CDF

Perl Reference Manual

Version 3.9.1, September 1, 2023

Space Physics Data Facility
NASA / Goddard Space Flight Center

Contents

Chapter 1

1 Compiling

Since Perl is an interpreter language and its scripts are checked for any syntax error during their execution, there are no separate steps for compilation and linking as other programming languages like C and Fortran.

The Perl-CDF package includes two interfaces: Internal Interface and Standard Interface. The Standard Interface only covers limited functions that deal mainly with the older rVariables and their attributes in the CDF. This interface is mirrored the original functions that are covered in the C's Standard Interface. The Internal Interface, based on the C's Internal Interface, provides a complete suite of CDF functionality.

1.1 How to use the Perl-CDF package

In order to use either one or both interfaces from any Perl script, the search path for the Perl-CDF package must be set up properly. In addition, the Perl-CDF package needs to be imported as well prior to using the either CDF interface. There are two ways to define the search path for the Perl-CDF package. One way is to include the location of the Perl-CDF package at the beginning of a Perl script. The following code illustrates how to define a Perl-CDF package that is installed under /home/cdf/PerlCDF32:

```
use strict;
BEGIN { unshift @INC,'/home/cdf/PerlCDF32/blib/arch',
                     '/home/cdf/PerlCDF32/blib/lib'; }
use CDF;        # Import the CDF module - optional
```

The other way is to define the location of the Perl-CDF package at the command line when invoking the Perl script. The following command is equivalent to the above example:

```
perl -I/home/cdf/PerlCDF32/blib/arch  -I/home/cdf/PerlCDF32/blib/lib  <perl script name>
```

Since the Perl CDF interface uses the shared CDF library, the user has to tell the operating system where to find the shared library. For Linux, DEC Alpha/OSF1, Sun Solaris or SGI, the environment variable **LD_LIBRARY_PATH** must be set to point to the directory that contains the shared CDF library, **libcdf.so**. For example, if the shared CDF library is installed under /usr/local/share/cdf32/lib and you are using the C-shell, enter:

```
setenv LD_LIBRARY_PATH /usr/local/share/cdf32/lib
```

For HP-UX, the shared library is **libcdf.sl**. For IBM RS6000, the library is **libcdf.o**.

For BSD-based Mac OS X, the environment variable is **DYLD_LIBRARY_PATH** that must be set to point to the directory containing the shared library **libcdf.dylib**.

For Windows 9x/NT/2000/XP, similarly, set the **PATH** variable to point to the directory that contains **dllcdf.dll**.

Two Perl test scripts, **testPerlCDFii.pl** and **testPerlCDFsi.pl**, are provided in the distribution. Both use extensive Perl-CDF interface functions: testPerlCDFii.pl tests CDF's Internal Interface functions while testPerlCDFsi.pl tests the Standard Interface functions. They can be used as sample scripts for development.

Chapter 2

2 Programming Interface

2.1 Item Referencing

The following sections describe various aspects of the Perl programming interface for CDF applications. These include the constants that are available to CDF applications written in Perl. These constants are defined in the Perl-CDF package.

Unlike other programming languages (e.g. C, Fortran, Java, etc.), Perl only has three basic data types: scalars, arrays of scalars and hashes of scalars. No other defined data types are needed for any of the Perl-CDF operation items.

For Perl applications, all CDF 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).

2.2 Passing Arguments

For calling Perl-CDF APIs, the arguments are passed by values or references, based on the input or output operation. The general rules for passing the arguments to APIs are:

Input	Normally, for a scalar argument, it is passed by value[1], e.g., $format, if it is sending information to the CDF for an operation (e.g. setting the CDF file format, data type, variable name, compression method, etc.). However, if the scalar is passed in as a data value[2], it is required by design that it be passed by reference, .e.g., \$dataValue, \$padValue, \$entryData, etc. For an argument requiring an array, no matter how many elements in the array, it is always passed by reference, e.g., \@indices.
Output	The argument is passed by reference, e.g., \$format for a scalar or \@indices for an array, if the argument(s) in an operation is to acquire information from the CDF.

[1] The scalar data can be interpreted properly into an integer (of data type long in C) by the CDF library for a non-string data. A string is also a valid scalar data.

[2] A data value is referred as a variable's record data or padded data, or a global or variable attribute's entry data. Its value will be interpreted based upon its data type when the variable or entry is created.

Refer to the two test Perl scripts mentioned above for example. Since Perl doesn't do type checking, it's application developer's responsibility to ensure that proper arguments are being used. For example, an integer data should be passed to an operation that writes the data value to a CDF variable that is defined as CDF_INT4 or CDF_INT2.

2.3 CDF Status Constants

All CDF functions, except CDFvarNum, CDFgetVarNum, CDFattrNum, CDFgetAttrNum, CDFgetFileBackward and CDFgetChecksum functions, return a status code indicating the completion status of the function. The CDFerror function can be used to inquire the meaning of the status code. Appendix A lists the possible status codes along with their explanations. Chapter 5 describes how to interpret status codes.

CDF_OK	A status code indicating the normal completion of a CDF function.
CDF_WARN	Threshold constant for testing severity of non-normal CDF status codes.

Chapter 5 describes how to use these constants to interpret status codes.

2.4 CDF Formats

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

2.5 CDF Data Types

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

CDF_BYTE	1-byte, signed integer.
CDF_CHAR	1-byte, signed character.
CDF_INT1	1-byte, signed integer.
CDF_UCHAR	1-byte, unsigned character.
CDF_UINT1	1-byte, unsigned integer.
CDF_INT2	2-byte, signed integer.
CDF_UINT2	2-byte, unsigned integer.
CDF_INT4	4-byte, signed integer.

CDF_UINT4	4-byte, unsigned integer.
CDF_INT8	8-byte, signed integer.
CDF_REAL4	4-byte, floating point.
CDF_FLOAT	4-byte, floating point.
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point.
CDF_EPOCH	8-byte, floating point.
CDF_EPOCH16	two 8-byte, floating point.
CDF_TIME_TT2000	8-byte, signed integer.

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).

2.6 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).
IBMPC_ENCODING	Indicates Intel i386 data representation.
NeXT_ENCODING	Indicates NeXT data representation.
MAC_ENCODING	Indicates Macintosh data representation.
ARM_LITTLE_ENCODING	Indicates ARM running little-endian data representation.
ARM_BIG_ENCODING	Indicates ARM running big-endian data representation.
IA64VMSi_ENCODING	Indicates Itanium 64 running OpenVMS data representation. Double-precision floating-point values are encoded in IEEE representation.
IA64VMSd_ENCODING	Indicates Itanium 64 running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.
IA64VMSg_ENCODING	Indicates Itanium 64 running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's G_FLOAT 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.)

2.7 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 the values 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).
IBMPC_DECODING	Indicates Intel i386 data representation.
NeXT_DECODING	Indicates NeXT data representation.
MAC_DECODING	Indicates Macintosh data representation.
ARM_LITTLE_DECODING	Indicates ARM running little-endian data representation.
ARM_BIG_DECODING	Indicates ARM running big-endian data representation.
IA64VMSi_DECODING	Indicates Itanium 64 running OpenVMS data representation. Double-precision floating-point values are encoded in IEEE representation.
IA64VMSd_DECODING	Indicates Itanium 64 running OpenVMS data representation. Double-precision floating-point values are encoded in Digital’s D_FLOAT representation.
IA64VMSg_DECODING	Indicates Itanium 64 running OpenVMS data representation. Double-precision floating-point values are encoded in Digital’s G_FLOAT 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.

2.8 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 rVariables 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.

2.9 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.)

2.10 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. Among the available compression types, GZIP provides the best result.

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.[3] 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 provide the most compression but require the most execution time. Values in-between provide varying compromises of these two extremes. 6 normally provides a better balance between compression and execution.

2.11 Sparseness

2.11.1 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.

[3] Disabled for PC running 16-bit DOS/Windows 3.x.

2.11.2 Sparse Arrays

The following types of sparse arrays for variables are supported.[4]

NO_SPARSEARRAYS	No sparse arrays.

2.12 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.)

2.13 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.

2.14 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.

[4] The sparse arrays are not supported and will not be implemented.

zMODEon2	Turns on zMode/2.

2.15 -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.

2.16 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.

2.17 Limits of Names and Other Character Strings

CDF_PATHNAME_LEN	Maximum length of a CDF file name (excluding the NUL[5] 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 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).

[5] The ASCII null character, 0x0.

2.18 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. A new Perl script, **CDFsetFileBackward**, can be called to control the backward compatibility from an application before a CDF file is created (i.e. CDFcreate). This function takes an argument to control the backward file compatibility. Passing a flag value of **BACKWARDFILEon**, also defined in the Perl-CDF package, to the function will cause new files 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 or V2.7 library. If this option is specified, the maximum file is limited to 2G bytes. Passing a flag value of **BACKWARDFILEoff** will use the default file creation mode and new files created 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 create two CDF files: "MY_TEST1.cdf" is a V3.* file while "MY_TEST2.cdf" a V2.7 file.

```
.
.
my   $id1, $id2;                 #  CDF  identifier.
my   $status;                    #  Returned status code.
my   $numDims = 0;               #  Number of dimensions.
my   @dimSizes = ( 0 );          #  Dimension sizes.
.
.
$status =  CDF::CDFlib (CREATE_, CDF_, "MY_TEST1", $numDims, \@dimSizes, \$id1,
                        NULL_);
UserStatusHandler ("1.0", $status) if ($status < CDF_OK) ;
.
.
CDF::CDFsetFileBackward(BACKWARDFILEon);
$status =  CDF::CDFlib (CREATE_, CDF_, "MY_TEST2", $numDims, \@dimSizes, \$id2,
                        NULL_);
UserStatusHandler ("2.0", $status) if ($status < CDF_OK) ;
.
.
```

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 **CDF$FILEBACKWARD** 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** script 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.

```
.
.
my   $status;          #  Returned status code.
my   $flag;            #  File backward flag.
.
$flag = CDF::CDFgetFileBackward();
```

2.19 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: 0 for **NO_CHECKSUM** and 1 for **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 3.13 and 3.18 for the Standards Interface functions and Section 4.6 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 **CDF$CHECKSUM** 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.

```
.
.
.
my   $id1, $id2;              #  CDF  identifier.
my   $status;                 #  Returned status code.
my   $numDims = 0;            #  Number of dimensions.
my   @dimSizes = (0);         #  Dimension sizes.
my   $checksum;               #  Checksum code.
.
.
$status =  CDF::CDFlib (CREATE_, CDF_, "MY_TEST1", $numDims, \@dimSizes, \$id1,
                        NULL_);
UserStatusHandler ("1.0", $status) if ($status < CDF_OK) ;
```

```
.
.
$checksum = 1;
$status =  CDF::CDFlib (SELECT_, CDF_, $id1,
                        PUT_, CDF_CHECKSUM_, $checksum,
                        NULL_);
UserStatusHandler ("2.0", $status) if ($status < CDF_OK) ;
.
$status =  CDF::CDFlib (OPEN_, CDF_, “MY_TEST2”, \$id2,
                        NULL_);
UserStatusHandler ("3.0", $status) if ($status < CDF_OK) ;
.
.
$checksum = 0;
$status =  CDF::CDFlib (SELECT_, CDF_, $id2,
                        PUT_, CDF_CHECKSUM_, $checksum,
                        NULL_);
UserStatusHandler ("4.0", $status) if ($status < CDF_OK) ;
.
.
```

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.

```
.
.
.
my   $id;                    #  CDF  identifier.
my   $status;                #  Returned status code.
my   $checksum;              #  Checksum code.
.
.
$status =  CDF::CDFlib (OPEN_, CDF_, “MY_TEST1”, \$id,
                        NULL_);
UserStatusHandler ("1.0", $status) if ($status < CDF_OK) ;
.
.
$status =  CDF::CDFlib (SELECT_, CDF_, $id,
                        GET_, CDF_CHECKSUM_, \$checksum,
                        NULL_);
UserStatusHandler ("2.0", $status) if ($status < CDF_OK) ;
if ($checksum == MD5_CHECKSUM) {
   …..
}

.
```

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

2.20 Data Validation

To ensure the data integrity from CDF files and secure opetating of CDF-based applications, a data validaion 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 consistnent 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 safe-guard 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. [6]

This validation feature is controlled by the setting /unseting the environment variable **CDF_VALIDATE** on all Unix platforms, Mac OS X and Windows, or **CDF$VALIDATE** 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 perfomed. 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, subroutine **CDFsetValidate**, at application level will overwrite the setup from the command line. The validation is set to be on when value **1** (one) is passed into as the argument. Value **0** (zero) will set off the validation. **CDFgetValidate** will return the validation mode, 1 (one) means data being validated, o (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 off when the CDF file, "TEST", is open.

```
.
.
my   $id;      ;                  #  CDF  identifier.
my   $status;                     #  Returned status code.
.
CDF::CDFsetValidate(0);
$status =  CDF::CDFlib (OPEN_, CDF_, "TEST", \$id,
                        NULL_);
UserStatusHandler ("2.0", $status) if ($status < CDF_OK) ;
.

.
```

2.21 8-Byte Integer

Both data types of CDF_INT8 and CDF_TIME_TT2000 use 8-byes signed integer. Tests show that on the 32-bit Perl environment, large values from these data types, especially common 18-digits values for TT2000 data type, will not be precisely preserved. In oder to preserve the data values, the **Math::BigInt** module is used.for these types. When a data of such types is returned by a CDF module, it is wraped into a BigInt object. Similarly, passing a value of these types, it should also be in BigInt object.

The following example shows the difference between a BigInt object and a regular value from CDF_TIME_TT2000 data type after it is encoded on a 32-bit Perl.

```
use Math::BigInt;

BEGIN { unshift @INC,'/Users/cdf/PerlCDF33_2/blib/arch',
```

[6] The data validation during the open process will not check the variable data. It is still possible that data could be corrupted, especially compression is involved. To fully validate a CDF file, use cdfdump tool with "-detect" switch.

```
                                '/Users/cdf/PerlCDF33_2/blib/lib'; }
use CDF;

my $ttb = Math::BigInt->new('340203790171876765');
my $ttr = 340203790171876765;
my ($tt2000b, $tt2000r);
CDF::encodeTT2000($ttb, $tt2000b);
CDF::encodeTT2000($ttr, $tt2000r);
print $tt2000b,"(bigint) vs ",$tt2000r,"(regular) \n";


2010-10-13T01:02:03.987876765(bigint) vs 2010-10-13T01:02:03.987876736(regular)
```

Chapter 3

3 Standard Interface

The Standard Interface functions described in this chapter represents the Standard Interface functions. They are based on the original Standard Interface developed for the C. This set of interfaces only provides a very limited functionality within the CDF library. For example, it can not handle zVariables and has no access to attribute's entry corresponding to the zVariables (zEntries). If you want to create or access zVariables and zEntries, or operate any single item not accessible from the Standard Interface in a CDF file, you must use the Internal Interface described in Chapter 4.

Standard Interface functions are easier-to-use and require a much shorter learning curve than the Internal Interface, but it's not as efficient as Internal Interface and can only create and maipulate rVariables, not zVariables. If you are not familiar with Internal Interface and need a very simple CDF in a short time, the use of Standard Interface is recommended. However, the Internal Interface (see Chapter 4 for details) is strongly recommended since it's not really hard to learn (see testPerlCDFii.pl included in the Perl-CDF package) and much more flexible and powerful than the Standard Interface.

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 Standard Interface functions callable from Perl applications. Any function in Standard Interface that deals with a variable, its value or attribute, applies only to rVariables. Most functions return a status code (see Chapter 5). The Internal Interface is described in Chapter 4. An application can use either or both interfaces when necessary.

3.1 CDFattrCreate

```
CDF::CDFattrCreate(          #  out --  Completion status code.
my  id,                      #  in --  CDF  identifier.
my  $attrName,               #  in -- Attribute name.
my  $attrScope,              #  in --  Scope of attribute.
my  \$attrNum);              #  out -- Attribute number.
```

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

The arguments to CDFattrCreate are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
attrName	The name of the attribute to create. This may be at most CDF_ATTR_NAME_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 2.12.
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.

3.1.1 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).

```
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $UNITSattrName  =  "Units";           #  Name of "Units" attribute.
my   $UNITSattrNum;                        #  "Units" attribute number.
my   $TITLEattrNum;                        #  "TITLE" attribute number.
my   $TITLEattrScope  =  GLOBAL_SCOPE;     #  "TITLE" attribute scope.
.
.
$status =  CDF::CDFattrCreate  ($id,  "TITLE",  $TITLEattrScope,  \$TITLEattrNum);
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrCreate  ($id,  $UNITSattrName,  VARIABLE_SCOPE,  \$UNITSattrnum);
UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);
.
.
```

3.2 CDFattrEntryInquire

```
CDF::CDFattrEntryInquire(          #  out --  Completion status code.
my $id,                            #  in --  CDF  identifier.
my $attrNum,                       #  in -- Attribute number.
my $entryNum,                      #  in --  Entry number.
my  \$dataType,                    #  out --  Data type.
```

```
my  \$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 CDFopen.
attrNum	The attribute number for which to inquire an entry. This number may be determined with a call to CDFattrNum (see Section 3.5).
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 2.5.
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.

3.2.1 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.

```
.
.
.
my   $id;                              #   CDF  identifier.
my   $status;                          #   Returned status code.
my   $attrN;                           #  Attribute number.
my   $entryN;                          #   Entry number.
my   $attrName;                        #  Attribute name.
my   $attrScope;                       #  Attribute scope.
my   $maxEntry;                        #   Maximum entry number used.
my   $dataType;                        #   Data type.
my   $numElems;                        #   Number of elements  (of the data type).
.
.
$attrN  =  CDF::CDFgetAttrNum  ($id,  "TMP");
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrInquire  ($id, $attrN, \$attrName,  \$attrScope,  \$maxEntry);
UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);

for ($entryN  =  0; $entryN  <=  $maxEntry; $entryN++)  {
     $status =  CDF::CDFattrEntryInquire  ($id, $attrN, $entryN,  \$dataType,  \$numElems);
```

```
    if ($status < CDF_OK) {
        if ($status != NO_SUCH_ENTRY) UserStatusHandler ("3.0". $status);
    }
    else {
        process entries
        .
        .
    }
}
```

3.3 CDFattrGet

```
CDF::CDFattrGet(          # out -- Completion status code.
my $id,                   # in -- CDF identifier.
my $attrNum,              # in -- Attribute number.
my $entryNum,             # in -- Entry number.
my \$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 CDFopen.
attrNum	The attribute number. This number may be determined with a call to CDFattrNum (Section 3.5).
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 in the variable value.

3.3.1 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).

```
.
.
.
my   $id;                 # CDF identifier.
my   $status;             # Returned status code.
my   $attrN;              # Attribute number.
```

```
my   $entryN;                    #  Entry number.
my   $dataType;                  #  Data type.
my   $numElems;                  #  Number of elements  (of data type).
my   $buffer;                    #  Buffer to receive  value.
.
.
$attrN  =  CDF::CDFattrNum  (id,  "UNITS");
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$entryN  =  CDF::CDFvarNum  (id,  "PRES_LVL");        #  The rEntry number is the rVariable number.

UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrEntryInquire  ($id, $attrN, $entryN,  \$dataType,  \$numElems);

UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
if ($dataType  ==  CDF_CHAR)  {
  $status =  CDF::CDFattrGet  ($id, $attrN, $entryN,  \$buffer);
  UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);

  print  "Units of PRES_LVL  variable:  $buffer \n";
}
.
.
```

3.4 CDFattrInquire

```
CDF::CDFattrInquire(          #  out --  Completion status code.
my $id,                       #  in --  CDF  identifier.
my $attrNum,                  #  in -- Attribute number.
my \$attrName,                #  out -- Attribute name.
my \$attrScope,               #  out -- Attribute scope.
my \$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 CDFopen.
attrNum	The number of the attribute to inquire. This number may be determined with a call to CDFattrNum (see Section 3.5).
attrName	The attribute's name.
attrScope	The scope of the attribute. Attribute scopes are defined in Section 2.12.
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 4). If no entries exist for the attribute, then a value of -1 will be passed back.

3.4.1 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.

```
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $numDims;                             #  Number of dimensions.
my   @dimSizes = (CDF_MAX_DIMS);           #  Dimension sizes (allocate to allow the maximum
                                                  number of dimensions).
my   $encoding;                            #  Data encoding.
my   $majority;                            #  Variable majority.
my   $maxRec;                              #  Maximum record number in CDF.
my   $numVars;                             #  Number of variables in CDF.
my   $numAttrs;                            #  Number of attributes in CDF.
my   $attrN;                               #  attribute number.
my   $attrName;                            #  attribute name --  +1 for NUL terminator.
my   $attrScope;                           #  attribute scope.
my   $maxEntry;                            #  Maximum entry number.
.
.
$status =  CDF::CDFinquire  ($id,  \$numDims,  \@dimSizes,  \$encoding,  \$majority,
                                  \$maxRec,  \$numVars,  \$numAttrs);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
for ($attrN  =  0; $attrN  <  $numAttrs; $attrN++)  {
     $status =  CDFattrInquire  ($id, $attrN, \$attrName,  \$attrScope,  \$maxEntry);
      if ($status <  CDF_OK)                                  #  INFO status codes  ignored.
          UserStatusHandler  ("2.0", $status);
     else
          print  ("$attrName \n");
}
.
.
```

3.5 CDFattrNum

```
CDF:: CDFattrNum(          #   out -- Attribute number.
my  $id,                   #   in --  CDF  id
my  $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 is returned. Error codes are less than zero (0).

