrgEntryDataSpec, 198
CDFsetAttrrEntryDataSpec, 199
CDFsetAttrScope, 200
CDFsetAttrzEntryDataSpec, 201
CDFsetCacheSize, 84
CDFsetChecksum, 84
CDFsetCompression, 85
CDFsetCompressionCacheSize, 86
CDFsetDecoding, 87
CDFsetEncoding, 88
CDFsetFileBackward, 88
CDFsetFormat, 89
CDFsetMajority, 90
CDFsetReadOnlyMode, 91
CDFsetStageCacheSize, 92
CDFsetValidate, 93
CDFsetzMode, 93
CDFsetzVarAllocBlockRecords, 149
CDFsetzVarAllocRecords, 150
CDFsetzVarBlockingFactor, 150
CDFsetzVarCacheSize, 151
CDFsetzVarCompression, 152
CDFsetzVarDataSpec, 153
CDFsetzVarDimVariances, 154
CDFsetzVarInitialRecs, 155
CDFsetzVarPadValue, 156
CDFsetzVarRecVariance, 156
CDFsetzVarResizePercent, 157
CDFsetzVarsCacheSize, 158
CDFsetzVarSeqPos, 159
CDFsetzVarSparseRecords, 160
CDFstatus, 11
CDFvarClose, 48
CDFvarCreate, 49
CDFvarGet, 51
CDFvarHyperGet, 52
CDFvarHyperPut, 53
CDFvarInquire, 54
CDFvarNum, 56
CDFvarPut, 57
CDFvarRename, 58
checksum
CDFspecifying, 245
Checksum, 68, 84
closing
zVar in a multi-file CDF, 94
COLUMN_MAJOR, 15
Compiling, 1
compression
CDF
inquiring, 226, 227
specifying, 246
types/parameters, 16
variables
inquiring, 232, 239
reserve percentage
confirming, 215, 219
selecting, 260, 264
specifying, 249, 253
computeEPOCH, 275
computeEPOCH16, 280
computeTT2000, 287
Data type
size
inquiring, 61
data types
constants, 12
CDF_BYTE, 12
CDF_CHAR, 12
CDF_DOUBLE, 12
CDF_EPOCH, 12
CDF_EPOCH16, 12
CDF_FLOAT, 12
CDF_INT1, 12
CDF_INT2, 12
CDF_INT4, 12
CDF_INT8, 12
CDF_REAL4, 12
CDF_REAL8, 12
CDF_TIME_TT2000, 13
CDF_UCHAR, 12
CDF_UINT1, 12
CDF_UINT2, 12
CDF_UINT4, 12
inquiring size, 229
DECESTATION_DECODING, 14
DECESTATION_ENCODING, 13
definitions file, 1
DEFINITIONS.COM, 1, 5
dimensions
limit, 18
encodeEPOCH, 276
encodeEPOCH1, 277
encodeEPOCH16, 281
encodeEPOCH16_1, 281
encodeEPOCH16_2, 281
encodeEPOCH16_3, 282
encodeEPOCH16_4, 282
encodeEPOCH16_x, 282
encodeEPOCH2, 277
encodeEPOCH3, 277
encodeEPOCH4, 277
encodeEPOCHx, 278
encodeTT2000, 289
EPOCH
computing, 275, 280
decomposing, 276, 280
encoding, 276, 277, 278, 281, 282
ISO 8601, 277, 280, 282, 284
parsing, 279, 280, 283, 284
utility routines, 275
computeEPOCH, 275
computeEPOCH16, 280
encodeEPOCH, 276
encodeEPOCH1, 277
encodeEPOCH16, 281
encodeEPOCH16_1, 281
encodeEPOCH16_2, 281
encodeEPOCH16_3, 282
encodeEPOCH16_4, 282
encodeEPOCH16_x, 282
codeEPOCH2, 277
codeEPOCH3, 277
codeEPOCH4, 277
codeEPOCHx, 278
codeTT2000, 289
EPOCH16breakdown, 280
EPOCHbreakdown, 280
parseEPOCH, 279
parseEPOCH1, 279
parseEPOCH16, 283
parseEPOCH16_1, 284
parseEPOCH16_2, 284
parseEPOCH16_3, 284
-0.0 to 0.0 mode
attribute
delete, 165
attribute
create, 165
data scope
set, 201
eexistence
confirm, 161
information
get, 189
number
get, 177
rename, 198
cache buffer size
get, 78
cache size
get, 68
set, 84
checksum
get, 68
close, 65
compression
set, 86
compression cache size
get, 70
compression information
get, 71
create, 66
decoding
set, 87
delete, 67
file backward
get, 74
format
get, 75
set, 89
gentry
existence
confirm, 162
global attribute
entry
delete, 166
global attribute
entry
information
get, 191
gentry
specification
set, 199
write, 195
last Entry number
get, 174
majority
get, 76
set, 90
max record numbers
zVariables and rVariables
get, 101
name
get, 76
number of global attributes