The arguments to CDFattrNum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
attrName	The name of the attribute for which to search. Attribute names are case-sensitive.

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

3.5.1 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.

```
.
.
.
my   $id;              #  CDF  identifier.
my   $status;          #  Returned status code.
my   $attrNum;         #  Attribute number.
.
.
$attrNum = CDF::CDFattrNum($id,"pressure");
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrRename ($id,  $attrNum,  "PRESSURE");
UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);
```

3.6 CDFattrPut

```
CDF::CDFattrPut(          #  out --  Completion status code.
my  $id,                  #  in --  CDF  identifier.
my  $attrNum,             #  in -- Attribute number.
my  $entryNum,            #  in --  Entry number.
my  $dataType,            #  in --  Data type of this entry.
my  $numElements,         #  in --  Number of elements  (of the data type).
my  $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 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 2.5.
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.

3.6.1 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.

```
.
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $entryNum;                            #  Entry number.
my   $numElements;                         #  Number of elements  (of data type).
my   $title = "CDF  title.";               #  Value of TITLE attribute, entry number 0.
my   @TMPvalids  =  (15,30);               #  Value(s) of VALIDs attribute, rEntry for rVariable TMP.
my   $TITLE_LEN = 10;                      #  Length of CDF  title.
my   $attrNum;                             #  Attribute number.
my   $varNum;                              #  rVariable number.    .
.
.
$entryNum  =  0;
$attrNum = CDF:: CDFgetAttrNum(id,"TITLE");
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrPut  ($id, $attrNum, $entryNum,  CDF_CHAR, 10,  $title);
UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);
.
.
$numElements  =  2;
$attrNum = CDF:: CDFgetAttrNum(id,"VALIDs");
UserStatusHandler (“3.0”. $status) if ($status <  CDF_OK);
$varNum = CDF:: CDFgetVarNum(id,"TMP");
UserStatusHandler (“4.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrPut  ($id,  $attrNum,  $varNum, CDF_INT2,  $numElements, \@TMPvalids);
UserStatusHandler  (“5.0”. $status) if ($status <  CDF_OK);
.
.
```

3.7 CDFattrRename

```
CDF::CDFattrRename(          #  out --  Completion status code.
my  $id,                     #  in --  CDF  identifier.
my  $attrNum,                #  in -- Attribute number.
my  $attrName);              #  in --  New attribute name.
```

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

The arguments to CDFattrRename are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
attrNum	The number of the attribute to rename. This number may be determined with a call to CDFattrNum (see Section 3.5).
attrName	The new attribute name. Attribute names are case-sensitive.

3.7.1 Example(s)

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

```
.
.
.
my   $id;                    #  CDF  identifier.
my   $status;                #  Returned status code.
my   $attrNum;               #  Attribute number.
.
.
$attrNum = CDF:: CDFgetAttrNum(id,"LAT");
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
$status =  CDF::CDFattrRename ($id,  $attrNum,  "LATITUDE");
UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);
.
.
```

3.8 CDFclose

```
CDF::CDFclose(               #  out --  Completion status code.
my  $id);                    #  in --  CDF  identifier.
```

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

NOTE: You must close a CDF with CDFclose to guarantee that all modifications you have made will actually be written to the CDF's file(s). If your program exits, normally or otherwise, without a successful call to CDFclose, the CDF's cache buffers are left unflushed.

The arguments to CDFclose are defined as follows:

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

3.8.1 Example(s)

The following example will close an open CDF.

```
.
.
.
my   $id;                    #  CDF  identifier.
my   $status;                #  Returned status code.
.
.
$status =  CDF::CDFclose  ($id);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
.
.
```

3.9 CDFcreate

```
CDF::CDFcreate(              #  out --  Completion status code.
my  $CDFname,                #  in --  CDF file name.
my  $numDims,                #  in --  Number of dimensions,  rVariables.
my  \@dimSizes,              #  in --  Dimension sizes,  rVariables.
my  $encoding,               #  in --  Data encoding.
my  $majority,               #  in --  Variable majority.
my  \$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 2.6.
majority	The majority for variable data. Specify one of the majorities described in Section 2.8.
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 3.8).

3.9.1 Example(s)

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

```
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $numDims  =  3;                       #  Number of dimensions, rVariables.
my   @dimSizes  =  (180,360,10);           #  Dimension sizes, rVariables.
my   $majority =  ROW_MAJOR;               #  Variable majority.
.
.
$status =  CDF::CDFcreate  ("test1",  $numDims,  \@dimSizes,  NETWORK_ENCODING, $majority,  &id);
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
.
.
```

ROW_MAJOR and NETWORK_ENCODING are defined in the Perl-CDF package.

3.10 CDFdelete

```
CDF::CDFdelete(                    #   out --  Completion status code.
```

```
my  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 CDFopen.

3.10.1 Example(s)

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

```
.
.
.
my   $id;                       #   CDF  identifier.
my   $status;                   #   Returned status code.
.
.
$status =  CDF::CDFopen  ("test2",  \$id);
if ($status <  CDF_OK)                          #  INFO status codes  ignored.
   UserStatusHandler  (“1.0”, $status);
else  {
  $status =  CDF::CDFdelete  ($id);
  UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);
}
.
.
```

3.11 CDFdoc

```
CDF::CDFdoc(                    #   out --  Completion status code.
my  $id,                        #   in --  CDF  identifier.
my  \$version,                  #   out --  Version number.
my  \$release,                  #   out --  Release number.
my  \$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 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 string will contain a newline character after each line of the Copyright notice.

3.11.1 Example(s)

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

```
.
.
.
my   $id;                    #  CDF  identifier.
my   $status;                #  Returned status code.
my   $version;               #  CDF  version number.
my   $release;               #  CDF  release number.
my   $Copyright;             #  Copyright  notice.
.
.
$status =  CDF::CDFdoc  ($id,  \$version,  \$release,  \$Copyright);
if ($status <  CDF_OK)                                          #  INFO status codes  ignored
   UserStatusHandler  ("1.0", status);
else  {
   print  ("CDF  V$version.$release\n");
   print ("$Copyright");
}
.
.
```

3.12 CDFerror

```
CDF::CDFerror(               #  out --  Completion status code.
my $status,                  #  in --  Status  code.
my $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 5 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.

3.12.1 Example(s)

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

```
.
.
.
my   $id;                    #  CDF  identifier.
my   $status;                #  Returned status code.
my   $text;                  #  Explanation text.
.
.
$status =  CDF::CDFopen  ("giss_wetl",  \$id);
if ($status <  CDF_WARN)  {          #  INFO and WARNING  codes  ignored.
    CDF::CDFerror ($status,  \$text);
    print  ("ERROR>  $text\n");
}
.
.
```

3.13 CDFgetChecksum

```
CDF::CDFgetChecksum (        #  out --  Completion status code.
my  $id,                     #  in --  CDF  identifier.
my \$checksum);              #  out --  CDF's checksum mode.
```

CDFgetChecksum returns the checksum mode of a CDF. The CDF checksum mode is described in Section 2.19.

The arguments to CDFgetChecksum are defined as follows:

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

3.13.1 Example(s)

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

```
.
.
.
my   $id;                          #  CDF  identifier.
my   $status;                      #  Returned status code.
my   $checksum;                    #  CDF's checksum.
.
.
$status = CDF::CDFgetChecksum ($id, \$checksum);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);
```

.
.

3.14 CDFgetFileBackward

CDF::CDFgetFileBackward() # out -- Backward file indicator.

CDFgetFileBackward is used to get the backward file indicator. When the indicator is 1 (true), all newly created files are of Version 2.7, backward compatible files, not V3.*.

The arguments to CDFgetFileBackward defined as follows:

N/A

3.14.1 Example(s)

In the following example, the backward file indicator is retrieved.

```
.
.
.
my    $backwardFlag;                    #   File backward flag.
.
.
$backwardFlag  =  CDF::CDFgetFileBackward();
.
.
```

3.15 CDFgetValidate

CDF::CDFgetValidate () # out -- Validation mode.

CDFgetValidate returns the validation mode when opening CDF files. The CDF validation mode is described in Section 2.20.

The arguments to CDFgetValidate are defined as follows:

N/A

3.15.1 Example(s)

The following example returns the data validation mode when opening the CDF files.

```
.
```

```
.
.
my    $validate;                        #   CDF's validation mode.
.
.
$validate = CDF::CDFgetValidate ();
.
.
```

3.16 CDFinquire

```
CDF::CDFinquire(                        #   out --  Completion status code.
my  $id,                                #    in --  CDF  identifier
my  \$numDims,                          #   out --  Number of dimensions,  rVariables.
my  \@dimSizes,                         #   out --  Dimension sizes,  rVariables.
my  \$encoding,                         #   out --  Data encoding.
my  \$majority,                         #   out --  Variable majority.
my  \$maxRec,                           #   out --  Maximum record number in the CDF,  rVariables.
my  \$numVars,                          #   out --  Number of rVariables in the CDF.
my  \$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 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 2.6.
majority	The majority of the variable data. The majorities are defined in Section 2.8.
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.

3.16.1 Example(s)

The following example returns the basic information about a CDF.

```
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $numDims;                             #  Number of dimensions,  rVariables.
my   @dimSizes                ;            #  Dimension sizes,  rVariables (allocate to allow the
                                           #  maximum number of dimensions).
my   $encoding;                            #  Data encoding.
my   $majority;                            #  Variable majority.
my   $maxRec;                              #  Maximum record number,  rVariables.
my   $numVars;                             #  Number of rVariables in CDF.
my   $numAttrs;                            #  Number of attributes in CDF.
.
.
$status =  CDF::CDFinquire  ($id,  \$numDims,  \@dimSizes,  \$encoding,  \$majority,
                                \$maxRec,  \$numVars,  \$numAttrs);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
.
.
```

3.17 CDFopen

```
CDF::CDFopen(                  #  out --  Completion status code.
my  $CDFname,                  #  in --  CDF file name.
my  \$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.

3.17.1 Example(s)

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

```
.
.
.
my   $id;                                  #   CDF  identifier.
my   $status;                              #   Returned status code.
my   $CDFname  = "NOAA1";                  #  File name of CDF.
.
.
$status =  CDF::CDFopen  ($CDFname,  \$id);
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
.
.
```

3.18 CDFsetChecksum

```
CDF::CDFsetChecksum (          #   out --  Completion status code.
my  $id,                       #   in --  CDF  identifier.
my  $checksum);                #   in --  CDF’s checksum mode.
```

CDFsetChecksum specifies the checksum mode for the CDF. The CDF checksum mode is described in Section 2.19.

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 CDFopen.
checksum	The checksum mode (NO_CHECKSUM or MD5_CHECKSUM).

3.18.1 Example(s)

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

```
.
.
.
my   $id;                          #   CDF  identifier.
my   $status;                      #   Returned status code.
my   $checksum;                    #   CDF’s checksum.
.
```

```
.
$checksum= 0;
$status = CDF::CDFsetChecksum ($id, $checksum);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);

.
.
```

3.19 CDFsetFileBackward

```
CDF::CDFsetFileBackward(                    #
my  $flag)                                  #   in --  Backward file flag
```

CDFsetFileBackward is used to set the backward file indicator. When the indicator is 1 (true), all newly created files are of Version 2.7, backward compatible files, not V3.*.

The arguments to CDFsetFileBackward defined as follows:

flag	The backward file flag

3.19.1 Example(s)

In the following example, the backward file indicator is set to true so a new CDF file(s) of V2.7, instead of V3.*, will be created.

```
.
.
.
my    $backwardFlag;                              #   Backward file flag.
.
.
 $backwadFlag = 1;
 CDF::CDFsetFileBackward($backwardFlag);
.
.
```

3.20 CDFsetValidate

```
CDF::CDFsetValidate (
my  $validate);                     #   in --  CDF's validation mode.
```

CDFsetValidate specifies the validation mode when opening a CDF file. The CDF validation mode is described in Section 2.20.

The arguments to CDFsetValidate are defined as follows:

validate The validation mode.

3.20.1 Example(s)

The following example turns on the data validation when opening the CDF file, “TEST”..

```
.
.
.
my   $id;                              #  CDF  identifier.
my   $status;                          #  Returned status code.
.
.
CDF::CDFsetValidate (1);
$status = CDF::CDFlib(OPEN_, CDF_, “TEST”, \$id,
                      NULL_);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);

.
.
```

3.21 CDFvarClose

```
CDF::CDFvarClose(              #  out --  Completion status code.
my  $id,                       #  in --  CDF  identifier.
my  $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.

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:

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

varNum The variable number for the open rVariable’s file. This identifier must have been initialized by a call to CDFgetVarNum.

3.21.1 Example(s)

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

```
.
.
.
my   $id;                    #  CDF  identifier.
my   $status;                #  Returned status code.
my   $varNum;                #  rVariable number.
.
.
$varNum = CDF::CDFvarNum (id, "Flux");
UserStatusHandler  ("1.0". $varNum) if ($varNum <  CDF_OK);
$status =  CDF::CDFvarClose  (id, $varNum);
UserStatusHandler  ("2.0". $status) if ($status <  CDF_OK);
.
.
```

3.22 CDFvarCreate

```
CDF::CDFvarCreate(                #  out --  Completion status code.
my  $id,                          #  in --  CDF  identifier.
my  $varName,                     #  in --  rVariable name.
my  $dataType,                    #  in --  Data type.
my  $numElements,                 #  in --  Number of elements  (of the data type).
my  $recVariance,                 #  in --  Record  variance.
my  \@dimVariances,               #  in --  Dimension variances.
my  \$varNum);                    #  out --  rVariable number.
```

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

The arguments to CDFvarCreate are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
varName	The name of the rVariable to create. This may be at most CDF_VAR_NAME_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 2.5.
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 2.9.
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 2.9. 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.

3.22.1 Example(s)

The following example will create several rVariables in a CDF. In this case EPOCH is a 0-dimensional, LATITUDE and LONGITUDE are 2-diemnational, and TEMPERATURE is a 1-dimensional.

```
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $EPOCHrecVary  =  VARY;               #  EPOCH record variance.
my   $LATrecVary  =  NOVARY;               #  LAT record variance.
my   $LONrecVary  =  NOVARY;               #  LON record variance.
my   $TMPrecVary  =  VARY;                 #  TMP record variance.
my   $EPOCHdimVarys  =  NOVARY;            #  EPOCH dimension variances.
my   @LATdimVarys  =  (VARY,VARY);         #  LAT dimension variances.
my   @LONdimVarys  =  (VARY,VARY);         #  LON dimension variances.
my   @TMPdimVarys  =  (VARY,VARY);         #  TMP dimension variances.
my   $EPOCHvarNum;                         #  EPOCH  zVariable number.
my   $LATvarNum;                           #  LAT zVariable number.
my   $LONvarNum;                           #  LON zVariable number.
my   TMPvarNum;                            #  TMP  zVariable number.
my   @EPOCHdimSizes = (3);                 #  EPOCH  dimension sizes.
my   @LATLONdimSizes = (2,3);              #  LAT/LON dimension sizes.
my   @TMPdimSizes = (3);                   #  TMP  dimension sizes.
.
.
$status =  CDF::CDFvarCreate  ($id,  "EPOCH",  CDF_EPOCH,  1,
                               $EPOCHrecVary,  \@EPOCHdimVarys,  \$EPOCH varNum);
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);

$status =  CDF::CDFvarCreate  ($id,  "LATITUDE",  CDF_INT2,  1,
                               $LATrecVary,  \@LATdimVarys,  \$LATvarNum);
UserStatusHandler  (“2.0”. $status) if ($status <  CDF_OK);

$status =  CDF::CDFvarCreate  ($id,  "LONGITUDE",  CDF_INT2,  1,
                               $LONrecVary,  \@LONdimVarys,  \$LONvarNum);
UserStatusHandler  (“3.0”. $status) if ($status <  CDF_OK);

$status =  CDF::CDFvarCreate  ($id, "TEMPERATURE", CDF_REAL4, 1,
                               $TMPrecVary, \@TMPdimVarys, \$TMPvarNum);
UserStatusHandler  (“4.0”. $status) if ($status <  CDF_OK);
.
.
```

3.23 CDFvarGet

```
CDF::CDFvarGet(          #  out --  Completion status code.
my  $id,                 #  in --  CDF  identifier.
my  $varNum,             #  in --  rVariable number.
my  $recNum,             #  in --  Record number.
my  \@indices,           #  in --  Dimension indices.
my  \$value);            #  out --  Value.
```

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

The arguments to CDFvarGet are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
varNum	The rVariable number from which to read data.
recNum	The record number at which to read.
indices	The dimension indices within the record.
value	The data value read. This buffer must be large enough to hold the value.

3.23.1 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.

```
.
.
.
my    $id;                       #  CDF identifier.
my    $varNum;                   #  rVariable number.
my    $recNum;                   #  The record number.
my    @indices;                  #  The dimension indices.
my    $value1, $value2;          #  The data values.
.
.
$varNum = CDF::CDFvarNum ($id, "MY_VAR");
if ($varNum < CDF_OK)  Quit  ("….");
$recNum = 0;
$indices[0] = 0;
$indices[1] = 0;
$status = CDF::CDFvarGet ($id, $varNum, $recNum, \@indices, \$value1);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
$indices[0] = 1L;
$indices[1] = 1L;
$status = CDF::CDFvarGet ($id, $varNum, $recNum, \@indices, \$value2);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
.
.
```

3.24 CDFvHpGet

```
CDF::CDFvHpGet(          #  out --  Completion status code.
my  $id,                 #  in --  CDF  identifier.
my  $varNum,             #  in --  rVariable number.
my  $recStart,           #  in --  Starting record number.
my  $recCount,           #  in --  Number of records.
my  $recInterval,        #  in --  Subsampling interval between records.
my  \@indices,           #  in --  Dimension indices of starting  value.
my  \@counts,            #  in --  Number of values along  each dimension.
my  \@intervals,         #  in --  Subsampling intervals along  each dimension.
my  \@buffer);           #  out --  Buffer of values.
```

CDFvHpGet 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 CDFvHpGet 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.

3.24.1 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 CDFvHpGet rather than numerous calls to CDFvarGet.

```
.
.
.
my   $id;                          #  CDF  identifier.
my   $status;                      #  Returned status code.
my   @tmp;                         #  Temperature  values.
my   $varN;                        #  rVariable number.
my   $recStart  =  13;             #  Record number.
my   $recCount  =  1;              #  Record  counts.
my   $recInterval  =  1;           #  Record interval.
my   @indices  =  (0,0,0);         #  Dimension indices.
my   @counts  =  (180,91,10);      #  Dimension counts.
my   @intervals  =  (1,1,1);       #  Dimension intervals.
.
.
$varN  =  CDF::CDFgetVarNum  ($id,  "Temperature");
if ($varN  <  CDF_OK)  UserStatusHandler  ($varN);
status = CDF::CDFgetHyperGet ($id,  $varN,  $recStart,  $recCount,  $recInterval,
                              \@indices,  \@counts, \@intervals,  \@tmp);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
.
.
```

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.

3.25 CDFvHpPut

```
CDF::CDFvHpPut(              # out -- Completion status code.
my  $id,                     # in -- CDF  identifier.
my  $varNum,                 # in -- rVariable number.
my  $recStart,               # in -- Starting record number.
my  $recCount,               # in -- Number of records.
my  $recInterval,            # in -- Interval between records.
my  \@indices,               # in -- Dimension indices of starting  value.
my  \@counts,                # in -- Number of values along  each dimension.
my  \@intervals,             # in -- Interval between values along  each dimension.
my  \@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.

3.25.1 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 CDvHpPut rather than numerous calls to CDFvarPut.

```
.
.
.
my   $id;                        #  CDF  identifier.
my   $status;                    #  Returned status code.
my   $lat;                       #  Latitude  value.
my   @lats;                      #  Buffer of latitude  values.
my   $varN;                      #  rVariable number.
my   $recStart  =  0;            #  Record number.
my   $recCount  =  1;            #  Record  counts.
my   $recInterval  =  1;         #  Record interval.
my   @indices  =  (0,0);         #  Dimension indices.
my   @counts  =  (1,181);        #  Dimension counts.
my   @intervals  =  (1,1);       #  Dimension intervals.

.
.
$varN  =  CDF::CDFvarNum  ($id,  "LATITUDE");
if ($varN  < CDF_OK)  UserStatusHandler  ($varN);
for ($lat =  -90;  $lat <=  90;  $lat ++)
      $lats[90+lat]  =  $lat;
```

```
$status = CDF::CDFvHpPut ($id, $varN, $recStart, $recCount, $recInterval,
                          \@indices, \@counts, \@intervals, \@lats);
UserStatusHandler ("1.0". $status) if ($status < CDF_OK);
.
.
```

3.26 CDFvarInquire

```
CDF::CDFvarInquire(          # out -- Completion status code.
my $id,                      # in -- CDF identifier.
my $varNum,                  # in -- rVariable number.
my $varName,                 # out -- rVariable name.
my \$dataType,               # out -- Data type.
my \$numElements,            # out -- Number of elements (of the data type).
my \$recVariance,            # out -- Record variance.
my \@dimVariances);          # 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 CDFvHpGet) to determine the data type and number of elements (of that data type).

The arguments to CDFvarInquire are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
varNum	The number of the rVariable to inquire. This number may be determined with a call to CDFvarNum (see Section 3.27).
varName	The rVariable's name. This character string must not be greater than CDF_VAR_NAME_LEN256 characters.
dataType	The data type of the rVariable. The data types are defined in Section 2.5.
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 2.9.
dimVariances	The dimension variances. Each element of dimVariances receives the corresponding dimension variance. The dimension variances are defined in Section 2.9. For 0-dimensional rVariables this argument is ignored (but a placeholder is necessary).

3.26.1 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.

```
.
.
.
my   $id;                                  #  CDF  identifier.
my   $status;                              #  Returned status code.
my   $varNum;                              #  rVariable number.
my   $varName;                             #  rVariable name.
my   $dataType;                            #  Data type of the rVariable.
my   $numElems;                            #  Number of elements  (of data type).
my   $recVary;                             #  Record  variance.
my   @dimVarys;                            #  Dimension variances  (allocate to allow the
                                           #  maximum number of dimensions).
.
.
$varNum = CDF:: CDFgetVarNum(id,"HEAT_FLUX");
UserStatusHandler  (“1.0”. $varNum) if ($varNum <  CDF_OK);
$status =  CDF::CDFvarInquire  ($id,  $varNum,  \$varName,  \$dataType,
                                \$numElems,  \$recVary,  \@dimVarys);
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
.
.
```

3.27 CDFvarNum

```
CDF::CDFvarNum(                    #  out --  Variable number.
my  $id,                           #  in --  CDF  identifier.
my  $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 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 CDFopen.
varName	The name of the variable to search. Variable names are case-sensitive.

3.27.1 Example(s)

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

```
.
.
.
```

```
my   $id;                                   #  CDF  identifier.
my   $status;                               #  Returned status code.
my   $varNum;                               #  rVariable number.
my   $varName;                              #  Variable name.
my   $dataType;                             #  Data type of the rVariable.
my   $numElements;                          #  Number of elements  (of the data type).
my   $recVariance;                          #  Record  variance.
my   @dimVariances;                         #  Dimension variances.
.
.
$varNum = CDF:: CDFvarNum(id,"LATITUDE");
UserStatusHandler  ("1.0". $varNum) if ($varNum <  CDF_OK);
$status =  CDF::CDFvarInquire  ($id,  $varNum,  \$varName,  \$dataType,
                                \$numElements,  \$recVariance,  \@dimVariances);
UserStatusHandler  ("2.0". $status) if ($status <  CDF_OK);
.
.
```

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 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 3.26.

3.28 CDFvarPut

```
CDF::CDFvarPut(              #  out --  Completion status code.
my  $id,                     #  in --  CDF  identifier.
my  $varNum,                 #  in --  rVariable number.
my  $recNum,                 #  in --  Record number.
my  \@indices,               #  in --  Dimension indices.
my  \$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 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.

3.28.1 Example(s)

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

```
.
.
.
my   $id;                          #  CDF identifier.
my   $varNum;                      #  rVariable number.
my   $recNum;                      #  The record number.
my   @indices;                     #  The dimension indices.
my   $value1, $value2;             #  The data values.
.
.
$varNum = CDF::CDFgetVarNum ($id, "MY_VAR");
if ($varNum < CDF_OK)  Quit  ("….");
$recNum = 0;
$indices[0] = 0;
$indices[1] = 0;
$value1 = 10.1;
$status = CDF::CDFvarPut ($id, $varNum, $recNum, \@indices, \$value1);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
$indices[0] = 1;
$indices[1] = 1;
$value2 = 20.2;
$status = CDF::CDFvarPut ($id, $varNum, $recNum, \@indices, \$value2);
UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
.
.
```

3.29 CDFvarRename

```
CDF::CDFvarRename(          #  out --  Completion status code.
my  $id,                    #  in --  CDF  identifier.
my  $varNum,                #  in --  rVariable number.
my  $varName);              #  in --  New  name.
```

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 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.

3.29.1 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.

```
.
.
.
my   $id;                          #  CDF  identifier.
my   $status;                      #  Returned status code.
my   $varNum;                      #  rVariable number.
.
.
$varNum  =  CDF::CDFvarNum  ($id,  "TEMPERATURE");
if ($varNum  < CDF_OK)  {
    if ($varNum  !=  NO_SUCH_VAR)  UserStatusHandler  (varNum);
}
else  {
       $status = CDF::CDFvarRename ($id,  $varNum,  "TMP");
        if ($status !=  CDF_OK)  UserStatusHandler  (status);
}
.
.
```

Chapter 4

4 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 4.6. The function prototype for CDFlib is as follows:

```
status = CDF::CDFlib  (function,  ...);
```

4.1 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).

```
.
.
.
my          $id;                                # CDF identifier (handle).
my          $status;                            # Status returned from CDF library.
my          $CDFname = "test1";                 # File name of the CDF.
my          $numDims = 2;                       # Number of dimensions.
my          @dimSizes = {100,200};              # Dimension sizes.
my          $encoding = HOST_ENCODING;          # Data encoding.
my          $majority = ROW_MAJOR;              # Variable data majority.
my          $format = SINGLE_FILE;              # Format of CDF.
.
.
$status = CDFcreate ($CDFname, $numDims, \@dimSizes, $encoding, $majority, \$id);
if ($status != CDF_OK) UserStatusHandler ($status);
```

```
$status = CDF::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.[7] 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 = CDF::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.
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.

[7] 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.

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).

4.2 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_>[8] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.[9]

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_>

[8] This notation is used to specify a function to be performed on an item. The syntax is <function_,item_>.

[9] 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.

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.

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

record interval, rVariables (state)
: An rVariable hyper read or write operation is always performed using the current record interval for the rVariables in the current CDF. When a CDF is opened or created, the current record interval for its rVariables is initialized to one (1). It may then be explicitly selected using the <SELECT_,rVARs_RECINTERVAL_> operation. Note that

the current record interval for rVariables is maintained for a CDF (not each rVariable) - it applies to all of the rVariables in that CDF.

dimension indices, rVariables (state)

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

dimension counts, rVariables (state)

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

dimension intervals, rVariables (state)

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

sequential value, rVariable (state)

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

record number, zVariable (state)

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

record count, zVariable (state)

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

record interval, zVariable (state)

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

dimension indices, zVariable (state)

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

dimension counts, zVariable (state)

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

dimension intervals, zVariable (state)

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

sequential value, zVariable (state)

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

status code (state)

When inquiring the explanation of a CDF status code, the text returned is always for the current status code. One current status code is maintained for the entire CDF library (regardless of the number of open CDFs). The current status code may be selected using the <SELECT_,CDF_STATUS_> operation. There is no default current status code. Note that the current status code is NOT the status code from the last operation performed.[10]

4.3 Returned Status

CDFlib returns a status code. 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.

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

4. In the absence of any errors, warnings, or informational conditions, CDF_OK is returned.

Chapter 5 explains how to interpret status codes. Appendix A lists the possible status codes and the type of each: error, warning, or informational.

4.4 Indentation/Style

Indentation should be used to make calls to CDFlib readable. The following example shows a call to CDFlib using proper indentation.

```
$status = CDF::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 = CDF::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.

4.5 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 by the C compiler and operating system being used. Under normal circumstances that limit would never be reached (or even approached). Note also that a call to CDFlib with a large number of arguments can always be broken up into two or more calls to CDFlib with fewer arguments.

The syntax for CDFlib is as follows:

```
$status = CDF::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.

4.6 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 4.2.

<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: \$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: \$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: \$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: \$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: \$decoding

The decoding. The decodings are described in Section 2.7.

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: \$CDFname

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: \$mode

The -0.0 to 0.0 mode. The -0.0 to 0.0 modes are described in Section 2.15.

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: \$mode

The read-only mode. The read-only modes are described in Section 2.13.

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: \$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: \$mode

The zMode. The zModes are described in Section 2.14.

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: \$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: \$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: \$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: \$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: $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: \$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: \$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: $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: \$percent

The reserve percentage.

The required preselected objects/states are the current CDF and its current rVariable.

<CONFIRM_,rVAR_SEQPOS_>

Confirms the current sequential value for sequential access for the current rVariable (in the current CDF). Note that a current sequential value is maintained for each rVariable individually. Required arguments are as follows:

out: \$recNum

Record number.

out: \@indices

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: \@counts

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: \@indices

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: \@intervals

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: \$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: \$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: \$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: \$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: \$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: $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: \$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: \$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: \@counts

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: \@indices

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: \@intervals

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: $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: \$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: \$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: \$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: \$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: \$recNum

Record number.

out: \@indices

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: \$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: \$scope

Scope of the new attribute. Specify one of the scopes described in Section 2.12.

out: \$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: \$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: $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: \@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: \$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_MAJORITY_> 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: $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: $dataType

Data type of the new rVariable. Specify one of the data types described in Section 2.5.

in: $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: $recVary

Record variance. Specify one of the variances described in Section 2.9.

in: \@dimVarys

Dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 2.9. For 0-dimensional rVariables this argument is ignored (but must be present).

out: \$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: $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: $dataType

Data type of the new zVariable. Specify one of the data types described in Section 2.5.

in: $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: $numDims

Number of dimensions for the zVariable. This may be as few as zero and at most CDF_MAX_DIMS.

in: \@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: $recVary

Record variance. Specify one of the variances described in Section 2.9.

in: \@dimVarys

Dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 2.9. For a 0-dimensional zVariable this argument is ignored (but must be present).

out: \$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:

in: $firstRecord

The record number of the first record to be deleted.

in: $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: $firstRecord

The record number of the first record to be deleted.

in: $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: \$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: \$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: \$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: \$attrName

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: \$attrName

Attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters (excluding the NUL terminator).

out: \$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: \$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: \$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: \$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:

out: \$scope

Attribute scope. The scopes are described in Section 2.12.

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:

out: \$checksum

The checksum mode of the current CDF (NO_CHECKSUM or MD5_CHECKSUM). The checksum mode is described in Section 2.19.

The required preselected objects/states is the current CDF.

<GET_,CDF_COMPRESSION_>
Inquires the compression type/parameters and compression percentage of the current CDF. This refers to the compression of the CDF - not of any compressed variables. The compression percentage is the result of the compressed file size divided by its original, uncompressed file size.[11] Required arguments are as follows:

out: \$cType

The compression type. The types of compressions are described in Section 2.10.

out: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

out: \$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:

out: \$Copyright

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:

out: \$encoding

Data encoding. The encodings are described in Section 2.6.

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: \$format

CDF format. The formats are described in Section 2.4.

The only required preselected object/state is the current CDF.

<GET_,CDF_INCREMENT_>

[11] The compression ratio is (100 – compression percentage): the lower the compression percentage, the better the compression ratio.

Inquires the incremental number of the CDF library that created the current CDF. Required arguments are as follows:

out: \$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: $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: \$cType

The CDF compression type. The types of compressions are described in Section 2.10.

out: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

out: \$cSize

If compressed, size in bytes of the dotCDF file. If not compressed, set to zero (0).

out: \$uSize

If compressed, size in bytes of the dotCDF file when decompressed. If not compressed, size in bytes of the dotCDF file.

There are no required preselected objects/states.

<GET_,CDF_LEAPSECONDLASTUPDATED_>

Inquires the variable lastupdated of the current CDF. Required arguments are as follows:

out: \$lastupdated

The last entry (date) that a new leap second was added to the leap second table on which the CDF is based upon. The value is of YYYYMMDD form. It can also be -1 (from older CDFs) or zero (0) if the table is not used.

The only required preselected object/state is the current CDF.

<GET_,CDF_MAJORITY_>

Inquires the variable majority of the current CDF. Required arguments are as follows:

out: \$majority

Variable majority. The majorities are described in Section 2.8.

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: \$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: \$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: \$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: \$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: \$numVars

Number of zVariables.

The only required preselected object/state is the current CDF.

<GET_,CDF_RELEASE_>
Inquires the release number of the CDF library that created the current CDF. Required arguments are as follows:

out: \$release

Release number.

The only required preselected object/state is the current CDF.

<GET_,CDF_VERSION_>
Inquires the version number of the CDF library that created the current CDF. Required arguments are as follows:

out: \$version

Version number.

The only required preselected object/state is the current CDF.

<GET_,DATATYPE_SIZE_>
Inquires the size (in bytes) of an element of the specified data type. Required arguments are as follows:

in: $dataType

Data type.

out: \$numBytes

Number of bytes per element.

There are no required preselected objects/states.

<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:

out: \$value

Value. This buffer must be large to hold the value. The value is read from the CDF and placed in the variable 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.

<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:

out: \$dataType

Data type. The data types are described in Section 2.5.

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_NUMELEMS_>
Inquires the number of elements (of the data type) of the gEntry at the current gEntry number for the current attribute (in the current CDF). Required arguments are as follows:

out: \$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: \$Copyright

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: \$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: \$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: \$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: \$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: \$value

Value. This buffer must be large to hold the value. The value is read from the CDF and placed in the variable 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_>[12]

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: \$dataType

Data type. The data types are described in Section 2.5.

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: \$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_,rENTRY_NUMSTRINGS_>

Inquires the number of strings (of CDF_CHAR or CDF_UCHAR data type) of the rEntry at the current rEntry number for the current attribute (in the current CDF). Required arguments are as follows:

out: \$numStrings

Number of strings of the character data type. It is only for character data types (CDF_CHAR and CDF_UCHAR). Strings are concatenated and stored in the CDF in a sequence of charaters, with a pre-defined delimiter (**"\N** "), separating the strings. The number of elements for this character data type contains the extra characters used for the delimiter. [13]

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.

---

[12] If the data is a string with with **"\N "** symbol (3 characters), it is an array of strings. Use GET_, rENTRY_STRINGDATA_ method (described in the following statement).

[13] This feature is added in CDF V3.7.0. CDFs of previously versions only allow one single string.

<GET_,rENTRY_STRINGDATA_>

Reads the strings (of CDF_CHAR or CDF_UCHAR data type) of the rEntry at the current rEntry number for the current attribute (in the current CDF). Required arguments are as follows:

out: \@strings

An array of retrieved strings. Spaces for the strings are dynamically allocated by the library. Once the strings are no longer needed, the application needs to free the spaces to avoid the memory leak. [14]

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

<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: $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: \$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: $startRecord

The record number at which to begin searching for the last allocated record.

out: \$nextRecord

The number of the last allocated record.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_BLOCKINGFACTOR_>[15]

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: \$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_>

---

[14] The function: CDF_Free_String (long numStrings, char **strings) can be called with the returned number of strings and pointer to the string array to free the spaces.

[15] The item rVAR_BLOCKINGFACTOR was previously named rVAR_EXTENDRECS.

Inquires the compression type/parameters and the compression percentage of the current rVariable (in the current CDF). The compression percentage is the result of the compressed size from all variable records divided by its original, uncompressed varible size. Required arguments are as follows:

out: \$cType

The compression type. The types of compressions are described in Section 2.10.

out: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

out: \$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: \$value

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed in the variable 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: \$dataType

Data type. The data types are described in Section 2.5.

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: \@dimVarys

Dimension variances. Each element of dimVarys receives the corresponding dimension variance. The variances are described in Section 2.9.

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: \@buffer

Values. The values are read from the CDF and placed in the variable 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: \$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: \$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: \$varName

Name of the rVariable.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_nINDEXENTRIES_>
Inquires the number of index entries for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. 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: \$numEntries

Number of index entries.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_nINDEXLEVELS_>
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: \$numLevels

Number of index levels.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_nINDEXRECORDS_>

Inquires the number of index records for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. 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: \$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: \$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: \$varName

The rVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

out: \$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: \$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: \$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: \$value

Pad value. The pad value is read from the CDF and placed in the variable value.

The required preselected objects/states are the current CDF and its current rVariable.

<GET_,rVAR_RECVARY_>

Inquires the record variance of the current rVariable (in the current CDF). Required arguments are as follows:

out: \$recVary

Record variance. The variances are described in Section 2.9.

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:

out: \$value

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed in the variable 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:

out: \$sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 2.11.2.

out: \@ArraysParms

The sparse arrays parameters. The sparse arrays parameters are described in Section 2.11.2.

out: \$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:

out: \$sRecordsType

The sparse records type. The types of sparse records are described in Section 2.11.1.

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: \@dimSizes

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: \$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: \$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: $numVars

The number of rVariables from which to read. This must be at least one (1).

in: \@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: \@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 in interpreting the buffer data after the it returns from the call. For example, a read operation for a full record for 3 rVariables, each a 2-dimensional (2 by 3), the buffer should have 18 elements after the read. As all variables' have the same number of data values, then the buffer should return with 18 elements (2*3 + 2*3 + 2*3), the first 6 for the first variable, the next 6 for the second variable and the last 6 for the third variable.

The required preselected objects/states are the current CDF and its current record number for rVariables. [16]

<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: \$text

Text explaining the status code.

The only required preselected object/state is the current status code.

<GET_,zENTRY_DATA_>[17]

Reads the zEntry data value from the current attribute at the current zEntry number (in the current CDF). Required arguments are as follows:

out: \$value

Value. This buffer must be large to hold the value. The value is read from the CDF and placed in the variable 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: \$dataType

Data type. The data types are described in Section 2.5.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

[16] A Standard Interface CDFgetrVarsRecordDatabyNumbers provides the same functionality.
[17] If the data is a string with with **"\N "** symbol (3 characters), it is an array of strings. Use GET_, zENTRY_STRINGDATA_ method (described in the following statement).

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: \$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_,zENTRY_NUMSTRINGS_>
Inquires the number of strings (of CDF_CHAR or CDF_UCHAR data type) of the zEntry at the current zEntry number for the current attribute (in the current CDF). Required arguments are as follows:

out: \$numStrings

Number of strings of the character data type. It is only for character data types (CDF_CHAR and CDF_UCHAR). Strings are concatenated and stored in the CDF in a sequence of charaters, with a pre-defined delimiter (**"\N** "), separating the strings. The number of elements for this character data type contains the extra characters used for the delimiter. [18]

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_,zENTRY_STRINGDATA_>
Reads the strings (of CDF_CHAR or CDF_UCHAR data type) of the zEntry at the current zEntry number for the current attribute (in the current CDF). Required arguments are as follows:

out: \@strings

An array of retrieved strings. Spaces for the strings are dynamically allocated by the library. Once the strings are no longer needed, the application needs to free the spaces to avoid the memory leak. [19]

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

<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: \$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.

[18] This feature is added in CDF V3.7.0. CDFs of previously versions only allow one single string.
[19] The function: CDF_Free_String (long numStrings, char **strings) can be called with the returned number of strings and pointer to the string array to free the spaces.

out: \$nextRecord

The number of the next allocated record.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_ALLOCATEDTO_>
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: $startRecord

The record number at which to begin searching for the last allocated record.

out: \$nextRecord

The number of the last allocated record.

The required preselected objects/states are the current CDF and its current zVariable.

<GET_,zVAR_BLOCKINGFACTOR_>[20]
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: \$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 and compression percentage of the current zVariable (in the current CDF). The compression percentage is the result of the compressed size from all variable records divided by its original, uncompressed varible size. Required arguments are as follows:

out: \$cType

The compression type. The types of compressions are described in Section 2.10.

out: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

out: \$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_>
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:

[20] The item zVAR_BLOCKINGFACTOR was previously named zVAR_EXTENDRECS .

out: \$value

Value. The value is read from the CDF and placed in the variable 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: \$dataType

Data type. The data types are described in Section 2.5.

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: \@dimSizes

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: \@dimVarys

Dimension variances. Each element of dimVarys receives the corresponding dimension variance. The variances are described in Section 2.9.

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: \@buffer

Values. The values are read from the CDF and placed in the variable 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: \$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: \$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: \$varName

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: \$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: \$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: \$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: \$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: $varName

The zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters (excluding the NUL terminator).

out: \$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: \$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: \$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: \$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: \$value

Pad value. The pad value is read from the CDF and placed in the variable 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: \$recVary

Record variance. The variances are described in Section 2.9.

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: \$value

Value. The value is read from the CDF and placed in the variable 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: \$sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 2.11.2.

out: \@sArraysParms

The sparse arrays parameters.

out: \$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: \$sRecordsType

The sparse records type. The types of sparse records are described in Section 2.11.1.

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: \$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: $numVars

The number of zVariables from which to read. This must be at least one (1).

in: \@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: \@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 in interpreting the buffer data after the it returns from the call. For example, a read operation for a full record for 3 zVariables, first a 2-dimensional (2 by 3), second as a 1-dimensional (3) and third a scalar, the buffer should have 10 (2*3 + 3 + 1) elements after the read. Among them, the first 6 for the first variable, the next 3 for the second variable and the last 1 for the third variable.

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_>). [21]

<NULL_>
Marks the end of the argument list that is passed to An internal interface call. No other arguments are allowed after it.

<OPEN ,CDF_>
Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

in: $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.

out: \$id

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

There are no required preselected objects/states.

<PUT_,ATTR_NAME_>
Renames the current attribute (in the current CDF). An attribute with the same name must not already exist in the CDF. Required arguments are as follows:

in: $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.

<PUT_,ATTR_SCOPE_>
Respecifies the scope for the current attribute (in the current CDF). Required arguments are as follows:

in: $scope

New attribute scope. Specify one of the scopes described in Section 2.12.

The required preselected objects/states are the current CDF and its current attribute.

<PUT_,CDF_CHECKSUM__>_
Respecifies the checksum mode of the current CDF. Required arguments are as follows:

in: $checksum

The checksum mode to be used (NO_CHECKSUM or MD5_CHECKSUM). The checksum mode is described in Section 2.19.

The required preselected objects/states is the current CDF.

[21] A Standard Interface CDFgetzVarsRecordDatabyNumbers provides the same functionality.

<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: $cType

The compression type. The types of compressions are described in Section 2.10.

in: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

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: $encoding

New data encoding. Specify one of the encodings described in Section 2.6.

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: $format

New CDF format. Specify one of the formats described in Section 2.4.

The only required preselected object/state is the current CDF.

<PUT_,CDF_LEAPSECONDLASTUPDATED_>

Respecifies the variable lastupdated of the current CDF. A CDF's lastupdated, the leap second last updated date, may not be set in the older CDFs. The last uppdated date must be a valid entry in the currently used leap second table, oe zero (0). When 0, the CDF was made without using the table. Required arguments are as follows:

in: $lastupdated

The value has to be in YYYYMMDD form.

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: $majority

New variable majority. Specify one of the majorities described in Section 2.8.

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: $dataType

Data type of the gEntry. Specify one of the data types described in Section 2.5.

in: $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: \$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: $dataType

New data type of the gEntry. Specify one of the data types described in Section 2.5.

in: $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: $dataType

Data type of the rEntry. Specify one of the data types described in Section 2.5.

in: $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: \$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: $dataType

New data type of the rEntry. Specify one of the data types described in Section 2.5.

in: $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_,rENTRY_STRINGDATA_>

Write an array of strings (of CDF_CHAR or CDF_UCHAR data type) of the rEntry at the current rEntry number for the current attribute (in the current CDF). Required arguments are as follows:

in: \@strings

An array of strings.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

<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: $firstRecord

The first record number to allocate.

in: $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: $nRecords

Number of records to allocate.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_,rVAR_BLOCKINGFACTOR_>[22]

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:

in: $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:

in: $cType

The compression type. The types of compressions are described in Section 2.10.

in: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

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:

in: \$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:

[22] The item rVAR_BLOCKINGFACTOR was previously named rVAR_EXTENDRECS .

in: $dataType

New data type. Specify one of the data types described in Section 2.5.

in: $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: \@dimVarys

New dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 2.9.

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: \@buffer

Values. The values in the variable 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: $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: $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: \$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: $recVary

New record variance. Specify one of the variances described in Section 2.9.

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: \$value

Value. The value is written to the CDF from the variable 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: $sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 2.11.2.

in: \@sArraysParms

The sparse arrays parameters. The sparse arrays parameters are described in Section 2.11.2.

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: $sRecordsType

The sparse records type. The types of sparse records are described in Section 2.11.1.

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: $numVars

The number of rVariables to which to write. This must be at least one (1).

in: \@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: \@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 in setting up the buffer. Make sure that the buffer contains the same number of data values from all variables involved. For examples, if the variables are all 2-dimensional (2 by 3) array, then the buffer should have 18 elements (2*3 + 2*3 + 2*3) for handling a process of three variables. Among them, the first 6 is from the first variable, the next 6 from the second variable and the last 6 from the third variable.

The required preselected objects/states are the current CDF and its current record number for rVariables. [23]

<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: dataType

Data type of the zEntry. Specify one of the data types described in Section 2.5.

in: $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: \$value

[23] A Standard Interface CDFputrVarsRecordDatabyNumbers provides the same functionality.

Value(s). The entry value is written to the CDF from the variable 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:

in: $dataType

New data type of the zEntry. Specify one of the data types described in Section 2.5.

in: $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_,zENTRY_STRINGDATA_>
Write an array of strings (of CDF_CHAR or CDF_UCHAR data type) of the zEntry at the current zEntry number for the current attribute (in the current CDF). Required arguments are as follows:

in: \@strings

An array of strings.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

<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: $firstRecord

The first record number to allocate.

in: $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: $nRecords

Number of records to allocate.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_BLOCKINGFACTOR_>[24]
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: $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:

in: $cType

The compression type. The types of compressions are described in Section 2.10.

in: \@cParms

The compression parameters. The compression parameters are described in Section 2.10.

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: \$value

Value. The value is written to the CDF from the variable 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: $dataType

New data type. Specify one of the data types described in Section 2.5.

in: $numElements

[24] The item zVAR_BLOCKINGFACTOR was previously named zVAR_EXTENDRECS .

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: \@dimVarys

New dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 2.9.

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: $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: \@buffer

Values. The values at the variable 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: $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: \$value

Pad value. The pad value is written to the CDF from the variable 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: $recVary

New record variance. Specify one of the variances described in Section 2.9.

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: \$value

Value. The value is written to the CDF from the variable 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: $sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 2.11.2.

in: \@sArraysParms

The sparse arrays parameters. The sparse arrays parameters are described in Section 2.11.2.

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: $sRecordsType

The sparse records type. The types of sparse records are described in Section 2.11.1.

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: $numVars

The number of zVariables to which to write. This must be at least one (1).

in: \@varNums

The zVariables to which to write. This array, whose size is determined by the value of numVars, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: \@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 in setting up the buffer. Make sure that the buffer contains the same number of data values from all variables involved. For examples, if the first variable is a 2-dimensional (2 by 3) array and the second variable is a 1-dimensional (5 elements) and the third variable is a scalar, then the buffer should have 12 elements (2*3 + 5 + 1), the first 6 from the first variable, the next 5 from the second variable and the last one from the third variable, while passing into the CDFlib.

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_>). [25]

<SELECT_,ATTR_>

Explicitly selects the current attribute (in the current CDF) by number. Required arguments are as follows:

in: $attrNum

Attribute number.

The only required preselected object/state is the current CDF.

<SELECT_,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:

in: $attrName

[25] A Standard Interface CDFputzVarsRecordDatabyNumbers provides the same functionality.

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.

<SELECT_,CDF_>
Explicitly selects the current CDF. Required arguments are as follows:

in: $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.

<SELECT_,CDF_CACHESIZE_>
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:

in: $numBuffers

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: $decoding

The decoding. Specify one of the decodings described in Section 2.7.

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: $mode

The -0.0 to 0.0 mode. Specify one of the -0.0 to 0.0 modes described in Section 2.15.

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: $mode

The read-only mode. Specify one of the read-only modes described in Section 2.13.

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: $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: $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:

in: $mode

The zMode. Specify one of the zModes described in Section 2.14.

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: $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: $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: $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: $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: $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: $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: $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_RESERVEPERCENT_>
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: $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: $recNum

Record number.

in: \@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: $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: \@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: \@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: \@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: $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: $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: $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: $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: $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: $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: $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: $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: \@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: \@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: \@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: $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: $recCount

Record count.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_,zVAR_RECINTERVAL_>
Selects the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

in: $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: $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: $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: $recNum

Record number.

in: \@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: $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: $recNum

Record number.

The only required preselected object/state is the current CDF.

4.7 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_>).

4.7.1 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.