get, 187
number of rVariables
get, 102
number of variable attributes
get, 188
number of zVariables
get, 102
open, 83
read-only mode
get, 78
rEntry
existence
confirm, 163
rVariable attribute
try
delete, 167
rVariable attribute
entry
information
get, 192
entry
number of elements
get, 179
specification
set, 200
write, 196
last Entry number
get, 174
number of entries
get, 186
Variable
all records
get, 103
put, 139
range records
get, 106
put, 140
Variable number
get, 105
zEntry
existence
confirm, 164
zVar
close, 95
zVariable
data records
delete, 100
existence
confirm, 96
pad value existence
confirm, 96
zVariable
all records
get, 108
put, 141, 143
blocking factor
get, 110
set, 151
cache size
get, 111
set, 152, 158
compression
get, 112
set, 153
compression reserve percentage
get, 125
set, 158
create, 98
data records
block
allocate, 149
sequential
allocate, 150
data type
get, 114
set, 153
data value
write, 142
data value
sequential write, 145
data value
get, 126
data values
write, 133
delete, 99
dimension sizes
get, 115
dimension variances
get, 115
set, 154
dimensionality
get, 118
inquire, 135
maximum number of records allocated
get, 116
maximum record number
get, 117
multiple values or records
get, 132
name
get, 118
number of elements
get, 119
number of initial records
set, 155
number of records allocated
get, 107
number of records written
get, 120
pad value
get, 121
set, 156
range records
get, 122
read position
get, 127
record data
get, 123
write, 144
record variance
get, 124
set, 157
rename, 148
sequential location set, 159
sparse record flag set, 160
sparse record type get, 128
variable data get, 113
zVariable attribute entry delete, 168
zVariable attribute entry get, 181
 entry data type get, 182
 information get, 193
 number of elements get, 183
 specification set, 202
 write, 197
 last entry number get, 175
 number of entries get, 187
zVariables
 maximum record number get, 128
 record data write, 147
 record data get, 129
closing
 CDF, 34
 rVariable, 49
creating
 attribute, 26
 CDF, 36, 203
 rVariable, 50, 265
 zVariable, 265
deleting
 CDF, 37
get
 CDF
 Copyright, 62
 library version, 63
data type size, 62
rVariable
 data, 51
inquiring
 attribute, 30
 entry, 27
 attribute number, 31
 CDF, 38, 43
 format, 271
 error code explanation text, 38, 64
 rVariable, 55
 variable number, 56
Internal Interface, 203, 265
interpreting
 status codes, 273
 opening
 CDF, 44
read
 multiple zVariables’ data, 41
 reading
 attribute entry, 29
 rVariable values
 hyper, 52, 266
 rVariables full record, 40
 zVariable values
 sequential, 267
renaming
 attribute, 34
 attributes, 267
 rVariable, 58
 rVariables
 inserting records, 136, 138
 status handler, 273
 Variables
 inserting records, 137
writing
 attribute
 gEntry, 32
 rEntry, 32, 268
 rVariable
 multiple records/values, 54
 rVariable, 57
 rVariables, 45
 rVariables full record, 45
 zVariable full record, 47
 zVariable values
 multiple variable, 269
Extended Standard Interface, 61
function prototypes, 25, 61
getAttrgEntryNumElements, 171
getAttrMaxgEntry, 173
GLOBAL_SCOPE, 17
HOST_DECODING, 14
HOST_ENCODING, 13
HP_DECODING, 14
HP_ENCODING, 13
IBMRS_DECODING, 14
IBMRS_ENCODING, 13
include files, 1
inquiring
 CDF information, 37