```
.
.
.
my      $status;                # Status returned from CDF library.
my      @dimVarys;              # Dimension variances.
my      $varNum;                # rVariable number.
my      $padValue = -999.9;     # Pad value.

.
.
$dimVarys[0] = VARY;
```

```
$dimVarys[1] = VARY;
$status = CDF::CDFlib (CREATE_, rVAR_, "HUMIDITY", CDF_REAL4, 1, VARY, \@dimVarys, \$varNum,
                       PUT__, rVAR_PADVALUE_, \$padValue,
                              rVAR_INITIALRECS_, 500,
                              rVAR_BLOCKINGFACTOR_, 50,
                       NULL_);
if ($status != CDF_OK) UserStatusHandler ($status);
.
.
```

4.7.2 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.

```
.
.
.
my      $status;                        #  Status returned from CDF library.
my      @dimVarys;                      #  Dimension variances.
my      $varNum;                        #  zVariable number.
my      $numDims = 1;                   #  Number of dimensions.
my      @dimSizes = { 20 };             #  Dimension sizes.
my      $numElems = 10;                 #  Number of elements (characters in this case).
my      $padValue = "**********";       #  Pad value.
.
.
$dimVarys[0] = VARY;
$status = CDF::CDFlib (CREATE_, zVAR_, "Station", CDF_CHAR, $numElems, $numDims,
                                       \@dimSizes, NOVARY, \@dimVarys, \$varNum,
                       PUT__, zVAR_PADVALUE_, \$padValue,
                       NULL_);
if ($status != CDF_OK) UserStatusHandler ($status);
.
.
```

4.7.3 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.

```
.
.
.
my      $status;                        #  Status returned from CDF library.
my      @values;                        #  Buffer to receive values.
my      $recCount = 1;                  #  Record count, one record per hyper get.
my      $recInterval = 1;               #  Record interval, set to one to indicate contiguous records
                                        #  (really meaningless since record count is one).
```

```
my      @indices = (0,0);               #  Dimension indices,  start  each read at 0,0 of the array.
my      @counts = (50,100);             #  Dimension counts,  half of the values along
                                        #  each dimension will be read.
my      @intervals = (2,2);             #  Dimension intervals,  every  other value along
                                        #  each dimension will be read.
my      $recNum;                        #  Record number.
my      $maxRec;                        #  Maximum  rVariable record number in the CDF  - This was
                                        #  determined with a call to CDFinquire.
.
.
$status =  CDF::CDFlib  (SELECT_,  rVAR_NAME_,  "BRIGHTNESS",
                                   rVARs_RECCOUNT_,  $recCount,
                                   rVARs_RECINTERVAL_,  $recInterval,
                                   rVARs_DIMINDICES_, \@indices,
                                   rVARs_DIMCOUNTS_,  \@counts,
                                   rVARs_DIMINTERVALS_, \@intervals,
                         NULL_);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);

for ($recNum  =  0;  $recNum  <=  $maxRec;  $recNum++)  {
      $status =  CDF::CDFlib  (SELECT_,  rVARs_RECNUMBER_,  $recNum,
                               GET_,  rVAR_HYPERDATA_,  \@values,
                               NULL_);
      if ($status !=  CDF_OK)  UserStatusHandler  ($status);
      .
      .
         process  values
      .
      .
}
.
.
```

4.7.4 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.

```
.
.
.
my      $status;                    #  Status  returned from CDF library.
.
.
$status =  CDF::CDFlib  (SELECT_, ATTR_NAME_,  "Tmp",
                         PUT__, ATTR_NAME,  "TMP",
                         NULL_);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);
.
.
```

4.7.5 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.

```
.
.
my      $status;                # Status returned from CDF library.
my      $varNum;                # zVariable number.
my      $recNum = 0;            # Record number, start at first record.
my      @indices = (0,0);       # Dimension indices.
my      $value;                 # Value read.
my      $sum = 0.0;             # Sum of all values.
my      $count = 0;             # Number of values.
my      $ave;                   # Average value.
.
.
$status = CDF::CDFlib (GET_, zVAR_NUMBER_, "FLUX", \$varNum,
                       NULL_);
if (status != CDF_OK) UserStatusHandler ($status);
$status = CDF::CDFlib (SELECT_, zVAR_, $varNum,
                                zVAR_SEQPOS_, $recNum, \@indices,
                       GET_, zVAR_SEQDATA_, \$value,
                       NULL_);

while ($status _>= CDF_OK) {
        $sum += $value;
        $count++;
        $status = CDF::CDFlib (GET_, zVAR_SEQDATA_, \$value,
                               NULL_);
}
if ($status != END_OF_VAR) UserStatusHandler ($status);

$ave = $sum / $count;
.
.
```

4.7.6 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.

```
.
.
.
my      $status;                    # Status returned from CDF library.
my      @scale = (-90.0,90.0);      # Scale, minimum/maximum.
.
.
$status = CDF::CDFlib (SELECT_, rENTRY_NAME_, "LATITUDE",
```

```
                                ATTR_NAME_,  "FIELDNAM",
                            PUT__,  rENTRY_DATA_,  CDF_CHAR, 20, "Latitude          ",
                            SELECT_, ATTR_NAME_,  "SCALE",
                            PUT__,  rENTRY_DATA_,  CDF_REAL, 4, 2,  \@scale,
                            SELECT_, ATTR_NAME_,  "UNITS",
                            PUT__,  rENTRY_DATA_,  CDF_CHAR, 20, "Degrees north     ",
                            NULL_);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);
.
.
```

4.7.7 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.

```
.
.
.
my      $status;                #  Status  returned from CDF library.
my      $time;                  #  `Time'  (short) value.
my      $vectorA;               #  `vectorA'  (characters of 3) values.
my      @vectorB;               #  `vectorB'  (5 doubles) values.
my      $recNumber;             #  Record number.
my      @buffer;                #  Buffer of full-physical records.
my      @varNumbers;            #  Variable numbers.
.
.
$status =  CDF::CDFlib  (GET_,  zVAR_NUMBER_,  "vectorB",  \$varNumbers[0],
                                zVAR_NUMBER_,  "time",  \$varNumbers[1],
                                zVAR_NUMBER_,  "vectorA",  \$varNumbers[2],
                            NULL_);
if ($status !=  CDF_OK)  UserStatusHandler  ($status);
.
.
my $ii;
for ($recNumber =  0;  $recNumber <  100;  $recNumber++)  {
    .
      read  values from input file
    .
    for ($ii = 0; $ii < 5; $ii++) {
      $buffer[$ii] =  $vectorB[$ii];
    }

    $buffer[5] = $time;
    $buffer[6] = $vectorA;
    $status =  CDF::CDFlib  (SELECT_,  zVARs_RECNUMBER_,  $recNumber,
                             PUT__,  zVARs_RECDATA_,  3,  \@varNumbers,  \@buffer,
                             NULL_);
    if ($status !=  CDF_OK)  UserStatusHandler  ($status);
}
.
.
```

Note that it would be more efficient to read the values directly into buffer. The method shown here was used to illustrate how to create the buffer of full-physical records.

4.8 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 Perl application. Please don't do something like the following:

```
.
.
.
my      $id;                    #  CDF  identifier  (handle).
my      $status;                #  Status  returned from CDF library.
my      $varNum;                #  zVariable number.
.
.
$status =  CDF::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 varNum is passed by value rather than reference.[26] 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.

[26] Fortran programmers can get away with doing something like this because everything is passed by reference.

Chapter 5

5 Quick Interface

The Quick Interface functions described in this chapter represent a set of easy to use functions that are based on Internal Interface to acquire information from the CDF library, a CDF file, its variable and variable data. Normally, a few basic calls to Internal Interface calls are needed to accomplish a function. These functions perform the basics for users so the data can be returned easily. Any function in this Interface that deals with a variable, either its data or attribute, applies only to zVariable.

5.1 CDFgetLIBInfo

($status, %info) = CDF::CDFgetLIBInfo();

CDFgetLIBInfo returns the currrent library version, release and increment information, along with the leap second table information. The latest leap second added to the leap second table is returned. Both fields are presented in a Perl Hash object.

No arguments to CDFgetLIBInfo are needed.

5.1.1 Example(s)

The following example shows what the current CDF library and the leap second table are used.

```
    .
    .
    .
    my   $status;                                  #   Returned status code.
    my   %info;                                    #   Returned hash.
    .
    .
    ($status, %info) =  CDF::CDFgetLIBInfo();
    UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
    print Dumper(\%info);

$VAR1 = {
        'LATEST_LEAPSECOND_IN_TABLE' => 20170101,
```

```
      'LIB_VERSION' => '3.8.0'
    };
  .
  .
```

5.2 CDFgetCDFInfo

($status, %info) = CDF::CDFgetCDFInfo($cdf);

CDFgetCDFInfo returns the information about a specified CDF. The information includes the name (if available), its format, majority, encoding, file version, the number of global and variable attributes, the number of rVariables and zVariables. It also has the time tag about the leap second table that this file is based on (only applicable if the CDF has TT2000 epoch data type variables). All fields are presented in a Perl Hash object.

The argument to CDFgetCDFInfo is defined as follow:

cdf	The cdf name or identifier. The identifier is from a CDF open or created process. If the argument is a CDF name, the file is open and then closed after the information is acquired. For an identifier, the file will not be closed after the information is returned.

5.2.1 Example(s)

The following example shows the information for a given CDF.

```
  .
  .
  .
  my   $status;                          #   Returned status code.
  my   %info;                            #   Returned hash.
  .
  .
  ($status, %info) =  CDF::CDFgetCDFInfo("testfile");
  UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
  print Dumper(\%info);

$VAR1 = {
      'FORMAT' => 'SINGLE',
      'NAME' => 'testfile',
      'BASED_LEAPSECOND_LAST_UPDATED' => 20170101,
      'NUMgATTRS' => 5,
      'MAJORITY' => 'ROW',
      'NUMrVARS' => 0,
      'ENCODING' => 'IBMPC',
      'CDF_VERSION' => '3.8.0',
      'NUMzVARS' => 22,
      'NUMvATTRS' => 7,
    };
  .
  .
```

5.3 CDFgetGlobalMetaData

($status, %info) = CDF::CDFgetGlobalMetaData($cdf [, \@globalids] [, $metaEncoding]);

CDFgetGlobalMetaData returns all or a list of selected global attributes from the spcified CDF. All fields are presented in a Perl Hash object. All data of epoch data types will be encoded accordingly.

The arguments to CDFgetGlobalMetaData is defined as follows:

cdf	The cdf name or identifier. If the CDF name is specified, the CDF will be open and then closed after the information is acquired. The identifier is from an open or created CDF process. The file will not be closed when the information is returned.

Optionally,

globalids	A list of selected global attribute names or identifiers to be read. For reading all attributes, passing in a reference to an empty array. If the followinging attribute: metatEncoding is specified, the this argument is required.

Optionally,

metaEncoding	Whether to encode the CDF epoch data into date/time string form. With a value of 1, the default, the epoch data is encoded. With a value of 0 or this argument not provided, the data will return as is.

5.3.1 Example(s)

The following example first shows all of the global attributes from a given CDF, testfile. The second example reads only 2 global variables: “Global1” and “Global3”.

```
    .
    .
    .
    my   $status;                                  #   Returned status code.
    my   %info;                                    #   Returned hash.
    .
    .
    ($status, %info) =  CDF::CDFgetGlobalMetaData(“testfile”);
    UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
    print Dumper(\%info);

$VAR1 = {
        'epTestDate' => {
                          '0' => '2004-05-13T15:08:11.022033044055'
                        },
        'Test' => {
                    '11' => 4294967295,
                    '7' => 1,
                    '2' => '5.5',
                    '0' => '5.3432',
                    '16' => 255,
                    '13' => 65535,
                    '6' => -32768,
                    '3' => '5.5',
```

```
            '9' => 4,
            '12' => 4294967295,
            '14' => 65535,
            '15' => 255,
            '8' => 3,
            '4' => 1,
            '10' => 'This is a string',
            '5' => 1
           },
    'Project' => {
              '0' => 'Using the CDFJava API'
            },
    'PI' => {
           '3' => 'Ernie Els'
         },
    'TestDate' => {
              '1' => '2002-04-25T00:00:00.000',
              '2' => '2008-02-04T06:08:10.012014016'
            }
  };

.
.
my $id;
$status = CDF::CDFlib(&OPEN_, &CDF_, “tesfile”, \$id, &NULL_);
my @globals = (“Global1”, “Global2”);
($status, %info) =  CDF::CDFgetGlobalMetaData($id, \@globals);
UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
```

5.4 CDFgetVarInfo

($status, %info) = CDF::CDFgetVarInfo($cdf, [$varName | $varid]);

CDFgetVarInfo returns specification about a given variable from the spcified CDF. The information includes its name, data type, number of elements, number of dimensions, dimensional sizes and variances if its dimension > 0, record variance, and last written record number (as MaxRec). All fields are presented in a Perl Hash object. All data of epoch data types will be encoded accordingly. zMODEon2 will be selected to present the dimensionality of rVariables, which eliminates all non-varying dimensions. As the CDF is selected as **zMODE2**, where all varaibles are handled as zVariables, the zVariable numbers need to be properly set if the CDF has both rVariables and zVariables. For example, a CDF with 3 rVariables, the first zVariable number should be 3 (as 0 + 3), the second zVariable shule be 4 (as 1 + 3).

The arguments to CDFgetVarInfo are defined as follows:

cdf	The CDF file name or an identifier of an open/created CDF. If a file name is specified, the file will be closed after the information is returned.
varName	The variable name, the preferred form.
Or,	
varid	The identifier of a zVariable.

5.4.1 Example(s)

The following example shows the information from variable: “Latitude” in a CDF: “testfile”.

```
    .
    .
    my   $id;                                   #  CDF identifier
    my   $status;                               #  Returned status code.
    my   %info;                                 #  Returned hash.
    .
    .
    $status = CDF::CDFlib(&OPEN_, &CDF_, “testfile”, \$id, &NULL_);
    ($status, %info) =  CDF::CDFgetVarInfo($id, “Latitude”);
    UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
    print Dumper(\%info);

$VAR1 = {
        'VarName' => 'Latitude',
        'NumElems' => 1,
        'RecVary' => 'False',
        'NumDims' => 1,
        'DimVarys' => {
                    '0' => 'True'
                  },
        'DataType' => 'CDF_INT1',
        'DimSizes' => {
                    '0' => 3
                  },
        'MaxRec' => 0
      };

    .
    .
```

5.5 CDFgetVarAllData

($status, @data) = CDF::CDFgetVarAllData($id, [$varName | $varid] [,$dataEncoding][,$matrix]);

CDFgetVarAllData returns all data for a given variable from the spcified CDF in an array object. The variable can be entered as a name or number. As the CDF is selected as **zMODE2**, where all varaibles are handled as zVariables, the zVariable numbers need to be properly set if the CDF has both rVariables and zVariables. For example, a CDF with 3 rVariables, the first zVariable number should be 3 (as 0 + 3), the second zVariable shule be 4 (as 1 + 3).

The arguments to CDFgetVarAllData are defined as follows:

cdf	The CDF file name or an identifier of an open/created CDF. If a file name is specified, the file will be closed after the information is returned.
varName	The variable name, the preferred form.
Or,	
varid	The identifier of a zVariable.

Optionally,

dataEncoding	Whether to encode the CDF epoch data into date/time string. With a value of 1, the epoch data is encoded. With a value of 0 or this argument not provided, the data will return as is. If the following argument, matrix, is provided, then this argument needs to be provided as well.
matrix	With this option, the returned data is presented in a matrixized form reflecting the variable's dimensions if a value is 1. If the variable has multiple data records, then an extra dimension as the first element is added. For a value of 0, the default, or without this argument, a simple vector of data is returned.

5.5.1 Example(s)

The following example reads all variable data, a total of 3 records, from variable: "Latitude1", a 1-D dimension of 3 elements with type: CDF_INT2, in a CDF: "testfile".

```
    .
    .
    my   $id;                                   #  CDF identifier
    my   $status;                               #  Returned status code.
    my   @data;                                 #  Returned data.
    .
    $status = CDF::CDFlib(&OPEN_, &CDF_, "testfile", \$id, &NULL_);

    ($status, @data) =  CDF::CDFgetVarAllData($id, "Latitude");
    UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
    print Dumper(\@data);

$VAR1 = [
        254,
        254,
        5,
        15,
        25,
        35,
        100,
        128,
        255
      ];

    ($status, @data) =  CDF::CDFgetVarAllData($id, "Latitude", 0, 1);
    UserStatusHandler  ("1.0". $status) if ($status <  CDF_OK);
    print Dumper(\@data);

$VAR1 = [
        [
          254,
          254,
          5
        ],
        [
          15,
```

```
      25,
      35
    ],
    [
      100,
      128,
      255
    ]
  ];

.
.
```

5.6 CDFgetVarMetaData

($status, %info) = CDF::CDFgetVarMetaData($id, [$varName | $varid] [,$metaEncoding]);

CDFgetVarMetaData returns all the variable attribute information for a given variable from the spcified CDF. All fields are presented in a Perl Hash object. All data of epoch type type will be encoded accordingly. As the CDF is selected as **zMODE2**, where all varaibles are handled as zVariables, the zVariable numbers need to be properly set if the CDF has both rVariables and zVariables. For example, a CDF with 3 rVariables, the first zVariable number should be 3 (as 0 + 3), the second zVariable shule be 4 (as 1 + 3).

The arguments to CDFgetVarMetaData are defined as follows:

id	The CDF file name or an identifier of an open/created CDF. If a file name is specified, the file will be closed after the information is returned.
varName	The variable name, the preferred form.
Or,	
varid	The identifier of a zVariable.
Optionally,	
metaEncoding	Whether to encode the CDF epoch data into date/time string form. With a value of 1, the epoch data is encoded. With a value of 0 or this argument not provided, the data will return as is.

5.6.1 Example(s)

The following example shows the variable attributes from variable: “Latitude” in a given CDF.

```
.
.
my   $id;                                   #  CDF identifier
my   $status;                               #  Returned status code.
my   %info;                                 #  Returned hash.
.
.
($status, %info) =  CDF::CDFgetVarMetaData($id, “Latitude”);
```

```
    UserStatusHandler  (“1.0”. $status) if ($status <  CDF_OK);
    print Dumper(\%info);

$VAR1 = {
          'FILLVAL' => 111,
          'validmin' => 20,
          'VALIDMAX' => 90
        };

    .
    .
```

6 Interpreting CDF Status Codes

Most CDF functions return a status code. 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.

status > CDF_OK	Indicates successful completion but some additional information is provided. These are informational codes.
status = CDF_OK	Indicates successful completion.
CDF_WARN < status < CDF_OK	Indicates that the function completed but probably not as expected. These are warning codes.
status < CDF_WARN	Indicates that the function did not complete. These are error codes.

The following example shows how you could check the status code returned from CDF functions.

```
my    $status;

.
.
$status =  CDF::CDFfunction (...);                  #  any CDF function returning status
if ($status !=  CDF_OK)  {
   UserStatusHandler  (“1.0”, $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.

```
sub  UserStatusHandler  {
  my ($where, $status)=@_;

  print "Aborting at $where ...\n";
  if ($status < CDF_OK) {
    my $text;
    CDF::CDFlib (SELECT_, CDF_STATUS_, $status,
                 GET_, STATUS_TEXT_, \$text,
                 NULL_);
    print $text;
  }
  CDF::CDFlib (CLOSE_, CDF_,
               NULL_);
  print "...test aborted.\n";
  exit;
```

```
}#endsub QuitCDF
```

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 6

7 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 01-Jan-0000 00:00:00.000 and 01-Jan-0000 00:00:00.000.000.000.000, respectively.

7.1 computeEPOCH

computeEPOCH calculates a single CDF_EPOCH value, given the individual components, or an array of values, given an array of individual components . If an illegal component is detected, the value returned will be ILLEGAL_EPOCH_VALUE.

```
CDF::computeEPOCH(          # out -- CDF_EPOCH value returned.
    my $year,               # in -- Year (AD, e.g., 1994).
    my $month,              # in -- Month (1-12).
    my $day,                # in -- Day (1-31).
    my $hour,               # in -- Hour (0-23).
    my $minute,             # in -- Minute (0-59).
    my $second,             # in -- Second (0-59).
    my $msec);              # in -- Millisecond (0-999).

CDF::computeEPOCH(          # out -- an array of CDF_EPOCH values returned.
    my \@year,              # in -- Years (AD, e.g., 1994).
    my \@month,             # in -- Months (1-12).
    my \@day,               # in -- Days (1-31).
    my \@hour,              # in -- Hours (0-23).
    my \@minute,            # in -- Minutes (0-59).
    my \@second,            # in -- Seconds (0-59).
    my \@msec);             # in -- Milliseconds (0-999).
```

NOTE: There are two variations on how computeEPOCH may be used. If the month argument is 0 (zero), then 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 0 (zero), then the msec argument is assumed to be the millisecond of the day having a range of 0 through 86400000.

7.2 EPOCHbreakdown

EPOCHbreakdown decomposes a single CDF_EPOCH value or an array of CDF_EPOCH values into the individual components.

```
CDF::EPOCHbreakdown(
    my      $epoch,             #  in  --  The CDF_EPOCH  value.
    my      $year,              #  out  --  Year  (AD,  e.g.,  1994).
    my      $month,             #  out  --  Month  (1-12).
    my      $day,               #  out  --  Day  (1-31).
    my      $hour,              #  out  --  Hour  (0-23).
    my      $minute,            #  out  --  Minute  (0-59).
    my      $second,            #  out  --  Second  (0-59).
    my      $msec);             #  out  --  Millisecond  (0-999).

CDF::EPOCHbreakdown(
    my      \@epoch,            #  in  --  Array of CDF_EPOCH  values.
    my      \@year,             #  out  --  Years (AD,  e.g.,  1994).
    my      \@month,            #  out  --  Months  (1-12).
    my      \@day,              #  out  --  Days  (1-31).
    my      \@hour,             #  out  --  Hours  (0-23).
    my      \@minute,           #  out  --  Minutes  (0-59).
    my      \@second,           #  out  --  Seconds  (0-59).
    my      \@msec);            #  out  --  Milliseconds  (0-999).
```

7.3 toEncodeEPOCH

toEncodeEPOCH encodes a single CDF_EPOCH value or an array of CDF_EPOCH values into one of the defined date/time character string(s), based on the passed style. The style is one of the following values:

-**0**: **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).

-**1**: **yyyymmdd.tttttt** where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), tttttt is a fraction of day.

-**2**: **yyyymmddhhmnss** 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), mn is the minute (0-59), ss is the second (0-59).

-**3**: **yyyy-mm-ddThh:mn:ss.cccZ** 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), ss is the second (0-59), and ccc is the millisecond (0-999).

-**4**: **yyyy-mm-ddThh:mn:ss.ccc** 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), mn is the minute (0-59), ss is the second (0-59), and ccc is the millisecond (0-999).

```
CDF::toEncodeEPOCH(
    my      $epoch;                 #  in  --  The CDF_EPOCH  value.
    my      $style;                 #  in  --  The encoded style.
    my      $epString);             #  out  --  The standard  date/time character string.

CDF::toEncodeEPOCH(
    my      \@epoch;                #  in  --  Array of CDF_EPOCH  values.
    my      $style;                 #  in  --  The encoded style.
    my      \@epString);            #  out  --  The standard  date/time character string array.
```

Each epString has the length of EPOCH_STRING_LEN, EPOCH1_STRING_LEN, EPOCH2_STRING_LEN, EPOCH3_STRING_LEN or EPOCH4_STRING_LEN, defined in Perl-CDF package. Passing style 0 is similar to calling encodeEPOCH. Passing style 1 is similar to calling encodeEPOCH1. Passing style 2 is similar to calling encodeEPOCH2. Passing style 3 is similar to calling encodeEPOCH3. Passing style 4 is similar to calling encodeEPOCH4. Style 4 calling is the currently most used, as it is comforming to ISO 8601 format.

7.4 encodeEPOCH

encodeEPOCH encodes a single CDF_EPOCH value or an array of CDF_EPOCH values into the standard date/time character string(s). 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).

```
CDF::encodeEPOCH(
    my      $epoch;                 #  in  --  The CDF_EPOCH  value.
    my      $epString);             #  out  --  The standard  date/time character string.

CDF::encodeEPOCH(
    my      \@epoch;                #  in  --  Array of CDF_EPOCH  values.
    my      \@epString);            #  out  --  The standard  date/time character string array.
```

Each epString has the length of EPOCH_STRING_LEN, defined in Perl-CDF package.

7.5 encodeEPOCH1

encodeEPOCH1 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmdd.ttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and ttttttt is the fraction of the day (e.g., 5000000 is 12 o'clock noon).

```
CDF::encodeEPOCH1(
    my      $epoch;                 #  in  --  The CDF_EPOCH  value.
    my      $epString);             #  out  --  The alternate  date/time character string.
```

epString has a length of EPOCH1_STRING_LEN.

7.6 encodeEPOCH2

encodeEPOCH2 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymoddhhmmss 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), and ss is the second (0-59).

```
CDF::encodeEPOCH2(
    my      $epoch;                     # in -- The CDF_EPOCH value.
    my      $epString);                 # out -- The alternate date/time character string.
```

spString has a length of EPOCH2_STRING_LEN.

7.7 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).

```
CDF::encodeEPOCH3(
    my      $epoch;                     # in -- The CDF_EPOCH value.
    my      \$epString);                # out -- The alternate date/time character string.
```

epString has a length of EPOCH3_STRING_LEN.

7.8 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).

```
CDF::encodeEPOCH4(
    my      $epoch;                     # in -- The CDF_EPOCH value.
    my      $epString);                 # out -- The ISO 8601 date/time character string.
```

epString has a length of EPOCH4_STRING_LEN.

7.9 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.

```
CDF::encodeEPOCHx(
    my      $epoch;                     # in -- The CDF_EPOCH value.
```

```
    my      $format;                    # in  ---The format string.
    my      $encoded);                  # out --  The custom  date/time character string.
```

The encoded string has a length up to EPOCHx_STRING_MAX. 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. . .

Token	Meaning	Default
dom	Day of month (1-31)	<dom.0>
doy	Day of year (001-366)	<doy.03>
month	Month (`Jan',`Feb',...,`Dec')	<month>
mm	Month (1,2,...,12)	<mm.0>
year	Year (4-digit)	<year.04>
yr	Year (2-digit)	<yr.02>
hour	Hour (00-23)	<hour.02>
min	Minute (00-59)	<min.02>
sec	Second (00-59)	<sec.02>
fos	Fraction of second.	<fos.3>
fod	Fraction of day.	<fod.8>

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 7.3) would be. . .

```
<dom.02>-<month>-<year>  <hour>:<min>:<sec>.<fos>
```

EPOCHx_FORMAT_LEN and EPOCHx_STRING_MAX are defined in cdf.h.

7.10 toParseEPOCH

toParseEPOCH parses a single or an array of encoded date/time string(s). The date/time string must conform to one of the encode styles (through the time encoding function for CDF_EPOCH value). This generalized function can be replaced all parsing functions for CDF_EPOCH type.

```
$epoch = CDF::toParseEPOCH(             # out  --  CDF_EPOCH  value returned.
    my      $epString);                 # in  --  The standard  date/time character string.

@epochs = CDF::toParseEPOCH(            # out  --  An array of CDF_EPOCH  values returned.
    my      \@epString);                # in  --  The standard  date/time character strings.
```

7.11 parseEPOCH

parseEPOCH parses a single, standard date/time character string or array of strings and returns a CDF_EPOCH value(s). The format of the string is that produced by the encodeEPOCH function described in Section 7.3. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```
$epoch = CDF::parseEPOCH(                # out -- CDF_EPOCH value returned.
    my       $epString);                 # in -- The standard date/time character string.

@epochs = CDF::parseEPOCH(               # out -- An array of CDF_EPOCH values returned.
    my       \@epString);                # in -- The standard date/time character strings.
```

7.12 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 7.4. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```
$epoch = CDF::parseEPOCH1(               # out -- CDF_EPOCH value returned.
    my       $epString);                 # in -- The alternate date/time character string.
```

7.13 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 7.6. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```
$epoch = CDF::parseEPOCH2(               # out -- CDF_EPOCH value returned.
    my       $epString);                 # in -- The alternate date/time character string.
```

7.14 parseEPOCH3

parseEPOCH3 parses an alternate, ISO8601 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 7.7. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```
$epoch = CDF::parseEPOCH3(               # out -- CDF_EPOCH value returned.
    my       $epString);                 # in -- The ISO8601 date/time character string.
```

7.15 parseEPOCH4

ParseEPOCH4 parses an alternate, ISO8601 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 7.8. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```
$epoch = CDF::parseEPOCH4(              #  out -- CDF_EPOCH value returned.
    my      $epString);                 #  in -- The ISO8601 date/time character string.
```

7.16 computeEPOCH16

computeEPOCH16 calculates a CDF_EPOCH16 value, a two-double array, given the individual components. If an illegal component is detected, the value returned will be ILLEGAL_EPOCH_VALUE.

```
$dummy = CDF::computeEPOCH16(           #  out -- status code returned.
    my      $year,                      #  in -- Year (AD, e.g., 1994).
    my      $month,                     #  in -- Month (1-12).
    my      $day,                       #  in -- Day (1-31).
    my      $hour,                      #  in -- Hour (0-23).
    my      $minute,                    #  in -- Minute (0-59).
    my      $second,                    #  in -- Second (0-59).
    my      $msec,                      #  in -- Millisecond (0-999).
    my      $microsec,                  #  in -- Microsecond (0-999).
    my      $nanosec,                   #  in -- Nanosecond (0-999).
    my      $picosec,                   #  in -- Picosecond (0-999).
    my      \@epoch16);                 #  out -- CDF_EPOCH16 value returned
```

epoch16, an array with two-double elements, contains the epoch time in picoseconds.

7.17 EPOCH16breakdown

EPOCH16breakdown decomposes a CDF_EPOCH16 value into the individual components.

```
CDF::EPOCH16breakdown(
    my      \@epoch16,              #  in   -- The CDF_EPOCH16 value.
    my      $year,                  #  out -- Year (AD, e.g., 1994).
    my      $month,                 #  out -- Month (1-12).
    my      $day,                   #  out -- Day (1-31).
    my      $hour,                  #  out -- Hour (0-23).
    my      $minute,                #  out -- Minute (0-59).
    my      $second,                #  out -- Second (0-59).
    my      $msec,                  #  out -- Millisecond (0-999).
    my      $microsec,              #  out -- Microsecond (0-999).
    my      $nanosec,               #  out -- Nanosecond (0-999).
    my      $picosec);              #  out -- Picosecond (0-999).
```

7.18 toEncodeEPOCH16

toEncodeEPOCH16 encodes a single CDF_EPOCH16 value or an array of CDF_EPOCH16 values into one of tge defined date/time character string(s), based on the passed style. The style is one of the following values:

-**0**: **dd-mmm-yyyy hh:mm:ss.ccc.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), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is nanosecond (0-999), and ppp is picosecond (0-999).

-**1**: **yyyymmdd.tttttttttttttt** where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), tttttttttttttt is a fraction of day.

-**2**: **yyyymmddhhmnss** 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), mn is the minute (0-59), ss is the second (0-59).

-**3**: **yyyy-mm-ddThh:mn:ss.ccc.uuu.nnn.pppZ** 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), mn is the minute (0-59), ss is the second (0-59), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is nanosecond (0-999), and ppp is picosecond (0-999).

-**4**: **yyyy-mm-ddThh:mn:ss.cccuuunnnppp** 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), mn is the minute (0-59), ss is the second (0-59), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is nanosecond (0-999), and ppp is picosecond (0-999).

Passing style 0 is the same as calling encodeEPOCH16. Passing style 1 is the same as calling encodeEPOCH16_1. Passing style 2 is the same as calling encodeEPOCH16_2. Passing style 3 is the same as calling encodeEPOCH16_3. Passing style 4 is the same as calling encodeEPOCH16_4.

```
CDF::toEncodeEPOCH16(
    my      \@epoch16;                  #  in  --  The CDF_EPOCH16  value.
    my      $style;                     #  in  --  The encoded style.
    my      $epString);                 #  out -- The date/time character string.
```

epString has a length of EPOCH16_STRING_LEN, EPOCH16_1_STRING_LEN, EPOCH16_2_STRING_LEN, EPOCH16_3_STRING_LEN, or EPOCH16_4_STRING_LEN. Passing style 0 is similar to calling encodeEPOCH16. Passing style 1 is similar to calling encodeEPOCH16_1. Passing style 2 is similar to calling encodeEPOCH16_2. Passing style 3 is similar to calling encodeEPOCH16_3. Passing style 4 is similar to calling encodeEPOCH16_4. Style 4 calling is the currently most used, as it is comforming to ISO 8601 format.

7.19 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.ccc: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), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```
CDF::encodeEPOCH16(
    my      \@epoch16;                  #  in  --  The CDF_EPOCH16  value.
    my      $epString);                 #  out -- The date/time character string.
```

epString has a length of EPOCH16_STRING_LEN.

7.20 encodeEPOCH16_1

encodeEPOCH16_1 encodes a CDF_EPOCH16 value into an alternate date/time character string. The format of the string is yyyymmdd.tttttttttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and tttttttttttttt is the fraction of the day (e.g., 500000000000000 is 12 o'clock noon).

```
CDF::encodeEPOCH16_1(
    my      \@epoch16;                  # in -- The CDF_EPOCH16 value.
    my      $epString);                 # out -- The date/time character string.
```

epString has a length of EPOCH16_1_STRING_LEN.

7.21 encodeEPOCH16_2

encodeEPOCH16_2 encodes a CDF_EPOCH16 value into an alternate date/time character string. The format of the string is yyyymoddhhmmss 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), and ss is the second (0-59).