 Interfaces
 Extended Standard, 61
 Internal, 203
 Original Standard, 25
 Internal Interface, 203
current mistakes, 270
 current objects/states, 205
 attribute, 206
 attribute entries, 206
 CDF, 205
 records/dimensions, 206, 207, 208
 sequential value, 207, 208
 status code, 208
 variables, 206
examples, 203, 265
Indentation/Style, 209
Operations, 210
status codes, returned, 209
syntax, 209
argument list, 210
limitations, 210

libcdf.a, 5
libcdf.lib, 6
LIBCDF.OLB, 5
Library
  error text
    inquiring, 64
Library
  Copyright
    inquiring, 62
  version
    inquiring, 63
limits
  attribute name, 19
  Copyright text, 19
dimensions, 18
explanation/status text, 19
file name, 18
parameters, 18
variable name, 19
Limits of names, 18
linking, 5
  shareable CDF library, 7
MAC_DECODING, 15
MAC_ENCODING, 14
MULTI_FILE, 12
NEGtoPOSfp0off, 18
NEGtoPOSfp0on, 18
NETWORK_DECODING, 14
NETWORK_ENCODING, 13
NeXT_DECODING, 15
NeXT_ENCODING, 14
NO_COMPRESSION, 16
NO_SPARSEARRAYS, 17
NO_SPARSERECORDS, 17
NOVARY, 15
Original Standard Interface, 25
PAD_SPARSERECORDS, 17
parseEPOCH, 279
parseEPOCH1, 279
parseEPOCH16, 283
parseEPOCH16_1, 284
parseEPOCH16_2, 284
parseEPOCH16_3, 284
parseEPOCH16_4, 284
parseEPOCH2, 279
parseEPOCH3, 279
parseEPOCH4, 280
parseTT2000, 290
PC_DECODING, 15
PC_ENCODING, 14
PREV_SPARSERECORDS, 17
programming interface
customizing, 270
typedef’s, 11
CDFfid, 11

CDFstatus, 11
reading
  multiple rVariables’ data, 39
  multiple zVariables’ data, 41
READONLYoff, 17
READONLYon, 17
ROW_MAJOR, 15
rVariables
  close, 48
  creating, 49
full record
  reading, 39
  writing, 45
hyper values
  accessing, 52
  writing, 53
insetting records, 135
renaming, 58
single value
  accessing, 51
  writing, 57
scratch directory
  specifying, 258
SGi_DECODING, 14
SGi_ENCODING, 13
SINGLE_FILE, 12
sparse arrays
  inquiring, 236, 243
  specifying, 251, 256
types, 17
sparse records
  inquiring, 236, 244
  specifying, 251, 256
types, 17
status codes
  constants, 11, 273
    CDF_OK, 11
    CDF_WARN, 12
current, 208
  confirming, 212
  selecting, 258
error, 293
explanation text
  inquiring, 38, 238
  max length, 19
informational, 293
interpreting, 273
status handler, example, 269
warning, 293
SUN_DECODING, 14
SUN_ENCODING, 13
TT2000
  computing, 287
  conversion, 290, 291
decomposing, 288
encoding, 289
encoding, 289
  parsing, 290
utility routines, 287
  CDF_TT2000_from.UTC_EPOCH, 290
  CDF_TT2000_from.UTC_EPOCH16, 291
  CDF_TT2000_from.UTC_parts, 287
  CDF_TT2000_from.UTC_string, 290
variables
  closing, 211
  compression
    confirming, 215, 219
    inquiring, 226, 232, 239
    selecting, 260, 264
    specifying, 249, 253
  types/parameters, 16
  creating, 220, 221
  current, 206
    confirming, 214, 217
    selecting
      by name, 260, 263
      by number, 259, 262
  data specification
    changing, 249, 254
    data type
      inquiring, 54, 233, 240
    number of elements
      inquiring, 54, 235, 242
  deleting, 223
  dimension counts
    current, 207, 208
    confirming, 215, 217
    selecting, 261, 263
  dimension indices, starting
    current, 207, 208
    confirming, 216, 218
    selecting, 261, 263
  dimension intervals
    current, 207, 208
    confirming, 216, 218
    selecting, 261, 263