```
CDF::encodeEPOCH16_2(
    my      \@epoch16;                  # in -- The CDF_EPOCH16 value.
    my      $epString);                 # out -- The date/time character string.
```

epString has a length of EPOCH16_2_STRING_LEN.

7.22 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:mn:ss.ccc:uuu:nnn:pppZ 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), mn is the minute (0-59), ss is the second (0-59), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```
CDF::encodeEPOCH16_3(
    my      \@epoch16;                  # in -- The CDF_EPOCH16 value.
    my      $epString);                 # out -- The alternate date/time character string.
```

epString has a length of EPOCH16_3_STRING_LEN.

7.23 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:mn:ss.cccuuunnnppp 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), mn is the minute (0-59), ss is the second (0-59), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```
CDF::encodeEPOCH16_4(
    my      \@epoch16;                  # in -- The CDF_EPOCH16 value.
```

```
    my      $epString);                 #  out  --  The alternate  date/time character string.
```

epString has a length of EPOCH16_4_STRING_LEN.

7.24 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.

```
CDF::encodeEPOCH16_x(
    mt      \@epoch16;              #  in  --  The CDF_EPOCH16  value.
    my      $format;                #  in  ---The format string.
    my      $encoded);              #  out  --  The date/time character string.
```

While the format string has a length up to EPOCHx_FORMAT_LEN, the encoded string has a length up to EPOCHx_STRING_MAX. 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. . .

Token	Meaning	Default
dom	Day of month (1-31)	<dom.0>
doy	Day of year (001-366)	<doy.03>
month	Month (`Jan',`Feb',...,`Dec')	<month>
mm	Month (1,2,...,12)	<mm.0>
year	Year (4-digit)	<year.04>
yr	Year (2-digit)	<yr.02>
hour	Hour (00-23)	<hour.02>
min	Minute (00-59)	<min.02>
sec	Second (00-59)	<sec.02>
msc	Millisecond (000-999)	<msc.3>
usc	Microsecond (000-999)	<usc.3>
nsc	Nanosecond (000-999)	<nsc.3>
psc	Picosecond (000-999)	<psc.3>
fos	Fraction of second.	<fos.12>
fod	Fraction of day.	<fod.8>

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 would be. . .

```
<dom.02>-<month>-<year>  <hour>:<min>:<sec>.<msc>.<usc>.<nsc>.<psc>.<fos>
```

7.25 toParseEPOCH16

toParseEPOCH16 parses a single or an array of encoded date/time string(s). The date/time string must conform to one of the encode styles (through the time encoding function for CDF_EPOCH16 data). This generalized function can be replaced all parsing functions for CDF_EPOCH16 type.

```
CDF::toParseEPOCH16(                    # out -- The status code returned.
    my      $epString;                  # in -- The standard date/time character string.
    my      \@epoch16);                 # out -- CDF_EPOCH16 value returned.
```

7.26 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.

```
CDF::parseEPOCH16(                      # out -- The status code returned.
    my      $epString,                  # in -- The date/time character string.
    my      \@epoch16);                 # out -- The CDF_EPOCH16 value returned
```

epString has a length of EPOCH16_STRING_LEN . epoch is an array of two elements.

7.27 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.

```
CDF::parseEPOCH16_1(                    # out -- The status code returned.
    my      $epString,                  # in -- The date/time character string.
    my      \@epoch16);                 # out -- The CDF_EPOCH16 value returned
```

epString has a length of EPOCH16_1_STRING_LEN . epoch is an array of two elements.

7.28 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.

```
CDF::parseEPOCH16_2(                    # out -- The status code returned.
    my      $epString,                  # in -- The date/time character string.
    my      \@epoch16);                 # out -- The CDF_EPOCH16 value returned
```

epString has a length of EPOCH16_2_STRING_LEN . epoch is an array of two elements

7.29 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.

```
CDF::parseEPOCH16_3(                    # out -- The status code returned.
    my      $epString,                  # in -- The date/time character string.
    my      \@epoch16);                 # out -- The CDF_EPOCH16 value returned
```

epString has a length of EPOCH16_3_STRING_LEN . epoch is an array of two elements

7.30 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.

```
CDF::parseEPOCH16_3(                    # out -- The status code returned.
    my      $epString,                  # in -- The ISO 8601 date/time character string.
    my      \@epoch16);                 # out -- The CDF_EPOCH16 value returned
```

epString has a length of EPOCH16_4_STRING_LEN . epoch is an array of two elements

7.31 EPOCHtoUnixTime

EPOCHtoUnixTime converts an epoch time(s) of CDF_EPOCH type into a Unix time(s). A CDF_EPOCH epoch, a double, is milliseconds from 0000-01-01T00:00:00.000 while Unix time, also a double, is seconds from 1970-01-01T00:00:00.000. The Unix time can have sub-second, with a time resolution of microseconds, in its fractional part.

```
CDF::EPOCHtoUnixTime (
    my  $epoch,                         # in -- CDF_EPOCH epoch time.
    my  $unixTime);                     # out -- Unix time.

CDF::EPOCHtoUnixTime (
    my  \@epochs,                       # in -- CDF_EPOCH epoch times.
    my  \@unixTimes);                   # out -- Unix times.
```

7.32 UnixTimetoEPOCH

UnixTimetoEPOCH converts a Unix time(s) into an epoch time(s) of CDF_EPOCH type. A Unix time, a double, is seconds from 1970-01-01T00:00:00.000 while a CDF_EPOCH epoch, also a double, is milliseconds from 0000-01-01T00:00:00.000. The Unix time can have sub-second, with a time resolution of microseconds, in its fractional part. Converting the Unix time to EPOCH will only keep the resolution to milliseconds.

```
CDF::UnixTimetoEPOCH (
    my $unixTime,                    # in  -- Unix time.
    my $epoch);                      # out  --  CDF_EPOCH epoch time.

CDF::UnixTimetoEPOCH (
    my \@unixTimes,                  # in  -- Unix times.
    my \@epochs);                    # out  --  CDF_EPOCH epoch times.
```

7.33 EPOCH16toUnixTime

EPOCH16toUnixTime converts an epoch time of CDF_EPOCH16 type into a Unix time. A CDF_EPOCH16 epoch, a two-double, is picoseconds from 0000-01-01T00:00:00.000.000.000.000 while Unix time, a double, is seconds from 1970-01-01T00:00:00.000. The Unix time can have sub-second, with a time resolution of microseconds, in its fractional part. **Note**: As CDF_EPOCH16 has much higher time resolution, sub-microseconds portion of its time might get lost during the conversion.

```
CDF::EPOCH16toUnixTime (
    my  \@epochs,                    # in  -- CDF_EPOCH16 epoch time.
    my  \$unixTimes);                # out  -- Unix time.
```

7.34 UnixTimetoEPOCH16

UnixTimetoEPOCH16 converts a Unix time into an epoch time of CDF_EPOCH16 type. A Unix time, a double, is seconds from 1970-01-01T00:00:00.000 while a CDF_EPOCH16 epoch, a two-double, is picoseconds from 0000-01-01T00:00:00.000.000.000.000. The Unix time can have sub-second, with a time resolution of microseconds, in its fractional part. Sub-microseconds will be filled with 0's when converting from Unix time to EPOCH16.

```
CDF::UnixTimetoEPOCH16 (
    my  $unixTime,                   # in  -- Unix time.
    my  \@epochs);                   # out  --  CDF_EPOCH16 epoch times.
```

8 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. The Concepts chapter in the CDF User's Guide describes CDF_TIME_TT2000 values. The date/time components for CDF_TIME_TT2000 are **UTC-based**, with leap seconds.

The CDF_TIME_TT2000 data type is used to store time values referenced from **J2000** (2000-01-01T12:00:00.000000000). For CDF, values in CDF_TIME_TT2000 are nanoseconds from J2000 with **leap seconds** included. TT2000 data can cover years between 1707 and 2292.

8.1 computeTT2000

computeTT2000 calculates a single CDF_TIME_TT2000 value, given the individual UTC-based date/time components, or array of date/time components. If an illegal component is detected, e.g., the year is outside of the valid range for TT2000 data, the value returned will be ILLEGAL_TT2000_VALUE.

```
$tt2000 = CDF::computeTT2000(       #  out -- CDF_TIME_TT2000 value returned.
    my $year,                       #  in -- Year (AD, e.g., 1994).
    my $month,                      #  in -- Month (1-12).
    my $day,                        #  in -- Day (1-31).
    my $hour,                       #  in -- Hour (0-23).
    my $minute,                     #  in -- Minute (0-59).
    my $second,                     #  in -- Second (0-59 or 0-60 if leap second).
    my $msec;                       #  in -- Millisecond (0-999).
    my $usec,                       #  in -- Microsecond (0-999).
    my $nsec);                      #  in -- Nanosecond (0-999).

@tt2000 = CDF::computeTT2000(       #  out -- An array of CDF_TIME_TT2000 values returned.
    my \@year,                      #  in -- Years (AD, e.g., 1994).
    my \@month,                     #  in -- Months (1-12).
    my \@day,                       #  in -- Days (1-31).
    my \@hour,                      #  in -- Hours (0-23).
    my \@minute,                    #  in -- Minutes (0-59).
    my \@second,                    #  in -- Seconds (0-59 or 0-60 if leap second).
    my \@msec;                      #  in -- Milliseconds (0-999).
    my \@usec,                      #  in -- Microseconds (0-999).
    my \@nsec);                     #  in -- Nanoseconds (0-999).
```

NOTE: There are two variations on how computeEPOCH may be used. If the month argument is 0 (zero), then 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 0 (zero), then the msec argument is assumed to be the millisecond of the day having a range of 0 through 86400000. Similar arrangements are for micro and nano-second. The day componment can be presented as day of the month or day of the yead (DOY). If DOY form is used, the month componment must have a value(s) of one (1).

8.2 TT2000breakdown

TT2000breakdown decomposes a single or an array of CDF_IME_TT2000 value(s) into the individual UTC-based Date/time components.

```
CDF::EPOCHbreakdown(
    my      $epoch,             #  in  --  The CDF_TIME_TT2000  value.
    my      $year,              #  out  --  Year  (AD,  e.g.,  1994).
    my      $month,             #  out  --  Month  (1-12).
    my      $day,               #  out  --  Day  (1-31).
    my      $hour,              #  out  --  Hour  (0-23).
    my      $minute,            #  out  --  Minute  (0-59).
    my      $second,            #  out  --  Second  (0-59 or 0-60 if leap second).
    my      $msec;              #  out  --  Millisecond  (0-999).
    my      $usec,              #  out  --  Microsecond  (0-999).
    my      $nsec);             #  out  --  Nanosecond  (0-999).

CDF::EPOCHbreakdown(
    my      \@epoch,            #  in  --  An array of the CDF_TIME_TT2000  values.
    my      \@year,             #  out  --  Years  (AD,  e.g.,  1994).
    my      \@month,            #  out  --  Months  (1-12).
    my      \@day,              #  out  --  Days  (1-31).
    my      \@hour,             #  out  --  Hours  (0-23).
    my      \@minute,           #  out  --  Minutes  (0-59).
    my      \@second,           #  out  --  Seconds  (0-59 or 0-60 if leap second).
    my      \@msec;             #  out  --  Milliseconds  (0-999).
    my      \@usec,             #  out  --  Microseconds  (0-999).
    my      \@nsec);            #  out  --  Nanoseconds  (0-999).
```

8.3 toEncodeTT2000

toEncodeTT2000 encodes a single or an array of CDF_TT2000 value(s) into one of the defined date/time character string(s), based on the passed style. The style is one of the following values:

-**0**: **dd-mmm-yyyy hh:mm:ss.ccc.uuu.nnn** 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|60), ccc is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is nanosecond (0-999).

-**1**: **yyyymmdd.tttttttttttttt** where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), tttttttttttttt is a fraction of day.

-**2**: **yyyymmddhhmnss** 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), mn is the minute (0-59), ss is the second (0-59|60).

-**3**: **yyyy-mm-ddThh:mn:ss.cccuuunnn** 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), mn is the minute (0-59), ss is the second (0-59|60), ccc is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is nanosecond (0-999).

-**4**: **yyyy-mm-ddThh:mn:ss.cccuuunnnZ** 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), mn is the minute (0-59), ss is the second (0-59|60), ccc is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is nanosecond (0-999).

```
CDF::toEncodeEPOCH(
    my      $epoch;                 #  in  --  The CDF_TIME_TT2000  value.
    my      $style;                 #  in  --  The encoded style.
    my      $epString);             #  out  --  The standard  date/time character string.

CDF::toEncodeEPOCH(
    my      \@epoch;                #  in  --  An array of CDF_TIME_TT2000  values.
    my      $style;                 #  in  --  The encoded style.
    my      \@epString);            #  out  --  The standard  date/time character strings.
```

Passing style 3 is similar to call encodeTT2000.

8.4 encodeTT2000

encodeTT2000 encodes a single or an array of CDF_TT2000 value(s) into the standard UTC-based date/time character string(s). The default format of the string is in **ISO 8601** format: **yyyy-mm-ddT hh:mn:ss.cccuuunnn** 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), mn is the minute (0-59), ss is the second (0-59 or 0-60 if leap second), ccc is the millisecond (0-999), uuu is the microsecond (0-999) and nnn is the nanosecond (0-999).

```
CDF::encodeEPOCH(
    my      $epoch;                 #  in  --  The CDF_TIME_TT2000  value.
    my      $epString);             #  out  --  The standard  date/time character string.

CDF::encodeEPOCH(
    my      \@epoch;                #  in  --  An array of CDF_TIME_TT2000  values.
    my      \@epString);            #  out  --  The standard  date/time character strings.
```

This module accepts an extra, optional argument field of 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 fomat 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.cccuuunnn**, 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), ss is the second (0-59 or 0-60 if leap second), ccc is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999).

For a format of value **1**, the encoded UTC string is **YYYYMMDD.tttttttt**, where YYYY is the year, MM is the month (1-12) DD is the day of the month (1-31), and tttttttt is sub-day.(0-999999999).

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).

For a format of value **3**, the encoded UTC string is **YYYY-MM-DDThh:mm:ss.cccuuunnn**, 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), ss is the second (0-59 or 0-60 if leap second), ccc is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999).

For a format of value **4**, the encoded UTC string is **YYYY-MM-DDThh:mm:ss.cccuuunnnZ**, 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), ss is the second (0-59 or 0-60 if leap second), ccc is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999).

8.5 toParseTT2000

toParseTT2000 parses a single or an array of encoded date/time string(s). The date/time string must conform to one of the encode styles (through the time encoding function for CDF_TIME_TT2000 value).

```
$tt2000 = CDF::toParseTT2000(                # out -- CDF_TIME_TT2000 value returned.
    my       $epString);                     # in -- The standard date/time character string.

@tt2000 = CDF::toParseTT2000(                # out -- An array of CDF_TIME_TT2000 values returned.
    my       \@epString);                    # in -- The standard date/time character strings.
```

8.6 parseTT2000

parseTT2000 parses a signle or an array of standard UTC-based date/time character strings and returns a CDF_TIME_TT2000 value(s). The format of the string is that produced by the encodeTT2000 function described in Section 7.3. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

```
$tt2000 = CDF::parseEPOCH(                   # out -- CDF_TIME_TT2000 value returned.
    my       $epString);                     # in -- The standard date/time character string.

@tt2000 = CDF::parseEPOCH(                   # out -- An array of CDF_TIME_TT2000 values returned.
    my       \@epString);                    # in -- The standard date/time character strings.
```

8.7 TT2000toUnixTime

TT2000toUnixTime converts an epoch time(s) of CDF_TIME_TT2000 (TT2000) type into a Unix time(s). A CDF_TIME_TT2000 epoch, a 8-byte integer, is nanoseconds from J2000 with leap seconds while Unix time, a double, is seconds from 1970-01-01T00:00:00.000. **Note**: As CDF_TIME_TT2000 has much higher time resolution, sub-milcroseconds portion of its time might get lost during the conversion. Also, TT2000's leap seconds will get lost after the conversion.

```
CDF::TT2000toUnixTime (
    my  $epoch,                              # in -- CDF_TIME_TT2000 epoch time.
    my  $unixTime);                          # out -- Unix time.

CDF::TT2000toUnixTime (
    my  \@epochs,                            # in -- CDF_TIME_TT2000 epoch times.
    my  \@unixTimes);                        # out -- Unix times.
```

8.8 UnixTimetoTT2000

UnixTimetoTT2000 converts a Unix time(s) into an epoch time(s) of CDF_TIME_TT2000 (TT2000) type. A Unix time, a double, is seconds from 1970-01-01T00:00:00.000 while a CDF_TIME_TT2000 epoch, a 8-byte integer, is

nanoseconds from J2000 with lepa seconds. The Unix time can have sub-second, with a time resolution of microseconds, in its fractional part. Sub-microseconds will be filled with 0's when converting from Unix time to TT2000.

```
CDF::UnixTimetoTT2000 (
    my $unixTime,                    #  in  -- Unix time.
    my $epoch);                      # out  --  CDF_TIME_TT2000 epoch time.

CDF::UnixTimetoTT2000 (
    my \@unixTimes,                  #  in  -- Unix times.
    my \@epochs);                    # out  --  CDF_TIME_TT2000 epoch times.
```

8.9 leapsecondsinfo

leapsecondinfo shows how the leap seconds table is accessed and when the last leap second was added. The table can be accessed externally or internally by the CDF library. Refer to User's Guide for leap seconds.

```
CDF::leapsecondinfo();
```

Optionally, a n integer value of 0 (zero) or non-zero can be passed to the module. If a non-zero value is passed in, the contents of the leap seconds table is dumped. No value or 0 is passed in, the table is not shown.

Appendix A

A.1 Introduction

A status code is returned from most CDF functions. The cdf.h include file, distributed with the Perl-CDF package, contains 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 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]

BAD_ATTR_NAME	Illegal attribute name specified. Attribute names must contain at least one character, and each character must be printable. [Error]
BAD_ATTR_NUM	Illegal attribute number specified. Attribute numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]
BAD_BLOCKING_FACTOR[27]	An illegal blocking factor was specified. Blocking factors must be at least zero (0). [Error]
BAD_CACHESIZE	An illegal number of cache buffers was specified. The value must be at least zero (0). [Error]
BAD_CDF_EXTENSION	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]
BAD_CDF_ID	CDF identifier is unknown or invalid. The CDF identifier specified is not for a currently open CDF. [Error]
BAD_CDF_NAME	Illegal CDF name specified. CDF names must contain at least one character, and each character must be printable. Trailing blanks are allowed but will be ignored. [Error]
BAD_CDFSTATUS	Unknown CDF status code received. The status code specified is not used by the CDF library. [Error]
BAD_CHECKSUM	An illegal checksum mode received. It is invlid or currently not supported. [Error]
BAD_COMPRESSION_PARM	An illegal compression parameter was specified. [Error]
BAD_DATA_TYPE	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]
BAD_DECODING	An unknown decoding was specified. The CDF decodings are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_DIM_COUNT	Illegal dimension count specified. A dimension count must be at least one (1) and not greater than the size of the dimension. [Error]
BAD_DIM_INDEX	One or more dimension index is out of range. A valid value must be specified regardless of the dimension variance. Note also that the combination of dimension index, count, and interval must not specify an element beyond the end of the dimension. [Error]
BAD_DIM_INTERVAL	Illegal dimension interval specified. Dimension intervals must be at least one (1). [Error]
BAD_DIM_SIZE	Illegal dimension size specified. A dimension size must be at least one (1). [Error]

[27] The status code BAD_BLOCKING_FACTOR was previously named BAD_EXTEND_RECS.

BAD_ENCODING	Unknown data encoding specified. The CDF encodings are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_ENTRY_NUM	Illegal attribute entry number specified. Entry numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. [Error]
BAD_FNC_OR_ITEM	The specified function or item is illegal. Check that the proper number of arguments are specified for each operation being performed. Also make sure that NULL_ is specified as the last operation. [Error]
BAD_FORMAT	Unknown format specified. The CDF formats are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_INITIAL_RECS	An illegal number of records to initially write has been specified. The number of initial records must be at least one (1). [Error]
BAD_MAJORITY	Unknown variable majority specified. The CDF variable majorities are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_MALLOC	Unable to allocate dynamic memory - system limit reached. Contact CDF User Support if this error occurs. [Error]
BAD_NEGtoPOSfp0_MODE	An illegal -0.0 to 0.0 mode was specified. The -0.0 to 0.0 modes are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_NUM_DIMS	The number of dimensions specified is out of the allowed range. Zero (0) through CDF_MAX_DIMS dimensions are allowed. If more are needed, contact CDF User Support. [Error]
BAD_NUM_ELEMS	The number of elements of the data type is illegal. The number of elements must be at least one (1). For variables with a non-character data type, the number of elements must always be one (1). [Error]
BAD_NUM_VARS	Illegal number of variables in a record access operation. [Error]
BAD_READONLY_MODE	Illegal read-only mode specified. The CDF read-only modes are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_REC_COUNT	Illegal record count specified. A record count must be at least one (1). [Error]
BAD_REC_INTERVAL	Illegal record interval specified. A record interval must be at least one (1). [Error]
BAD_REC_NUM	Record number is out of range. Record numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. Note that a valid value must be specified regardless of the record variance. [Error]

BAD_SCOPE	Unknown attribute scope specified. The attribute scopes are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
BAD_SCRATCH_DIR	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]
BAD_SPARSEARRAYS_PARM	An illegal sparse arrays parameter was specified. [Error]
BAD_VAR_NAME	Illegal variable name specified. Variable names must contain at least one character and each character must be printable. [Error]
BAD_VAR_NUM	Illegal variable number specified. Variable numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]
BAD_zMODE	Illegal zMode specified. The CDF zModes are defined in cdf.h for C applications and in cdf.inc for Fortran applications. [Error]
CANNOT_ALLOCATE_RECORDS	Records cannot be allocated for the given type of variable (e.g., a compressed variable). [Error]
CANNOT_CHANGE	Because of dependencies on the value, it cannot be changed. Some possible causes of this error follow: 1. Changing a CDF's data encoding after a variable value (including a pad value) or an attribute entry has been written. 2. Changing a CDF's format after a variable has been created 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]
CDF_READ_ERROR	Failed to read the CDF file - error from file system. Check that the dotCDF file is not corrupted. [Error]
CDF_WRITE_ERROR	Failed to write the CDF file - error from file system. Check that the dotCDF file is not corrupted. [Error]
CHECKSUM_ERROR	Data integrity verification through the checksum failed. [Error]

CHECKSUM_NOT_ALLOWED	The checksum is not allowed for old versioned files. [Error]
COMPRESSION_ERROR	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. [Error]
CORRUPTED_V2_CDF	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. [Error]
DECOMPRESSION_ERROR	An error occurred while decompressing a CDF or block of variable records. The most likely cause is a corrupted dotCDF file. [Error]
DID_NOT_COMPRESS	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. [Informational]
EMPTY_COMPRESSED_CDF	The compressed CDF being opened is empty. This will result if a program, which was creating/modifying, the CDF abnormally terminated. [Error]
END_OF_VAR	The sequential access current value is at the end of the variable. Reading beyond the end of the last physical value for a variable is not allowed (when performing sequential access). [Error]
FORCED_PARAMETER	A specified parameter was forced to an acceptable value (rather than an error being returned). [Warning]
IBM_PC_OVERFLOW	An operation involving a buffer greater than 64k bytes in size has been specified for PCs running 16-bit DOS/Windows 3.*. [Error]
ILLEGAL_EPOCH_VALUE	Illegal component is detected in computing an epoch value or an illegal epoch value is provided in decomposing an epoch value. [Error]
ILLEGAL_FOR_SCOPE	The operation is illegal for the attribute's scope. For example, only gEntries may be written for gAttributes - not rEntries or zEntries. [Error]
ILLEGAL_IN_zMODE	The attempted operation is illegal while in zMode. Most operations involving rVariables or rEntries will be illegal. [Error]
ILLEGAL_ON_V1_CDF	The specified operation (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]
NO_VARS_IN_CDF	This CDF contains no rVariables. The operation performed is not applicable to a CDF with no rVariables. [Informational]
NO_WRITE_ACCESS	Write access is not allowed on the CDF file(s). Make sure that the CDF file(s) have the proper file system privileges and ownership. [Error]
NOT_A_CDF	Named CDF is corrupted or not actually a CDF. Contact CDF User Support if you are sure that the specified file is a CDF that should be readable by the CDF distribution being used. [Error]
NOT_A_CDF_OR_NOT_SUPPORTED	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]
PRECEEDING_RECORDS_ALLOCATED	Because of the type of variable, records preceding the range of records being allocated were automatically allocated as well. [Informational]
READ_ONLY_DISTRIBUTION	Your CDF distribution has been built to allow only read access to CDFs. Check with your system manager if you require write access. [Error]
READ_ONLY_MODE	The CDF is in read-only mode - modifications are not allowed. [Error]
SCRATCH_CREATE_ERROR	Cannot create a scratch file - error from file system. If a scratch directory has been specified, ensure that it is writeable. [Error]
SCRATCH_DELETE_ERROR	Cannot delete a scratch file - error from file system. [Error]
SCRATCH_READ_ERROR	Cannot read from a scratch file - error from file system. [Error]
SCRATCH_WRITE_ERROR	Cannot write to a scratch file - error from file system. [Error]
SINGLE_FILE_FORMAT	The specified operation is not applicable to CDFs with the single-file format. For example, it does not make sense to close a variable in a single-file CDF. [Informational]
SOME_ALREADY_ALLOCATED	Some of the records being allocated were already allocated. [Informational]
TOO_MANY_PARMS	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]
TOO_MANY_VARS	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]
UNKNOWN_COMPRESSION	An unknown type of compression was specified or encountered. [Error]
UNKNOWN_SPARSENESS	An unknown type of sparseness was specified or encountered. [Error]
UNSUPPORTED_OPERATION	The attempted operation is not supported at this time. [Error]
VAR_ALREADY_CLOSED	The specified variable is already closed. [Informational]
VAR_CLOSE_ERROR	Error detected while trying to close variable file. Check that sufficient disk space exists for the variable file and that it has not been corrupted. [Error]
VAR_CREATE_ERROR	An error occurred while creating a variable file in a multi-file CDF. Check that a file quota has not been reached. [Error]

VAR_DELETE_ERROR	An error occurred while deleting a variable file in a multi-file CDF. Check that sufficient privilege exist to delete the CDF files. [Error]
VAR_EXISTS	Named variable already exists - cannot create or rename. Each variable in a CDF must have a unique name (rVariables and zVariables can not share names). Note that trailing blanks are ignored by the CDF library when comparing variable names. [Error]
VAR_NAME_TRUNC	Variable name truncated to CDF_VAR_NAME_LEN256 characters. The variable was created but with a truncated name. [Warning]
VAR_OPEN_ERROR	An error occurred while opening variable file. Check that sufficient privilege exists to open the variable file. Also make sure that the associated variable file exists. [Error]
VAR_READ_ERROR	Failed to read variable as requested - error from file system. Check that the associated file is not corrupted. [Error]
VAR_WRITE_ERROR	Failed to write variable as requested - error from file system. Check that the associated file is not corrupted. [Error]
VIRTUAL_RECORD_DATA	One or more of the records are virtual (never actually written to the CDF). Virtual records do not physically exist in the CDF file(s) but are part of the conceptual view of the data provided by the CDF library. Virtual records are described in the Concepts chapter in the CDF User's Guide. [Informational]

Appendix B

B.1 Standard Interface

```
$status = CDF::CDFattrCreate  ($id, $attrName, $attrScope, \$attrNum)
my      $id;                                                    # in
my      $attrName;                                              # in
my      $attrScope;                                             # in
my      \$attrNum;                                              # out

$status = CDF::CDFattrEntryInquire  ($id, $attrNum, $entryNum, \$dataType, \$numElements)
my      $id;                                                    # in
my      $attrNum;                                               # in
my      $entryNum;                                              # in
my      \$dataType;                                             # out
my      \$numElements;                                          # out

$status = CDF::CDFattrGet  ($id, $attrNum, $entryNum,  \$value)
my      $id;                                                    # in
my      $attrNum;                                               # in
my      $entryNum;                                              # in
my      \$value;                                                # out

$status = CDF::CDFattrInquire  ($id, $attrNum, \$attrName, \$attrScope,  \$maxEntry)
my      $id;                                                    # in
my      $attrNum;                                               # in
my      \$attrName;                                             # out
my      \$attrScope;                                            # out
my      \$maxEntry;                                             # out

$varNum = CDF::CDFattrNum  ($id, $attrName)
my      $id;                                                    # in
my      $attrName;                                              # in

$status = CDF::CDFattrPut  ($id, $attrNum, $entryNum, $dataType,  $numElements, \$value)
my      $id;                                                    # in
my      $attrNum;                                               # in
my      $entryNum;                                              # in
my      $dataType;                                              # in
my      $numElements;                                           # in
my      \$value;                                                # in

$status = CDF::CDFattrRename ($id, $attrNum, $attrName)
my      $id;                                                    # in
my      $attrNum;                                               # in
```

```
my      $attrName;                                                      # in

$status = CDF::CDFclose  ($id)
my      $id;                                                            # in

$status = CDF::CDFcreate  ($CDFname,  $numDims,  \@dimSizes, $encoding,  $majority,  \$id)
my      $CDFname;                                                       # in
my      $numDims;                                                       # in
my      \@dimSizes;                                                     # in
my      $encoding;                                                      # in
my      $majority;                                                      # in
my      \$id;                                                           # out

$status = CDF::CDFdelete  ($id)
my      $id;                                                            # in

$status = CDF::CDFdoc  ($id,  \$version,  \$release,  \$text)
my      $id;                                                            # in
my      \$version;                                                      # out
my      \$release;                                                      # out
my      \$text;                                                         # out

$status = CDF:: CDFerror ($status,  \$message)
my      $status;                                                        # in
my      \$message;                                                      # out

$status = CDF::CDFgetChecksum  ($id,  \$checksum)
my      $id;                                                            # in
my      \$checksum;                                                     # out

$flag = CDF::CDFgetFileBackward ()
my      $flag                                                           # out

$flag = CDF::CDFgetValidate ()
my      $flag                                                           # out

$status = CDF::CDFinquire  ($id,  \$numDims,  \@dimSizes, \$encoding,  \$majority,  \$maxRec,
                            \$numVars,  \$numAttrs)
my      $id;                                                            # in
my      \$numDims;                                                      # out
my      \@dimSizes;                                                     # out
my      \$encoding;                                                     # out
my      \$majority;                                                     # out
my      \$maxRec;                                                       # out
my      \$numVars;                                                      # out
my      \$numAttrs;                                                     # out

$status = CDF:: CDFopen  ($CDFname,  $id)
my      $CDFname;                                                       # in
my      \$id;                                                           # out

$status = CDF::CDFsetChecksum  ($id, $checksum)
my      $id;                                                            # in
my      $checksum;                                                      # in

CDF::CDFsetFileBackward ($cdf27BackwardCompatibleFlag)
```

```
my      $cdf27BackwardCompatibleFlag;                                   # in

$ CDF::CDFsetValidate ($validationFlag)
my      $validationFlag;                                                # in

status = CDF::CDFvarClose  ($id,  $varNum)
my      $id;                                                            # in
my      $varNum;                                                        # in

$status = CDF::CDFvarCreate  ($id,  $varName, $dataType,  $numElements, $recVariances,
                              \@dimVariances,  \$varNum)
my      $id;                                                            # in
my      $varName;                                                       # in
my      $dataType;                                                      # in
my      $numElements;                                                   # in
my      $recVariance;                                                   # in
my      \@dimVariances;                                                 # in
my      \$varNum;                                                       # out

$status = CDF::CDFvarGet  ($id,  $varNum,  $recNum, \@indices,  \$value)
my      $id;                                                            # in
my      $varNum;                                                        # in
my      $recNum;                                                        # in
my      \@indices;                                                      # in
my      \$value;                                                        # out

$status = CDF::CDFvHpGet  ($id,  $varNum,  $recStart,  $recCount,  $recInterval,
                           \@indices,  \@counts, \@intervals,  \@buffer)
my      $id;                                                            # in
my      $varNum;                                                        # in
my      $recStart;                                                      # in
my      $recCount;                                                      # in
my      $recInterval;                                                   # in
my      \@indices;                                                      # in
my      \@counts;                                                       # in
my      \@intervals;                                                    # in
my      \@buffer;                                                       # out

$status = CDF::CDFvHpPut  ($id,  $varNum,  $recStart,  $recCount,  $recInterval,
                           \@indices,  \@counts, \@intervals,  \@buffer)
my      $id;                                                            # in
my      $varNum;                                                        # in
my      $recStart;                                                      # in
my      $recCount;                                                      # in
my      $recInterval;                                                   # in
my      \@indices;                                                      # in
my      \@counts;                                                       # in
my      \@intervals;                                                    # in
my      \@buffer;                                                       # in

$status = CDF::CDFvarInquire  ($id,  $varNum,  \$varName, \$dataType,  \$numElements,
                               \$recVariance,  \@dimVariances)
my      $id;                                                            # in
my      $varNum;                                                        # in
my      \$varName;                                                      # out
my      \$dataType;                                                     # out
```

```
my      \$numElements;                                          # out
my      \$recVariance;                                          # out
my      \@dimVariances;                                         # out

$varNum = CDF::CDFvarNum  ($id,  $varName)                      # out
my      $id;                                                    # in
my      $varName;                                               # in

$status = CDF::CDFvarPut  ($id,  $varNum,  $recNum, \@indices,  \$value)
my      $id;                                                    # in
my      $varNum;                                                # in
my      $recNum;                                                # in
my      \@indices;                                              # in
my      \$value;                                                # in

$status = CDF::CDFvarRename ($id,  $varNum,  $varName)
my      $id;                                                    # in
my      $varNum;                                                # in
my      $varName;                                               # in
```

B.2 Internal Interface

```
$status = CDF::CDFlib  (op,  ...)
        op;                                                     # in
        CLOSE_
                CDF_
                rVAR_
                zVAR_

        CONFIRM_
                ATTR_                   \$attrNum               # out
                ATTR_EXISTENCE_         $attrName               # in
                CDF_                    \$id                    # out
                CDF_ACCESS_
                CDF_CACHESIZE_          \$numBuffers            # out
                CDF_DECODING_           \$decoding              # out
                CDF_NAME_               \$CDFname               # out
                CDF_NEGtoPOSfp0_MODE_   \$mode                  # out
                CDF_READONLY_MODE_      \$mode                  # out
                CDF_STATUS_             \$status                # out
                CDF_zMODE_              \$mode                  # out
                COMPRESS_CACHESIZE_     \$numBuffers            # out
                CURgENTRY_EXISTENCE_
                CURrENTRY_EXISTENCE_
                CURzENTRY_EXISTENCE_
                gENTRY_                 \$entryNum              # out
                gENTRY_EXISTENCE_       $entryNum               # in
                rENTRY_                 \$entryNum              # out
                rENTRY_EXISTENCE_       $entryNum               # in
                rVAR_                   \$varNum                # out
                rVAR_CACHESIZE_         \$numBuffers            # out
                rVAR_EXISTENCE_         $varName                # in
                rVAR_PADVALUE_
                rVAR_RESERVEPERCENT_    \$percent               # out
                rVAR_SEQPOS_            \$recNum                # out
                                        \@indices               # out
                rVARs_DIMCOUNTS_        \@counts                # out
                rVARs_DIMINDICES_       \@indices               # out
                rVARs_DIMINTERVALS_     \@intervals             # out
                rVARs_RECCOUNT_         \$recCount              # out
                rVARs_RECINTERVAL_      \$recInterval           # out
                rVARs_RECNUMBER_        \$recNum                # out
                STAGE_CACHESIZE_        \$numBuffers            # out
                zENTRY_                 \$entryNum              # out
                zENTRY_EXISTENCE_       $entryNum               # in
                zVAR_                   \$varNum                # out
                zVAR_CACHESIZE_         \$numBuffers            # out
                zVAR_DIMCOUNTS_         \@counts                # out
                zVAR_DIMINDICES_        \@indices               # out
                zVAR_DIMINTERVALS_      \@intervals             # out
                zVAR_EXISTENCE_         $varName                # in
                zVAR_PADVALUE_
```

```
        zVAR_RECCOUNT_              \$recCount                  #  out
        zVAR_RECINTERVAL_           \$recInterval               #  out
        zVAR_RECNUMBER_             \$recNum                    #  out
        zVAR_RESERVEPERCENT_        \$percent                   #  out
        zVAR_SEQPOS_                \$recNum                    #  out
                                    \@indices                   #  out

CREATE_
        ATTR_                       $attrName                   #  in
                                    $scope                      #  in
                                    \$attrNum                   #  out

        CDF_                        $CDFname                    #  in
                                    $numDims                    #  in
                                    \@dimSizes                  #  in
                                    \$id                        #  out

        rVAR_                       $varName                    #  in
                                    $dataType                   #  in
                                    $numElements                #  in
                                    $recVary                    #  in
                                    \@dimVarys                  #  in
                                    \$varNum                    #  out

        zVAR_                       $varName                    #  in
                                    $dataType                   #  in
                                    $numElements                #  in
                                    $numDims                    #  in
                                    \@dimSizes                  #  in
                                    $recVary                    #  in
                                    \@dimVarys                  #  in
                                    \$varNum                    #  out

DELETE_
        ATTR_
        CDF_
        gENTRY_
        rENTRY_
        rVAR_
        rVAR_RECORDS_               $firstRecord                #  in
                                    $lastRecord                 #  in
        rVAR_RECORDS_RENUMBER_      $firstRecord                #  in
                                    $lastRecord                 #  in
        zENTRY_
        zVAR_
        zVAR_RECORDS_               $firstRecord                #  in
                                    $lastRecord                 #  in
        zVAR_RECORDS_RENUMBER_      $firstRecord                #  in
                                    $lastRecord                 #  in

GET_
        ATTR_MAXgENTRY_             \$maxEntry                  #  out
        ATTR_MAXrENTRY_             \$maxEntry                  #  out
        ATTR_MAXzENTRY_             \$maxEntry                  #  out
        ATTR_NAME_                  \$attrName                  #  out
        ATTR_NUMBER_                $attrName                   #  in
```

	\$attrNum	#	out
ATTR_NUMgENTRIES_	\$numEntries	#	out
ATTR_NUMrENTRIES_	\$numEntries	#	out
ATTR_NUMzENTRIES_	\$numEntries	#	out
ATTR_SCOPE_	\$scope	#	out
CDF_CHECKSUM_	\$checksum	#	out
CDF_COMPRESSION_	\$cType	#	out
	\@cParms	#	out
	\$cPct	#	out
CDF_COPYRIGHT_	\$Copyright	#	out
CDF_ENCODING_	\$encoding	#	out
CDF_FORMAT_	\$format	#	out
CDF_INCREMENT_	\$increment	#	out
CDF_INFO_	$name	#	in
	\$cType	#	out
	\@cParms	#	out
	\$cSize	#	out
	\$uSize	#	out
CDF_LEAPSECONDLASTUPDATED_	\$lastUpdated	#	out
CDF_MAJORITY_	\$majority	#	out
CDF_NUMATTRS_	\$numAttrs	#	out
CDF_NUMgATTRS_	\$numAttrs	#	out
CDF_NUMrVARS_	\$numVars	#	out
CDF_NUMvATTRS_	\$numAttrs	#	out
CDF_NUMzVARS_	\$numVars	#	out
CDF_RELEASE_	\$release	#	out
CDF_VERSION_	\$version	#	out
DATATYPE_SIZE_	$dataType	#	in
	\$numBytes	#	out
gENTRY_DATA_	\$value	#	out
gENTRY_DATATYPE_	\$dataType	#	out
gENTRY_NUMELEMS_	\$numElements	#	out
LIB_COPYRIGHT_	\$Copyright	#	out
LIB_INCREMENT_	\$increment	#	out
LIB_RELEASE_	\$release	#	out
LIB_subINCREMENT_	\$subincrement	#	out
LIB_VERSION_	\$version	#	out
rENTRY_DATA_	\$value	#	out
rENTRY_DATATYPE_	\$dataType	#	out
rENTRY_NUMELEMS_	\$numElements	#	out
rENTRY_NUMSTRINGS_	\$numStrings	#	out
rENTRY_STRINGDATA_	\$@strings	#	out
rVAR_ALLOCATEDFROM_	$startRecord	#	in
	\$nextRecord	#	out
rVAR_ALLOCATEDTO_	$startRecord	#	in
	\$lastRecord	#	out
rVAR_BLOCKINGFACTOR_	\$blockingFactor	#	out
rVAR_COMPRESSION_	\$cType	#	out
	\@cParms	#	out
	\$cPct	#	out
rVAR_DATA_	\$value	#	out
rVAR_DATATYPE_	\$dataType	#	out
rVAR_DIMVARYS_	\@dimVarys	#	out
rVAR_HYPERDATA_	\@buffer	#	out
rVAR_MAXallocREC_	\$maxRec	#	out
rVAR_MAXREC_	\$maxRec	#	out

rVAR_NAME_	\$varName	#	out
rVAR_nINDEXENTRIES_	\$numEntries	#	out
rVAR_nINDEXLEVELS_	\$numLevels	#	out
rVAR_nINDEXRECORDS_	\$numRecords	#	out
rVAR_NUMallocRECS_	\$numRecords	#	out
rVAR_NUMBER_	$varName	#	in
	\$varNum	#	out
rVAR_NUMELEMS_	\$numElements	#	out
rVAR_NUMRECS_	\$numRecords	#	out
rVAR_PADVALUE_	\$value	#	out
rVAR_RECVARY_	\$recVary	#	out
rVAR_SEQDATA_	\$value	#	out
rVAR_SPARSEARRAYS_	\$sArraysType	#	out
	\@sArraysParms	#	out
	\$sArraysPct	#	out
rVAR_SPARSERECORDS_	\$sRecordsType	#	out
rVARs_DIMSIZES_	\@dimSizes	#	out
rVARs_MAXREC_	\$maxRec	#	out
rVARs_NUMDIMS_	\$numDims	#	out
rVARs_RECDATA_	$numVars	#	in
	\@varNums	#	in
	\@buffer	#	out
STATUS_TEXT_	\$text	#	out
zENTRY_DATA_	\$value	#	out
zENTRY_DATATYPE_	\$dataType	#	out
zENTRY_NUMELEMS_	\$numElements	#	out
zENTRY_NUMSTRINGS_	\$numStrings	#	out
zENTRY_STRINGDATA_	\$@strings	#	out
zVAR_ALLOCATEDFROM_	$startRecord	#	in
	\$nextRecord	#	out
zVAR_ALLOCATEDTO_	$startRecord	#	in
	\$lastRecord	#	out
zVAR_BLOCKINGFACTOR_	\$blockingFactor	#	out
zVAR_COMPRESSION_	\$cType	#	out
	\@cParms	#	out
	\$cPct	#	out
zVAR_DATA_	\$value	#	out
zVAR_DATATYPE_	\$dataType	#	out
zVAR_DIMSIZES_	\@dimSizes	#	out
zVAR_DIMVARYS_	\@dimVarys	#	out
zVAR_HYPERDATA_	\@buffer	#	out
zVAR_MAXallocREC_	\$maxRec	#	out
zVAR_MAXREC_	\$maxRec	#	out
zVAR_NAME_	\$varName	#	out
zVAR_nINDEXENTRIES_	\$numEntries	#	out
zVAR_nINDEXLEVELS_	\$numLevels	#	out
zVAR_nINDEXRECORDS_	\$numRecords	#	out
zVAR_NUMallocRECS_	\$numRecords	#	out
zVAR_NUMBER_	$varName	#	in
	\$varNum	#	out
zVAR_NUMDIMS_	\$numDims	#	out
zVAR_NUMELEMS_	\$numElements	#	out
zVAR_NUMRECS_	\$numRecords	#	out
zVAR_PADVALUE_	\$value	#	out
zVAR_RECVARY_	\$recVary	#	out
zVAR_SEQDATA_	\$value	#	out