  dimensionality
    inquiring, 42, 237, 242
  existence, determining, 215, 218
  insetting records, 137
  majority
    changing, 246
    considering, 15
    constants, 15
    COLUMN_MAJOR, 15
    ROW_MAJOR, 15
  default, 220
  inquiring, 228
  naming, 50, 97
    inquiring, 54, 234, 241
  max length, 19
  renaming, 250, 255
  number
    inquiring, 56, 104
    number of
      inquiring, 42
      number of, inquiring, 228
      numbering
        inquiring, 235, 242
  pad value
    confirming, 215, 218
    inquiring, 235, 243
    specifying, 250, 255
  read range records, 106
  reading, 232, 233, 239, 240
  record count
    current, 207
    confirming, 216, 218
    selecting, 261, 263
  record interval
    current, 207, 208
    confirming, 216, 218
    selecting, 261, 263
  record number, starting
    current, 206, 207
    confirming, 216, 219
    selecting, 261, 264
  records
    allocated
      inquiring, 231, 232, 234, 238, 239, 242
      specifying, 248, 253
    blocking factor
      inquiring, 232, 239
      specifying, 248, 253
    deleting, 223, 224
    indexing
      inquiring, 234, 241
      initial
        writing, 250, 254
    maximum
      inquiring, 42, 233, 237, 241, 244
      number of
        inquiring, 235, 243
        sparse, 17
        inquiring, 236, 244
        specifying, 251, 256
    sparse arrays
      inquiring, 236, 243, 251, 256
      types, 17
  variances
    constants, 15
      NOVARY, 15
      VARY, 15
    dimensional
      inquiring, 233, 240
      specifying, 249, 254
    record
      changing, 251, 255
      inquiring, 235, 243
      write range records, 140
      writing, 250, 255
    VARY, 15
    VAX_DECODING, 14
    VAX_ENCODING, 13
  variables
    read all records, 103
    zMODEoff, 18
    zMODEon1, 18
    zMODEon2, 18
    zVariables
      data records
deleting, 100
zVariables
blocking factor
inquiring, 109
resetting, 150
cache size
inquiring, 110
resetting, 151, 158
check existence, 95
compression
inquiring, 111
reserve percentage
inquiring, 125
resetting, 157
resetting, 152
creating, 97
data specification
resetting, 153
data type
inquiring, 113
deleting, 99
dimension sizes
inquiring, 114
dimension variances
inquiring, 115
resetting, 154
dimensionality
inquiring, 118
full record
reading, 41
writing, 47
inquiring, 134
insetting records, 138
name
inquiring, 117
number of elements
inquiring, 119
pad value
checking existence, 96
pad value
inquiring, 120
resetting, 156
read all records, 108
read range records, 121
reading data, 112
reading multiple values or records, 131
reading one record, 123
reading record
multiple zVariables, 129
record numbers
allocated records
inquiring, 107
maximum
inquiring, 116
written records
maximum
inquiring, 116
rVariables and zVariables, 127
number of
inquiring, 120
record variance
inquiring, 124
resetting, 156
records
allocation, 149, 150
writing initially, 155
renaming, 148
sequential data
reading one value, 125
sequential position
inquiring, 126
resetting, 159
sparse records type
inquiring, 128
resetting, 160
write all records, 139, 141
write range records, 143
writing
multiple values or records, 132
writing data, 142
writing record
multiple variables, 146
writing record data, 144
writing sequential data, 145