```
        zVAR_SPARSEARRAYS_      \$sArraysType           #  out
                                \@sArraysParms          #  out
                                \$sArraysPct            #  out
        zVAR_SPARSERECORDS_     \$sRecordsType          #  out
        zVARs_MAXREC_           \$maxRec                #  out
        zVARs_RECDATA_          $numVars                #  in
                                \@varNums               #  in
                                \@buffer                #  out
NULL_

OPEN_
        CDF_                    $CDFname                #  in
                                \$id                    #  out
PUT__
        ATTR_NAME_              $attrName               #  in
        ATTR_SCOPE_             $scope                  #  in
        CDF_CHECKSUM_           $checksum               #  in
        CDF_COMPRESSION_        $cType                  #  in
                                \@cParms                #  in
        CDF_ENCODING_           $encoding               #  in
        CDF_FORMAT_             $format                 #  in
        CDF_LEAPSECONDLASTUPDATED_  $lastUpdated        #  in
        CDF_MAJORITY_           $majority               #  in
        gENTRY_DATA_            $dataType               #  in
                                $numElements            #  in
                                \$value                 #  in
        gENTRY_DATASPEC_        $dataType               #  in
                                $numElements            #  in
        rENTRY_DATA_            $dataType               #  in
                                $numElements            #  in
                                \$value                 #  in
        rENTRY_DATASPEC_        $dataType               #  in
                                $numElements            #  in
        rENTRY_STRINGDATA_      \@strings               #  in
        rVAR_ALLOCATEBLOCK_     $firstRecord            #  in
                                $lastRecord             #  in
        rVAR_ALLOCATERECS_      $numRecords             #  in
        rVAR_BLOCKINGFACTOR_    $blockingFactor         #  in
        rVAR_COMPRESSION_       $cType                  #  in
                                \@cParms                #  in
        rVAR_DATA_              \$value                 #  in
        rVAR_DATASPEC_          $dataType               #  in
                                $numElements            #  in
        rVAR_DIMVARYS_          \@dimVarys              #  in
        rVAR_HYPERDATA_         \@buffer                #  in
        rVAR_INITIALRECS_       $nRecords               #  in
        rVAR_NAME_              $varName                #  in
        rVAR_PADVALUE_          \$value                 #  in
        rVAR_RECVARY_           $recVary                #  in
        rVAR_SEQDATA_           \$value                 #  in
        rVAR_SPARSEARRAYS_      $sArraysType            #  in
                                \@sArraysParms          #  in
        rVAR_SPARSERECORDS_     $sRecordsType           #  in
        rVARs_RECDATA_          $numVars                #  in
                                \@varNums               #  in
                                \$buffer                #  in
```

```
        zENTRY_DATA_            long dataType           # in
                                $numElements            # in
                                \$value                 # in
        zENTRY_DATASPEC_        $dataType               # in
                                $numElements            # in
        zENTRY_STRINGDATA_      \@strings               # in
        zVAR_ALLOCATEBLOCK_     $firstRecord            # in
                                $lastRecord             # in
        zVAR_ALLOCATERECS_      $numRecords             # in
        zVAR_BLOCKINGFACTOR_    $blockingFactor         # in
        zVAR_COMPRESSION_       $cType                  # in
                                \@$cParms               # in
        zVAR_DATA_              \$value                 # in
        zVAR_DATASPEC_          $dataType               # in
                                $numElements            # in
        zVAR_DIMVARYS_          \@dimVarys              # in
        zVAR_INITIALRECS_       $nRecords               # in
        zVAR_HYPERDATA_         \@buffer                # in
        zVAR_NAME_              $varName                # in
        zVAR_PADVALUE_          \$value                 # in
        zVAR_RECVARY_           $recVary                # in
        zVAR_SEQDATA_           \$value                 # in
        zVAR_SPARSEARRAYS_      $sArraysType            # in
                                \@sArraysParms          # in
        zVAR_SPARSERECORDS_     $sRecordsType           # in
        zVARs_RECDATA_          $numVars                # in
                                \@varNums               # in
                                \@buffer                # in
SELECT_
        ATTR_                   $attrNum                # in
        ATTR_NAME_              $attrName               # in
        CDF_                    $id                     # in
        CDF_CACHESIZE_          $numBuffers             # in
        CDF_DECODING_           $decoding               # in
        CDF_NEGtoPOSfp0_MODE_   $mode                   # in
        CDF_READONLY_MODE_      $mode                   # in
        CDF_SCRATCHDIR_         $dirPath                # in
        CDF_STATUS_             $status                 # in
        CDF_zMODE_              $mode                   # in
        COMPRESS_CACHESIZE_     $numBuffers             # in
        gENTRY_                 $entryNum               # in
        rENTRY_                 $entryNum               # in
        rENTRY_NAME_            $varName                # in
        rVAR_                   $varNum                 # in
        rVAR_CACHESIZE_         $numBuffers             # in
        rVAR_NAME_              $varName                # in
        rVAR_RESERVEPERCENT_    $percent                # in
        rVAR_SEQPOS_            $recNum                 # in
                                \@indices               # in
        rVARs_CACHESIZE_        $numBuffers             # in
        rVARs_DIMCOUNTS_        \@counts                # in
        rVARs_DIMINDICES_       \@indices               # in
        rVARs_DIMINTERVALS_     \@intervals             # in
        rVARs_RECCOUNT_         $recCount               # in
        rVARs_RECINTERVAL_      $recInterval            # in
        rVARs_RECNUMBER_        $recNum                 # in
```

STAGE_CACHESIZE_	$numBuffers	#	in
zENTRY_	$entryNum	#	in
zENTRY_NAME_	$varName	#	in
zVAR_	$varNum	#	in
zVAR_CACHESIZE_	$numBuffers	#	in
zVAR_DIMCOUNTS_	\@counts	#	in
zVAR_DIMINDICES_	\@indices	#	in
zVAR_DIMINTERVALS_	\@intervals	#	in
zVAR_NAME_	$varName	#	in
zVAR_RECCOUNT_	$recCount	#	in
zVAR_RECINTERVAL_	$recInterval	#	in
zVAR_RECNUMBER_	$recNum	#	in
zVAR_RESERVEPERCENT_	$percent	#	in
zVAR_SEQPOS_	$recNum	#	in
	\@indices	#	in
zVARs_CACHESIZE_	$numBuffers	#	in
zVARs_RECNUMBER_	$recNum	#	in

B.3 Quick Interface

($status, %info) = CDF::CDFgetLIBInfo ()

($status, %info) = CDF::CDFgetCDFInfo ([cdfid | cdfName])

($status, %info) = CDF::CDFgetVarInfo ([cdfid | cdfName], [varName | varid])

($status, %meta) = CDF::CDFgetGlobalMetaData ([cdfid | cdfName] [, globalNames | globalids] [, metaEncoding])

($status, %meta) = CDF::CDFgetVarMetaData ([cdfid | cdfName], [varName | varid] [, metaEncoding])

($status, @data) = CDF::CDFgetVarAllData ([cdfid | cdfname], [varName | varid] [, dataEncoding] [, matrix])

B.4 EPOCH Utility Routines

```
$epoch = CDF::computeEPOCH ($year, $month, $day, $hour, $minute, $second, $msec)
my $year;                                                                    # in
my $month;                                                                   # in
my $day;                                                                     # in
my $hour;                                                                    # in
my $minute;                                                                  # in
my $second;                                                                  # in
my $msec;                                                                    # in

@epoch = CDF::computeEPOCH (\@year, \@month, \@day, \@hour, \@minute, \@second, \@msec)
my \@year;                                                                   # in
my \@month;                                                                  # in
my \@day;                                                                    # in
my \@hour;                                                                   # in
my \@minute;                                                                 # in
my \@second;                                                                 # in
my \@msec;                                                                   # in

CDF::EPOCHbreakdown ($epoch, $year, $month, $day, $hour, $minute, $second, $msec)
my $epoch;                                                                   # in
my $year;                                                                    # out
my $month;                                                                   # out
my $day;                                                                     # out
my $hour;                                                                    # out
my $minute;                                                                  # out
my $second;                                                                  # out
my $msec;                                                                    # out

CDF::EPOCHbreakdown (\@epoch, \@year, \@month, \@day, \@hour, \@minute, \@second, \@msec)
my \@epoch;                                                                  # in
my \@year;                                                                   # out
my \@month;                                                                  # out
my \@day;                                                                    # out
my \@hour;                                                                   # out
my \@minute;                                                                 # out
my \@second;                                                                 # out
my \@msec;                                                                   # out

CDF::toEncodeEPOCH ($epoch, $style, $epString)
my \@epoch;                                                                  # in
my $style;                                                                   # in
my $epString;                                                                # out

CDF::toEncodeEPOCH (\@epoch, $style, \@epString)
my \@epoch;                                                                  # in
my $style;                                                                   # in
my \@epString;                                                               # out

CDF::encodeEPOCH ($epoch, $epString)
my $epoch;                                                                   # in
my $epString;                                                                # out

CDF::encodeEPOCH (\@epoch, \@epString)
```

```
my  \@epoch;                                                                    #  in
my  \@epString;                                                                 #  out

CDF::encodeEPOCH1  ($epoch,  $epString)
my  $epoch;                                                                     #  in
my  $epString;                                                                  #  out

CDF::encodeEPOCH2  ($epoch,  $epString)
my  $epoch;                                                                     #  in
my  $epString;                                                                  #  out

CDF::encodeEPOCH3  ($epoch,  $epString)
my  $epoch;                                                                     #  in
my  $epString;                                                                  #  out

CDF::encodeEPOCH4  ($epoch,  $epString)
my  $epoch;                                                                     #  in
my  $epString;                                                                  #  out

CDF::encodeEPOCHx  ($epoch,  $format,  $epString)
my  $epoch;                                                                     #  in
my  $format;                                                                    #  in
my  $epString;                                                                  #  out

$epoch = CDF::toParseEPOCH  ($epString)                                         #  out
my  $epString;                                                                  #  in

@epoch = CDF::toParseEPOCH  (\@epString)                                        #  out
my  \@epString;                                                                 #  in

$epoch = CDF::parseEPOCH  ($epString)                                           #  out
my  $epString;                                                                  #  in

@epoch = CDF::parseEPOCH  (\@epString)                                          #  out
my  \@epString;                                                                 #  in

$epoch = CDF::parseEPOCH1  ($epString)                                          #  out
my  $epString                                                                   #  in

$epoch = CDF::parseEPOCH2  ($epString)                                          #  out
my  $epString;                                                                  #  in

$epoch = CDF::parseEPOCH3  ($epString)                                          #  out
my  $epString;                                                                  #  in

$epoch = CDF::parseEPOCH4  ($epString)                                          #  out
my  $epString;                                                                  #  in

$status = CDF:: computeEPOCH16  ($year,  $month,  $day,  $hour,  $minute,  $second,  $msec, $microsec,
                                  $nanosec, $picosec. \@epoch)                  #  out
my  $year;                                                                      #  in
my  $month;                                                                     #  in
my  $day;                                                                       #  in
my  $hour;                                                                      #  in
my  $minute;                                                                    #  in
my  $second;                                                                    #  in
```

```
my  $msec;                                                          #  in
my  $microsec;                                                      #  in
my  $nanosec;                                                       #  in
my  $picosec;                                                       #  in
my  \@epoch;                                                        #  out

CDF::EPOCH16breakdown  (\@epoch, $year, $month, $day, $hour, $minute,  $second,  $msec,
                        $microsec, $nanosec, $picosec)
my  \@epoch;                                                        #  in
my  $year;                                                          #  out
my  $month;                                                         #  out
my  $day;                                                           #  out
my  $hour;                                                          #  out
my  $minute;                                                        #  out
my  $second;                                                        #  out
my  $msec;                                                          #  out
my  $microsec;                                                      #  out
my  $nanosec;                                                       #  out
my  $picosec;                                                       #  out

CDF::toEncodeEPOCH16  (\@epoch,  style, epString)
my  \@epoch;                                                        #  in
my  $style;                                                         #  in
my  $epString;                                                      #  out

CDF::encodeEPOCH16  (\@epoch,  epString)
my  \@epoch;                                                        #  in
my  $epString;                                                      #  out

CDF::encodeEPOCH16_1  (\@epoch,  epString)
my  \@epoch;                                                        #  in
my  \$epString;                                                     #  out

CDF::encodeEPOCH16_2  (\@epoch,  epString)
my  \@epoch;                                                        #  in
my  \$epString;                                                     #  out

CDF::encodeEPOCH16_3  (\@epoch,  epString)
my  \@epoch;                                                        #  in
my  \$epString;                                                     #  out

CDF::encodeEPOCH16_4  (\@epoch,  epString)
my  \@epoch;                                                        #  in
my  \$epString;                                                     #  out

CDF::encodeEPOCH16_x  (\@epoch,  format,  epString)
my  \@epoch;                                                        #  in
my  $format;                                                        #  in
my  $epString;                                                      #  out

CDF:: toParseEPOCH16  ($epString, \@epoch)
my  $epString;                                                      #  in
my  \@epoch;                                                        #  out

CDF:: parseEPOCH16  ($epString, \@epoch)
my  $epString;                                                      #  in
```

```
my  \@epoch;                                                    #  out

$status = CDF::parseEPOCH16_1  ($epString, \@epoch)             #  out
my  $epString;                                                  #  in
my  \@epoch;                                                    #  out

$status = CDF::parseEPOCH16_2  ($epString, \@epoch)             #  out
my  $epString;                                                  #  in
my  \@epoch;                                                    #  out

$status = CDF::parseEPOCH16_3  ($epString, \@epoch)             #  out
my  $epString;                                                  #  in
my  \@epoch;                                                    #  out

$status = CDF::parseEPOCH16_4  ($epString, \@epoch)             #  out
my  $epString;                                                  #  in
my  \@epoch;                                                    #  out

CDF::EPOCHtoUnixTime  ($epoch, $unixTime)
my  $epoch;                                                     #  in
my  $unixTime;                                                  #  out

CDF::EPOCHtoUnixTime  (\@epoch, \@unixTime)
my  \@epoch;                                                    #  in
my  \@unixTime;                                                 #  out

CDF::EPOCH16toUnixTime  (\@epoch, \$unixTime)
my  \@epoch;                                                    #  in
my  \$unixTime;                                                 #  out

CDF::EPOCH16toUnixTime  (\@epoch, \@unixTime)
my  \@epoch;                                                    #  in
my  \@unixTime;                                                 #  out

CDF::UnixTimetoEPOCH  ($unixTime, $epoch)
my  $unixTime;                                                  #  in
my  $epoch;                                                     #  out

CDF:: UnixTimetoEPOCH  (\@unixTime, \@epoch)
my  \@unixTime;                                                 #  in
my  \@epoch;                                                    #  out

CDF:: UnixTimetoEPOCH16  ($unixTime, \@epoch)
my  \$unixTime;                                                 #  in
my  \@epoch;                                                    #  out

CDF:: UnixTimetoEPOCH16  (\@unixTime, \@epoch)
my  \@unixTime;                                                 #  in
my  \@epoch;                                                    #  out
```

B.5 TT2000 Utility Routines

```
$tt2000 = CDF::computeTT2000 ($year, $month, $day, $hour, $minute, $second, $msec, $usec, $nansec)
my $year;                                                                    # in
my $month;                                                                   # in
my $day;                                                                     # in
my $hour;                                                                    # in
my $minute;                                                                  # in
my $second;                                                                  # in
my $msec;                                                                    # in
my $usec;                                                                    # in
my $nsec;                                                                    # in

@tt2000 = CDF::computeTT2000 (\@year, \@month, \@day, \@hour, \@minute, \@second, \@msec, \@usec,
\@nansec)
my \@year;                                                                   # in
my \@month;                                                                  # in
my \@day;                                                                    # in
my \@hour;                                                                   # in
my \@minute;                                                                 # in
my \@second;                                                                 # in
my \@msec;                                                                   # in
my \@usec;                                                                   # in
my \@nsec;                                                                   # in

CDF::TT2000breakdown ($tt2000, $year, $month, $day, $hour, $minute, $second, $msec, $usec, $nsec)
my $tt2000;                                                                  # in
my $year;                                                                    # out
my $month;                                                                   # out
my $day;                                                                     # out
my $hour;                                                                    # out
my $minute;                                                                  # out
my $second;                                                                  # out
my $msec;                                                                    # out
my $usec;                                                                    # out
my $nsec;                                                                    # out

CDF::TT2000breakdown (\@tt2000, \@year, \@month, \@day, \@hour, \@minute, \@second, \@msec, \@usec,
\@nsec)
my \@tt2000;                                                                 # in
my \@year;                                                                   # out
my \@month;                                                                  # out
my \@day;                                                                    # out
my \@hour;                                                                   # out
my \@minute;                                                                 # out
my \@second;                                                                 # out
my \@msec;                                                                   # out
my \@usec;                                                                   # out
my \@nsec;                                                                   # out

CDF::toEncodeTT2000 ($tt2000, $style, $epString)
my $tt2000;                                                                  # in
my $style;                                                                   # in
my $epString;                                                                # out
```

```
CDF::toEncodeTT2000  (\@tt2000,  $style, \@epString)
my  \@tt2000;                                                    #  in
my  $style;                                                      #  in
my  \@epString;                                                  #  out

CDF::encodeTT2000  ($tt2000,  $epString, $format[28])
my  $tt2000;                                                     #  in
my  $epString;                                                   #  out
my  $format;                                                     #  in

CDF::encodeTT2000  (\@tt2000,  \@epString, $format[29])
my  \@tt2000;                                                    #  in
my  \@epString;                                                  #  out
my  format;                                                      #  in

$epoch = CDF::toParseTT2000  ($epString)                         #  out
my  $epString;                                                   #  in

@epoch = CDF::toParseTT2000  (\@epString)                        #  out
my  \@epString;                                                  #  in

$epoch = CDF::parseTT2000  ($epString)                           #  out
my  $epString;                                                   #  in

@epoch = CDF::parseTT2000  (\@epString)                          #  out
my  \@epString;                                                  #  in

CDF::TT2000toUnixTime  ($epoch, $unixTime)
my  $epoch;                                                      #  in
my  $unixTime;                                                   #  out

CDF::TT2000toUnixTime  (\@epoch, \@unixTime)
my  \@epoch;                                                     #  in
my  \@unixTime;                                                  #  out

CDF::UnixTimetoTT2000  ($unixTime, $epoch)
my  $unixTime;                                                   #  in
my  $epoch;                                                      #  out

CDF::UnixTimetoTT2000  (\@unixTime, \@epoch)
my  \@unixTime;                                                  #  in
my  \@epoch;                                                     #  out

CDF::leapsecondsinfo  ($dump[30])
my  $dump;                                                       #  in
```

[28] An optional field.
[29] An optional field.
[30] An optional field.

Index